Васильев А. Н.

Python на примерах

Практический курс по программированию

НиТ
Издательство

Васильев А. Н.

Python

НА ПРИМЕРАХ

Практический курс по программированию

Наука и Техника
Санкт-Петербург
2016

Васильев А. Н.

Python на примерах. Практический курс по программированию. – СПб.: Наука и Техника, 2016. – 432 с.: ил.

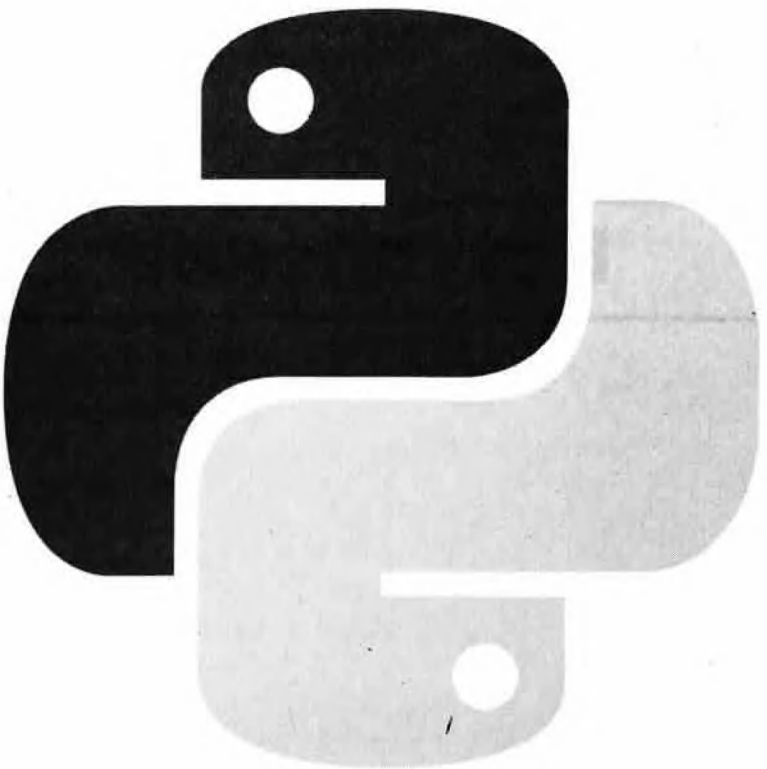
Серия "Просто о сложном"

В этой книге речь будет идти о том, как писать программы на языке программирования, который называется Python (правильно читается как пайтон, но обычно название языка читают как питон, что тоже вполне приемлемо). Таким образом, решать будем две задачи, одна из которых приоритетная, а вторая, хотя и вспомогательная, но достаточно важная. Наша основная задача, конечно же, изучение синтаксиса языка программирования Python. Параллельно мы будем осваивать программирование как таковое, явно или неявно принимая во внимание, что соответствующие алгоритмы предполагается реализовывать на языке Python.

Большинство авторов книг в своих трудах рассматривают теоретические основы языка и уделяют основное внимание базовому синтаксису языка, не рассматривая при этом практическую сторону его применения. Эта же книга старается восполнить недостаток практического материала, содержит множество примеров с комментариями, которые вы сможете использовать в качестве основы своих программных решений, изучения Python.

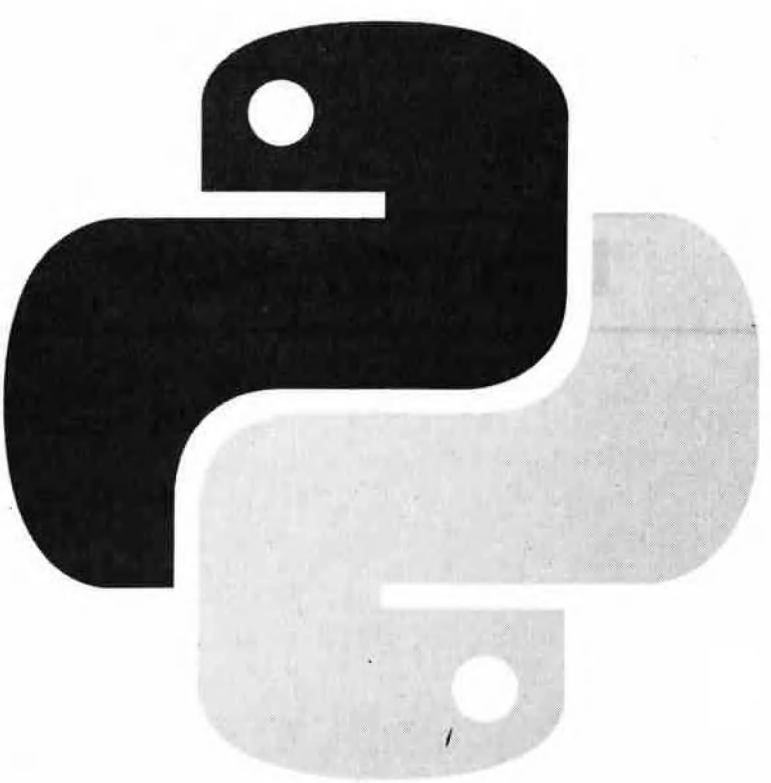
Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений.

Контактные телефоны издательства:
(812) 412 70 25, (812) 412 70 26, (044) 516 38 66
Официальный сайт: www.nit.com.ru



Оглавление

Содержание



Вступление

Знакомство с Python

Во всем есть своя мораль, нужно только уметь ее найти!

Л. Кэрролл "Алиса в стране чудес"

В этой книге речь будет идти о том, как писать программы на языке программирования, который называется Python (правильно читается как пайтон, но обычно название языка читают как питон, что тоже вполне приемлемо). Таким образом, решать будем две задачи, одна из которых приоритетная, а вторая, хотя и вспомогательная, но достаточно важная. Наша основная задача - конечно же, изучение синтаксиса языка программирования Python. Параллельно мы будем осваивать азы программирования, явно или неявно принимая во внимание, что соответствующие алгоритмы предполагается реализовывать на языке Python.

На заметку

Даже если у читателя есть опыт программирования на других языках, не следует относиться поверхностно или пренебрежительно к процессу построения алгоритма программы. Правила хорошего тона в программировании подразумевают, что написание программы начинается задолго до набора программного кода. Хорошо взять лист бумаги и набросать общую схему того, как будет выполняться программа. А для этого процедуру решения большой и сложной задачи следует разбить на последовательность простых действий. С одной стороны данный процесс универсальный. С другой стороны, те задачи, которые мы назвали выше "простыми", решаются посредством базовых команд или функций языка программирования, на котором предполагается составлять программу. Поэтому обдумывая алгоритм, полностью абстрагироваться от конкретных возможностей языка программирования вряд ли удастся. Учитывая же гибкость и эффективность языка программирования Python, следует признать, что алгоритмы даже для "классических" задач при реализации на Python становятся проще и понятнее. Другими словами, даже если читатель имеет опыт составления алгоритмов, знакомство с языком программирования Python позволит ему посмотреть на многие знакомые вещи совершенно по-иному.

Вообще, языков программирования довольно много. Более того, время от времени появляются новые языки. Поэтому естественным образом возникает вопрос: почему именно Python? Наш ответ будет состоять из нескольких пунктов.

- Язык программирования Python - язык высокого уровня, достаточно "молодой", но очень популярный, который уже сейчас широко ис-

пользуется на практике и сфера применения Python постоянно расширяется.

На заметку

По отношению к языкам программирования нередко применяют такие выражения, как язык *высокого уровня*, язык *среднего уровня* или язык *низкого уровня*. Эта классификация достаточно условная и базируется на уровне абстракции языка. Ведь язык, на котором разговаривают люди, немного отличается от того "языка", который "понимают" компьютеры. Команда, написанная на простом человеческом языке, будет совершенно неприемлемой для компьютера. Команда, готовая к исполнению компьютером (*машинный* код), будет непонятной для большинства простых смертных. Поэтому выбирать приходится между Сциллой и Харибдой. Про языки, ориентированные на программиста, говорят, что это языки высокого уровня. Про языки, ориентированные на компьютер, говорят, что это языки низкого уровня. Промежуточная группа языков называется языками среднего уровня. Хотя еще раз подчеркнем, что деление это достаточно условное.

- Синтаксис языка Python минималистический и гибкий. На этом языке можно составлять простые и эффективные программы.
- Стандартная библиотека для этого языка содержит множество полезных функций, что значительно облегчает процесс создания программных кодов.
- Язык Python поддерживает несколько парадигм программирования, включая структурное, объектно-ориентированное и функциональное программирование. И это далеко не полный список.
- Язык Python вполне удачный выбор для первого языка при обучении программированию.

Существуют и иные причины к тому, чтобы выучить язык программирования Python - возможно, даже весомее тех, что перечислены выше. О некоторых мы еще будем говорить (в контексте особенностей языка программирования Python). Во всяком случае, здесь будем исходить из того, что читатель для себя решение об изучении языка Python принял, или, по крайней мере, проявляет интерес к этому языку программирования.

На заметку

Парадигма программирования - это наиболее общая концепция, которая определяет фундаментальные характеристики и базовые методы реализации программных кодов. Например, парадигма *объектно-ориентированного* программирования (сокращенно ООП) подразумевает, что программа реализуется через набор взаимодействующих объектов, которые, в свою очередь, обычно создаются на основе классов. В рамках *структурного* программирования программа представляет собой комбинацию данных и процедур (функций) для их обработки. Язык может поддерживать сразу несколько парадигм. Так, языки Java и C# полностью объектно-

ориентированные, поэтому для написания самой маленькой программы на этих языках придется описать как минимум один класс. В языке С поддерживается парадигма структурного программирования, поэтому классов и объектов в языке С нет. Зато они есть в языке C++. Последний поддерживает как парадигму объектно-ориентированного программирования, так и парадигму структурного программирования. Как следствие, при работе с языком C++ классы и объекты можно использовать, а можно и не использовать - в зависимости от потребностей программиста и специфики решаемой задачи. Это же замечание относится к языку Python: с одной стороны, при написании программы на языке Python у нас имеется возможность прибегнуть к мощному арсеналу объектно-ориентированного программирования, а с другой стороны, часто бывают приемлемыми и методы структурного программирования.

Существуют и другие, более утонченные концепции программирования. Скажем, парадигма функционального программирования подразумевает, что результат функции в программе определяется исключительно значениями аргументов, переданных функции, и не зависит от состояния внешних (по отношению к функции) переменных. Соответствующие функции принято называть *чистыми функциями*, и они обладают рядом полезных свойств, позволяющих существенно оптимизировать и ускорить вычислительный процесс. Эта концепция, как и ряд других, находят реализацию в языке Python.

Далее обсудим некоторые важные моменты и "подводные камни", которые могут встретиться на, местами трудном, но все же интересном и увлекательном пути освоения новых вершин в программировании.

Краткая история и особенности языка Python

Серьёзное отношение к чему бы то ни было в этом мире является роковой ошибкой.

Л. Кэрролл "Алиса в стране чудес"

У языка Python есть автор - *Гвидо ван Россум* (Guido van Rossum). И хотя в разработке и популяризации языка на данный момент успело поучаствовать много талантливых разработчиков, именно Гвидо ван Россум пожинает заслуженные лавры создателя этого перспективного и популярного языка программирования. Вообще же работа над языком началась в 80-х годах прошлого столетия. Считается, что первая версия языка появилась в 1991 году. Касательно названия языка программирования Python, то формально это название рептилии. Соответственно, обычно в качестве логотипа используется милая (или не очень) змеючка типа "питон". И хотя практиче-

ски любое учебное или справочное пособие по языку Python содержит повествование о том, что на самом деле Python - это не "питон", а название юмористической передачи "Летающий цирк Монти Пайтона", для истории это уже не важно.

На заметку

Уместно вспомнить слова капитана Врунгеля: "Как вы яхту назовете, так она и поплывет". У языка программирования Python достаточно агрессивное название, и, надо признать, это название себя оправдывает. Аргументом к такому утверждению может быть как гибкость и эффективность самого языка, так и та быстрота, с которой он завоевал себе "место под солнцем" среди наиболее популярных языков программирования.

Язык Python бурно развивается. Этому способствует не только достаточно удачная концепция языка, но также сформировавшееся сплоченное сообщество разработчиков и популяризаторов языка. Немаловажен и тот факт, что необходимое программное обеспечение, включая среды разработки, в основной своей массе бесплатны. Все это дает основания рассматривать Python в качестве одного из наиболее перспективных языков программирования.

На сегодняшний день Python используется при разработке самых различных проектов, среди которых:

- разработка сценариев для работы с Web и Internet-приложений;
- сетевое программирование;
- средства поддержка технологий HTML и XML;
- приложения для работы с электронной почтой и поддержки Internet-протоколов;
- приложения для обслуживания всевозможных баз данных;
- программы для научных расчетов;
- приложения с графическим интерфейсом;
- создание игр и компьютерной графики,

и многое другое. Разумеется, охватить все эти темы в одной книге достаточно сложно. Да мы и не ставим перед собой такой цели. Тем не менее, даже на относительно небольшом количестве несложных примеров вполне можно продемонстрировать элегантность и исключительную эффективность языка Python. Этим, собственно, мы и займемся в основной части книги - то

есть немного позже. Сейчас обсудим особенности и некоторые "технические" моменты, которые важны для понимания основ программирования на Python.

Язык Python относится к *интерпретируемым* языкам программирования, что имеет определенные последствия. Формально то, что язык программирования интерпретируемый, означает, что программный код выполняется с помощью специальной программы. Программа называется *интерпретатором*. Интерпретатор выполняет программный код построчно (с предварительным анализом выполняемого кода). Недостаток такого подхода состоит в том, что, во-первых, ошибки выявляются фактически на этапе выполнения программы и, во-вторых, скорость выполнения программы относительно невысокая. Поэтому нередко используется более сложная схема: исходный программный код компилируется в промежуточный код, а уже этот промежуточный код выполняется непосредственно интерпретатором. В этом случае скорость выполнения программы увеличивается, но вместе с ней увеличивается и запрос на системные ресурсы. Примерно по такой схеме выполняется программный код, написанный на языке Python.

На заметку

Напомним, что помимо интерпретируемых, существуют *компилируемые* языки (имеется в виду компиляция в машинный код). В этом случае исходный программный код компилируется в исполнительный (машинный) код. Исполнительный код выполняется (обычно) под управлением операционной системы. Компилируемые в исполнительный код программы характеризуются относительно высокой скоростью выполнения.

Если мы хотим написать программу на Python, то для этого, как минимум, понадобится набрать соответствующий программный код. Здесь возможны варианты, но в принципе код набирается в любом текстовом редакторе, а соответствующий файл сохраняется с расширением *py* или *pyw* (для программ с графическим интерфейсом). При первом запуске (после внесения изменений в программный код) создается промежуточный код, который записывается в файл с расширением *pyc*. Если после этого в программу изменения не вносились, то при выполнении программы будет выполняться соответствующий *pyc*-файл. После внесения изменений в программу она при очередном запуске перекомпилируется в *pyc*-файл. Это общая схема. Нам, на самом деле, она интересна исключительно в плане общего развития, хотя на практике иногда бывает важно учитывать особенности выполнения программы, написанной на языке Python.

Как отмечалось выше, программный код можем набирать хоть в текстовом редакторе. Но вот для выполнения такого программного кода понадобит-

ся специальная программа, которая называется *интерпретатором*. Другими словами, для работы с Python на компьютер необходимо установить программу-интерпретатор. Мы отдельно обсудим этот вопрос. Сейчас же только заметим, что обычно используется не просто интерпретатор, а *интегрированная среда разработки*, которая, кроме прочего, включает в себя как интерпретатор, так и редактор программных кодов.

Интерпретатор выполняет программу команда за командой. Поэтому в принципе, если программа состоит из нескольких команд, ее можно организовать в виде файла с программным кодом, а затем "отправить" этот файл на выполнение. Еще один вариант - "передавать" интерпретатору для выполнения по одной команде. И тот, и другой режимы возможны и поддерживаются интерпретатором языка Python. Мы будем составлять и запоминать программные коды в файлах - то есть используем "традиционный" подход.

На заметку

Про режим, при котором в командном окне интерпретатора команды вводятся и выполняются одна за другой, говорят, как о *режиме командной строки* или *режиме калькулятора*.

Поскольку язык Python развивается интенсивно и от версии к версии в синтаксис и структуру языка вносятся изменения, важно учитывать, для какой версии языка Python составляется (и предполагается использовать) программный код. Особенно это важно с учетом того обстоятельства, что при внесении изменений *принцип обратной совместимости* работает далеко не всегда: если программный код корректно выполняется в более старых версиях языка, то совсем не факт, что будет выполняться при работе с более новыми версиями. Однако паниковать по этому поводу не стоит. Обычно проблема несовместимости кодов для разных версий связана с особенностями синтаксиса некоторых функций или конструкций языка и достаточно легко устраняется.

На заметку

На момент написания книги актуальной является версия Python 3.3. Именно она использовалась для тестирования примеров в книге. При появлении новых версий или стандартов языка, для обеспечения совместимости программных кодов, имеет смысл просмотреть тот раздел справочной системы, в котором описываются нововведения. Сделать это можно, например, на официальном сайте поддержки языка Python *www.python.org/doc/* в разделе с названием *What's New In Python* (в переводе означает *Что нового в Python*).

Но все это технические детали, которые хоть и важны, но все же не первостепенны. А первостепенными для нас будут синтаксис языка Python и его основные управляющие инструкции. И собственно здесь нас ждет много приятных сюрпризов.

На заметку

Особенно много сюрпризов будет для читателей, которые знакомы с такими языками программирования, как Java или C++, например. Но это, кстати, совсем не означает, что новичков в программировании язык Python оставит равнодушными. Просто те, кто изучал основы ООП (объектно-ориентированное программирование) и программирует на упомянутых языках, значительно расширят свой кругозор в плане методов и приемов программирования, а также в некотором смысле им предстоит изменить свое представление о языках программирования.

Синтаксис языка Python более чем интересен. Во-первых, он прост, понятен и нагляден. В некотором смысле его можно даже назвать по-спартански лаконичным. Одновременно с этим программные коды, написанные на Python, обычно легко читаются и анализируются, а объем программного кода намного меньше, если сравнивать с аналогичными программами, написанными на других языках программирования. В качестве иллюстрации в листинге В.1 приведен код программы на языке C++, в результате выполнения которой в консольное окно выводится сообщение `Hello, world!`.

Листинг В.1. Программа на языке C++

```
#include <iostream>
using namespace std;
int main(){
cout<<"Hello, world!"<<endl;
return 0;
}
```

Аналогичный программный код, но уже на языке Java, представлен в листинге В.2.

Листинг В.2. Программа на языке Java

```
classMyClass{
public static void main(String[] args){
System.out.println ("Hello, world!");
}
}
```

Наконец, в листинге В.3 показано, как будет выглядеть программа, выводящая в консольное окно текстовое сообщение, если для ее написания воспользоваться языком программирования Python.

Листинг В.3. Программа на языке Python

```
print("Hello, world!")
```

Несложно заметить, что это всего одна команда, в которой встроенной функции `print()` аргументом передается текст, который необходимо напечатать в консольном окне. Разумеется, далеко не всегда у нас будет получаться писать такие "экономные" коды, но пример все же во многом показательный.

На всякий случай кратко прокомментируем представленные выше коды на языках C++ и Java - просто чтобы у читателя, не знакомого с этими языками, не появилось комплекса неполноценности. Начнем с программы из листинга В.1, написанной на языке C++:

- Инструкцией `#include <iostream>` подключается заголовочный файл библиотеки ввода/вывода.
- Команда `using namespace std` означает использование стандартного пространства имен.
- Функция с названием `main()` называется главной функцией программы. Выполнение программы в C++ - это выполнение главной функции программы.
- Идентификатор `int` слева от функции `main()` означает, что функция возвращает целочисленный результат.
- Пара фигурных скобок (`{` и `}`) выделяет тело главной функции.
- Командой `cout<<"Hello, world!"<<endl` в консоль выводится текстовое сообщение `Hello, world!` и выполняется переход к новой строке (инструкция `endl`).Оператор вывода << выводит текст, указанный слева от него, на устройство, определяемое идентификатором `cout` (по умолчанию - консоль).
- Инструкция `return 0` означает, что выполнение главной функции (то есть программы) завершено и в качестве результата функцией возвращается `0` (означает завершение работы программы "в штатном режиме" - то есть без ошибок).

Программа в листинге В.2, напомним, написана на языке Java и в соответствующем программном коде назначение инструкций следующее:

- Создается класс с названием `MyClass`: перед названием класса указывается ключевое слово `class`, а тело класса заключается в фигурные скобки (внешняя пара скобок `{` и `}`).
- В теле класса описывается главный метод с названием `main()`, тело метода заключается в блок из фигурных скобок (внутренняя пара скобок `{` и `}`).
- Перед названием главного метода указаны такие идентификаторы: `public` (открытый метод - то есть доступен вне класса), `static` (статический метод - для вызова метода нет необходимости создавать объект класса), `void` (метод не возвращает результат).
- Параметром (аргументом) метода `main()` указана переменная `args`, которая является текстовым массивом (текст - это объект класса `String`, а наличие пустых квадратных скобок `[]` свидетельствует о том, что это текстовый массив - упорядоченный набор текстовых значений).
- Текст в консольное окно выводится методом `println()`: аргументом методу передается выводимый в консоль текст, а сам метод вызывается из объекта потока вывода `out`, который, в свою очередь, является статическим полем класса `System`.

Разумеется, на фоне всего этого разнообразия программа (а точнее, однаединственная команда) на языке Python выглядит более чем эффектно. Но еще раз подчеркнем, что даже если читатель понял не все из изложенного выше (по поводу кодов C++ и Java) - это не страшно. Нам, в нашей дальнейшей работе, все это не пригодится. Мы просто хотели проиллюстрировать масштабы, так сказать, различий для разных языков.

При работе с языком Python мы не встретим многих привычных для других языков конструкций. Например, в отличие от языков C++ и Java, в которых командные блоки выделяются фигурными скобками `{` и `}`, и в отличие от языка Pascal, в котором блоки выделяются инструкциями `begin` и `end`, в языке Python блок команд выделяется отступом (рекомендуемый отступ - четыре пробела).

В Python нет необходимости заканчивать каждую команду точкой с запятой. Существуют и многие другие особенности языка Python. Мы будем знакомиться с ними постепенно, и, перефразируя *Михаила Евграфовича Салтыкова-Щедрина*, постараемся при этом не применять насилие.

На заметку

Имеется в виду цитата "Просвещение внедрять с умеренностью, по возможности избегая кровопролития" из сатирического романа "История одного города" авторства М.Е. Салтыкова-Щедрина.

Здесь просто важно понять, что язык Python достаточно своеобразный, самобытный и во многих отношениях не похож на прочие, популярные на сегодня, языки программирования.

Немного о книге

- Вот по этому поводу первый тост.
- Сейчас запишу.
- Потом запишешь.
из к/ф "Кавказская пленница"

Пришло самое время сказать (прочитать, написать - кому как больше нравится) несколько слов непосредственно о книге: что она собой представляет, для кого написана, и вообще, чего читателю ожидать от прочтения.

Разумеется, книга писалась в первую очередь для тех, кто решил освоить язык программирования Python. То есть естественным образом предполагается, что читатель с языком Python не знаком совсем. Более того, мы неявно будем исходить из того, что читатель вообще имеет минимальный опыт программирования. Последнее не помешает нам периодически ссылаться к таким языкам программирования, как C++ и Java. Конечно, делаться это будет в первую очередь в расчете на тех читателей, кто имеет хотя бы минимальное представление об этих языках и/или объектно-ориентированном программировании (сокращенно ООП).

Чтобы компенсировать неудобства, которые могли бы возникнуть у читателей, не знакомых с C++ и Java, пояснения максимально адаптируются для восприятия полностью неподготовленной аудиторией. Проще говоря, не имеет значения, знает читатель другие языки программирования или нет - в любом случае он может рассчитывать на успех.

Материал книги охватывает все основные темы, необходимые для успешной работы с Python, включая методы ООП. В основном мы будем рассматривать прикладные примеры - то есть теория будет представлена "на примерах". Это прием, который на практике неплохо себя зарекомендовал. Особенно он эффективен, когда необходимо в сжатые сроки с минималь-

ной затратой энергии и ресурсов освоить на качественном уровне большой объем материала. При таком подходе есть и дополнительный бонус: помимо особенностей языка читатель имеет возможность познакомиться с алгоритмами, которые применяются для решения ряда прикладных задач. Что касается подбора примеров и задач, то они выбирались в первую очередь так, чтобы наиболее ярко проиллюстрировать возможности и особенности языка Python.

Программное обеспечение

- Нет, ей об этом думать еще рано.
- Об этом думать никому не рано, и никогда не поздно.
из к/ф "Кавказская пленница"

Прежде, чем погрузиться в самые глубины мира под название Python и начать черпать бесценные знания, разумно внести ясность в вопрос о программном обеспечении, которое нам пригодится для тестирования примеров из книги и написания собственных оригинальных программ.

Как уже отмечалось выше, для выполнения программных кодов, написанных на Python, нам понадобится программа-интерпретатор. Но лучше всего воспользоваться какой-нибудь интегрированной средой разработки (сокращенно *IDE* от английского *Integrated Development Environment*). Среда разработки предоставляет пользователю не только интерпретатор, но и редактор кодов, равно как ряд других полезных утилит.

Интегрированных сред разработки для работы с Python существует достаточно много и в известном смысле перед программистом возникает непростая проблема выбора. Критерии для выбора среды разработки могут быть самыми разными. Но главные среди них, конечно же - это удобство в использовании, набор встроенных возможностей/функций интегрированной среды разработки, а также ее стоимость (существуют как коммерческие продукты, так и свободно распространяемые). Мы рассмотрим несколько наиболее популярных и доступных интегрированных сред разработки для Python.

Если мы говорим о программном обеспечении, то в первую очередь имеет смысл выйти на официальный сайт поддержки Python по адресу *www.python.org*. Окно браузера, открытое на соответствующей странице, показано на рис. В.1.

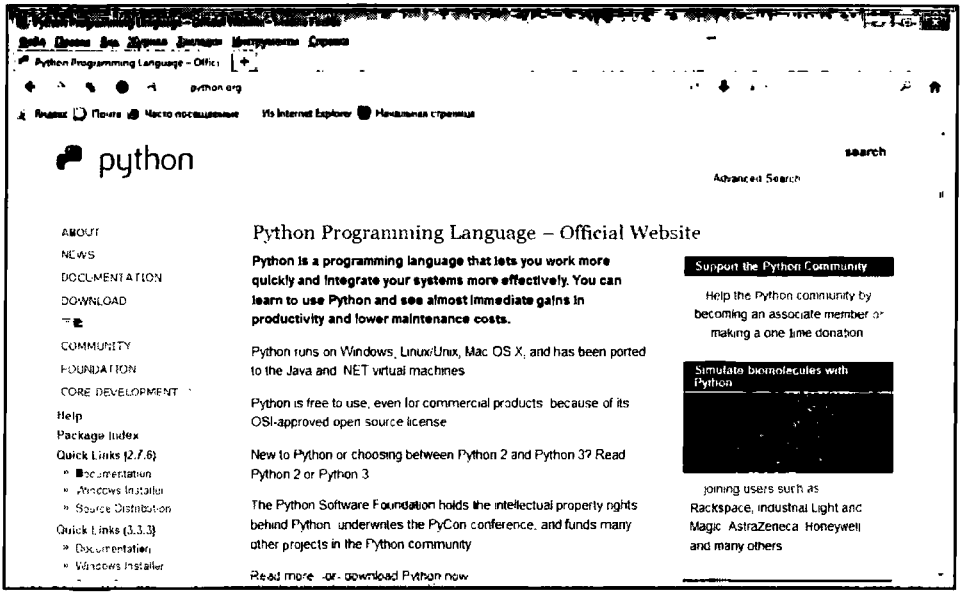


Рис. В.1. Окно браузера открыто на официальной странице поддержки Python www.python.org

Сайт содержит множество полезной информации, включая всеобъемлющую справку и, кроме прочего, позволяет загрузить необходимое программное обеспечение - в том числе и среду разработки, которая называется *IDLE* (сокращение от *Integrated DeveLopment Environment*, что буквально означает интегрированная среда разработки). Для загрузки программного обеспечения необходимо перейти к разделу *DOWNLOAD* (*загрузка*) - адрес *www.python.org/download*.

Сам процесс загрузки и установки достаточно простой и интуитивно понятный, поэтому останавливаться на нем не будем. Нас интересует конечный результат. А в результате мы получаем полноценную среду для работы с программными кодами Python. Рабочее окно среды IDLE представлено на рис. В.2.

Перед нами командная оболочка интерпретатора. Это окно с несколькими меню и большой рабочей областью, в которой после символа тройной стрелки >>> мигает курсор - это командная строка. В этом месте вводится команда, которая выполняется после нажатия клавиши <Enter>. Например, если мы хотим выполнить команду `print("Hello, world!")`, нам следует ввести эту команду в командной строке (то есть там, где мигает курсор - после символа >>>) и нажать клавишу <Enter>. Как следствие, коман-

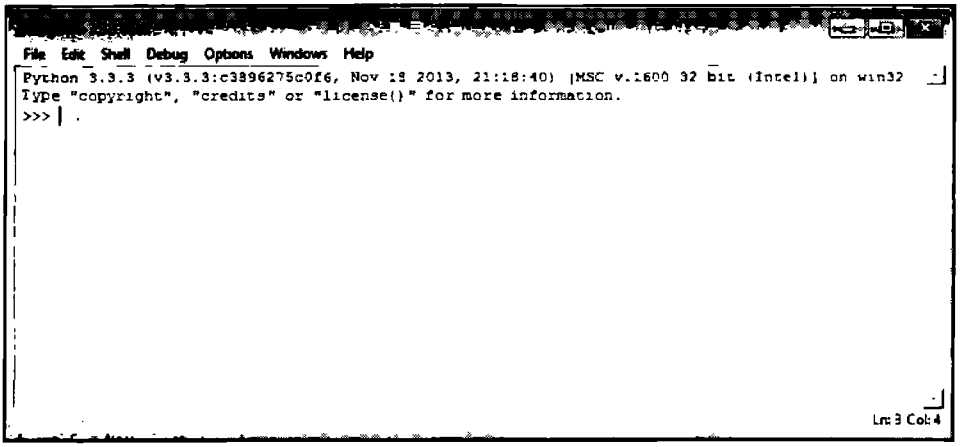


Рис. В.2. Рабочее окно среды разработки IDLE

да будет выполнена, а результат ее выполнения отображается снизу, под командной строкой. Ситуация проиллюстрирована на рис. В.3.

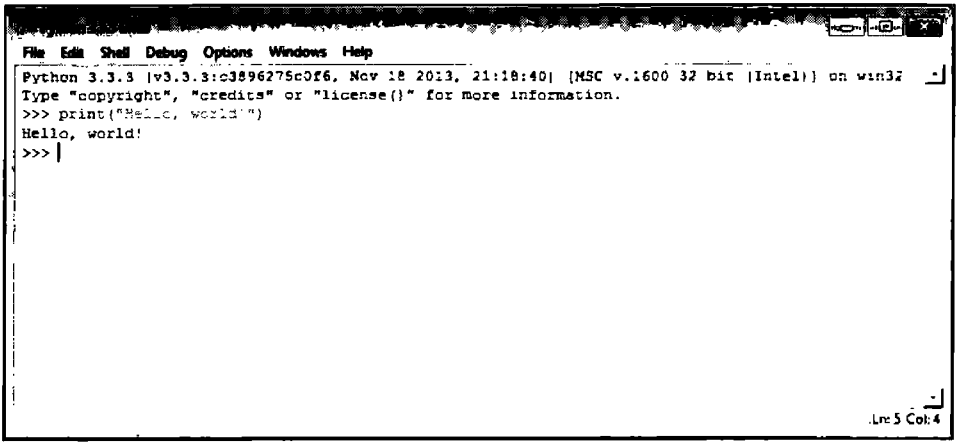


Рис. В.3. Результат выполнения команды в командной оболочке среды разработки IDLE

При этом под результатом выполнения команды снова появляется тройная стрелка >>> и в этом месте можно вводить новую команду. Например, можем ввести какое-нибудь алгебраическое выражение - скажем, пускай это будет `5+3*4`, как показано на рис. В.4.

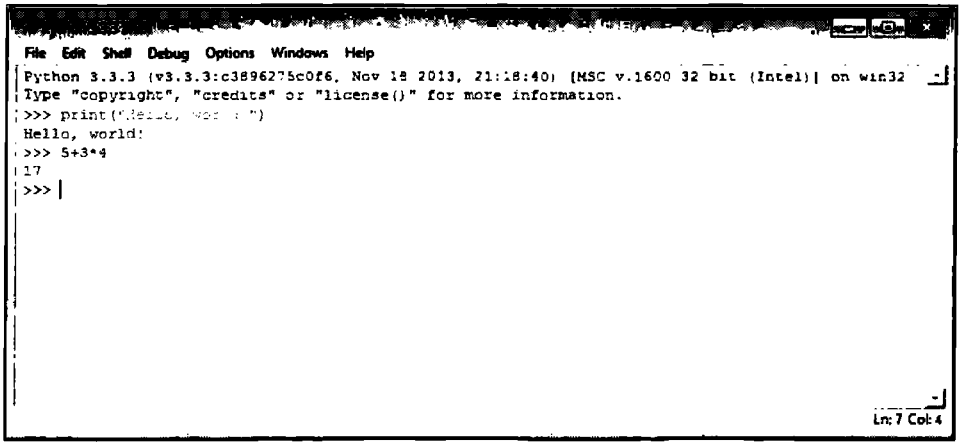


Рис. В.4. Результат вычисления алгебраического выражения

Разумеется, команды могут быть и более замысловатыми, равно как никто не запрещает нам команда за командой выполнять программный код. Но это достаточно неудобно. Обычно при написании более-менее серьезной программы ее оформляют в виде последовательности инструкций и записывают в отдельный файл. Затем соответствующая программа выполняется.

Создать файл программы можем с помощью все той же оболочки среды разработки. Если щелкнуть меню **File**, откроется список команд и подменю, среди которых имеется и команда **New File** (рис. В.5).

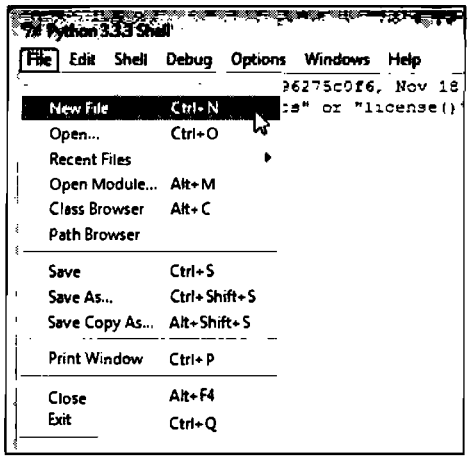


Рис. В.5. Создаем файл с программой

Сразу после выбора этой команды открывается редактор кодов, показанный на рис. В.6.

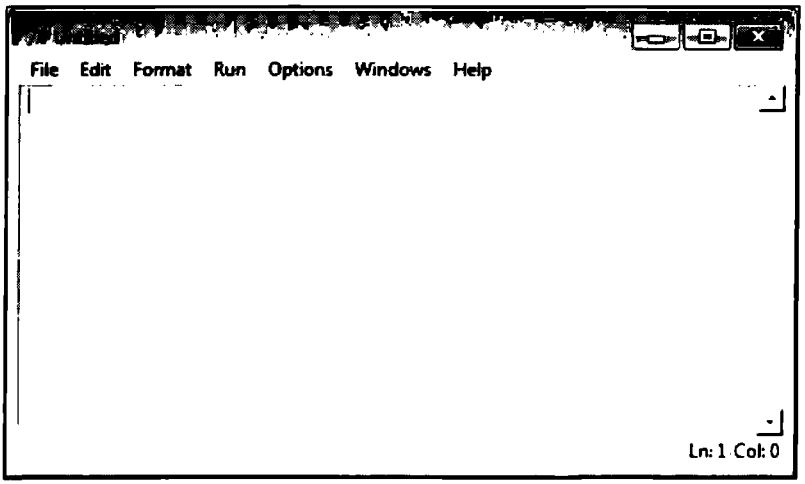


Рис. В.6. Редактор кодов используем для создания файла с программой

В окне редактора вводим программный код. В данном случае наша программа будет состоять всего из нескольких команд, которые приведены в листинге В.4.

Листинг В.4. Несколько команд для записи в файл

```
print("Начинаем вычисления!")
a=4
print("Значение переменной a=",a)
b=12
print("Значение переменной b=",b)
c=b/a
print("Результат деления b/a=",c)
print("Вычисления закончены!")
```

Именно такой программный код мы вводим в окне редактора кодов. Окно редактора с программным кодом показано на рис. В.7.

После того, как программный код набран, его можно сразу выполнить. Для этого в меню **Run** выбираем команду **Run Module**, как показано на рис. В.8.

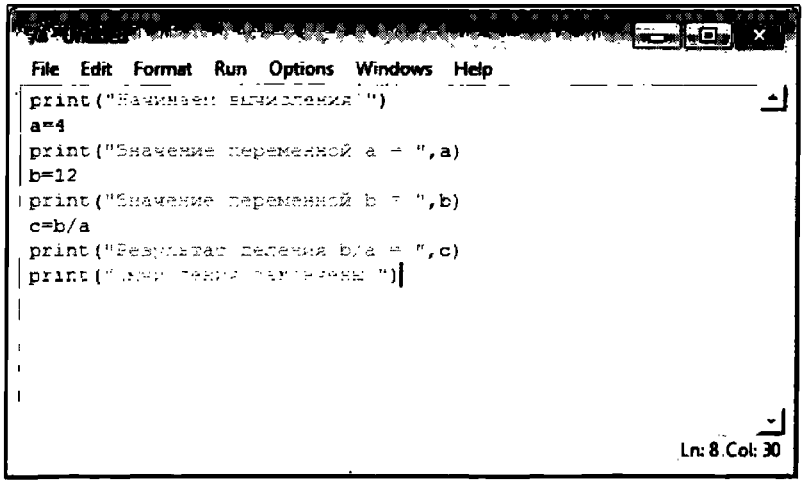


Рис. В.7. Окно редактора кодов с кодом программы

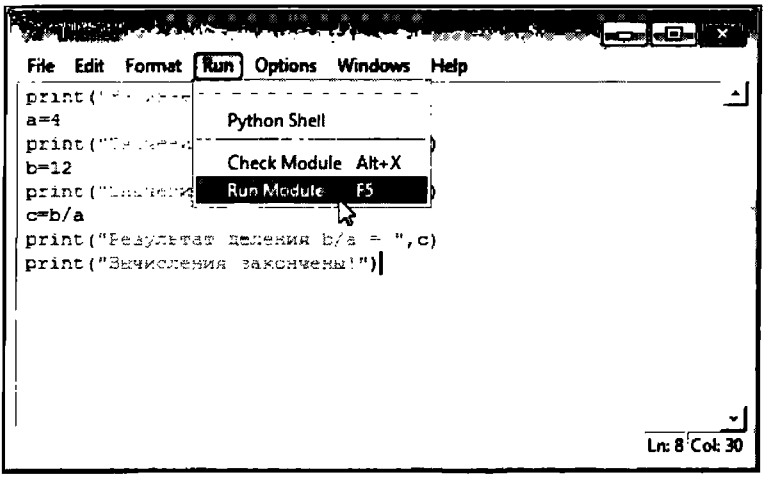


Рис. В.8. Запуск программы на выполнение

Правда, предварительно все же лучше сохранить файл с программой, для чего полезной будет команда **Save** из меню **File** (рис. В.9).

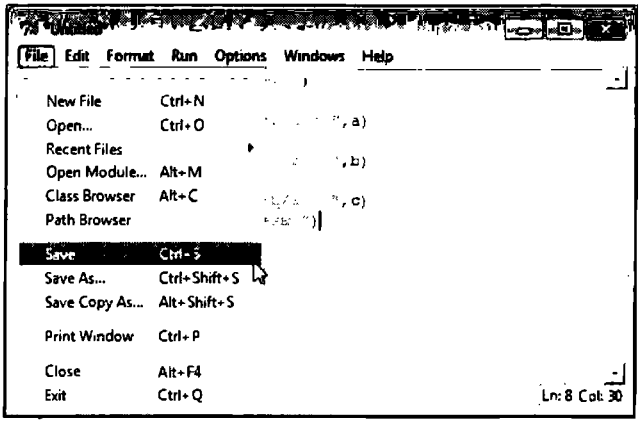


Рис. В.9. Сохранение файла с программой

На заметку

Если перед запуском программы на выполнение файл не сохранить, появится диалоговое окно с предложением сохранить файл. Поэтому лучше это сделать сразу.

Но как бы то ни было, в результате выполнения программы в окне командной оболочки появится результат, как на рис. В.10.

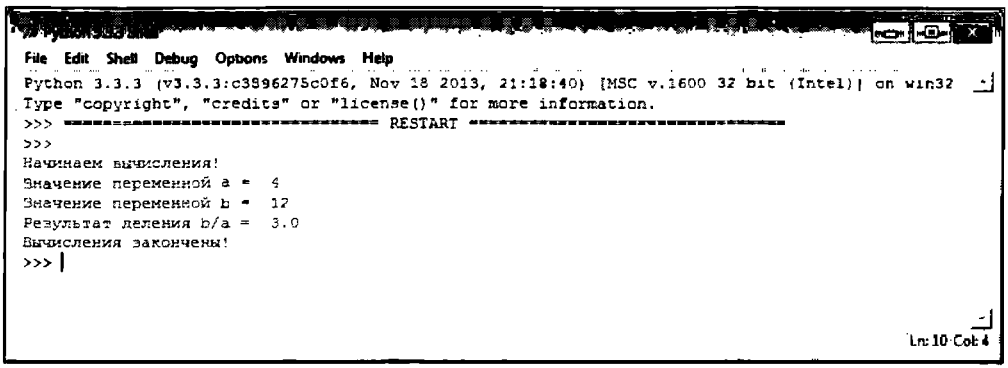


Рис. В.10. Результат выполнения программы

Как видим, в области вывода результатов (под символом >>>)появляется несколько сообщений, которые, очевидно, являются следствием выполнения программы, представленной в листинге В.4. Результат программы представлен ниже:

Результат выполнения программы (из листинга В.4)

```
Начинаем вычисления!
Значение переменной a =  4
Значение переменной b =  12
Результат деления b/a =  3.0
Вычисления закончены!
```

И хотя мы еще "официально" как бы не приступили к изучению языка Python, имеет смысл прокомментировать соответствующие команды и результат их выполнения.

Итак, командой `print("Начинаем вычисления!")` в начале выполнения программы выводится текстовое сообщение. Таким образом, в начале выполнения программы появляется текст `Начинаем вычисления!`. Аналогично, благодаря команде `print("Вычисления закончены!")`, которая завершает программный код, признаком окончания выполнения программы является текстовое сообщение `Вычисления закончены!`.

Между этими командами выполняются небольшие вычисления:

- командой `a=4` переменной `a` присваивается числовое значение `4`;
- командой `print("Значение переменной a = ",a)` выводится текст, а затем значение переменной `a`;
- командой `b=12` переменной `b` присваивается числовое значение `12`;
- командой `print("Значение переменной b = ",b)` выводится текст и значение переменной `b`;
- командой `c=b/a` переменной `c` в качестве значения присваивается результат деления значения переменной `b` на значение переменной `a`;
- командой `print("Результат деления b/a = ",c)` выводится текст и числовое значение переменной `c`.
- В справедливости этих утверждений читатель может убедиться, еще раз взглянув на рис. В.10.

На заметку

Наверняка пытливый читатель заметил, что переменные в программном коде использованы *без объявления их типа*. Другими словами мы нигде не указывали явно тип переменных, которые использовали в программе. Это стандартная ситуация для программ, написанных на Python - тип переменных указывать не нужно (он определяется автоматически по значению, которое присваивается переменной).

Также мы увидели, что функции `print()` можно передавать не только один, а несколько аргументов. В этом случае в область вывода (или консоль) последовательно, в одну строку, выводятся значения аргументов функции `print()`.

О том, как правильно составлять программные коды на Python, мы будем говорить в основной части книги. В данном случае нам важно лишь проиллюстрировать, что потом с этими программными кодами делать. Также нам важно дать читателю самое общее представление о тех приложениях и средах разработки, которые позволяют в удобном режиме составлять коды и запускать их на выполнение. Что касается кратко описанной выше среды IDLE, то назвать ее очень уж удачной вряд ли можно, хотя конечно, это субъективная точка зрения автора и читатель не обязан ее разделять.

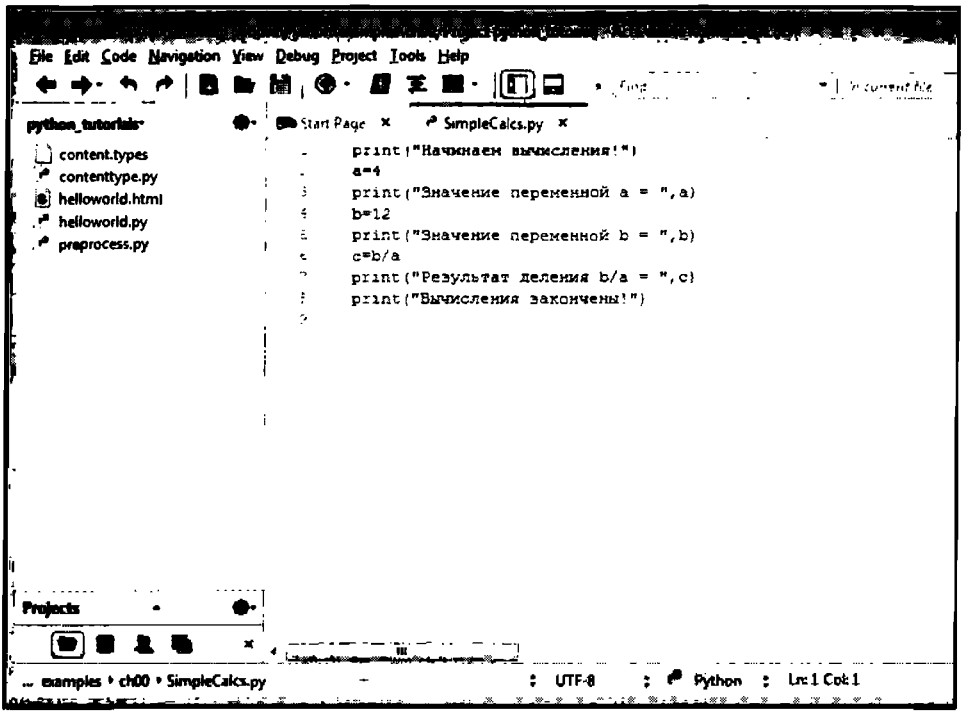


Рис. В.11. Окно интегрированной среды разработки Komodo IDE с программным кодом

Среди коммерческих продуктов можно выделить интегрированную среду разработки *Komodo IDE* (официальный сайт *www.activestate.com*). Окно приложения с открытым в нем файлом рассмотренной ранее программы показано на рис. В.11.

Результат выполнения программы в среде Komodo IDE показан на рис. В.12 (для запуска программы на выполнение можно воспользоваться командой **Run Without Debagging** из меню **Debug**).

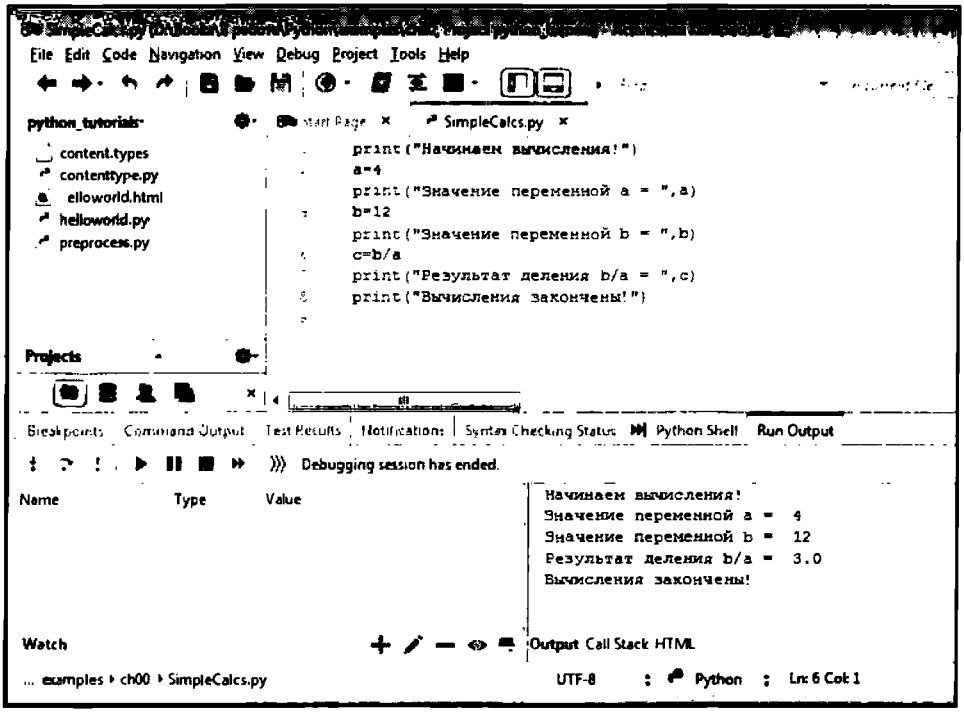


Рис. В.12. Результат выполнения программы в среде Komodo IDE

Мы, тем не менее, будем пользоваться для тестирования примеров из книги некоммерческую среду разработки *PyScripter*. Среда разработки PyScripter удобная, простая и бесплатная. Установочные файлы можно свободно загрузить на странице *https://code.google.com/p/pyscripter* в разделе `Downloads`. Окно браузера, открытое на странице поддержки проекта PyScripter, показано на рис. В.13.

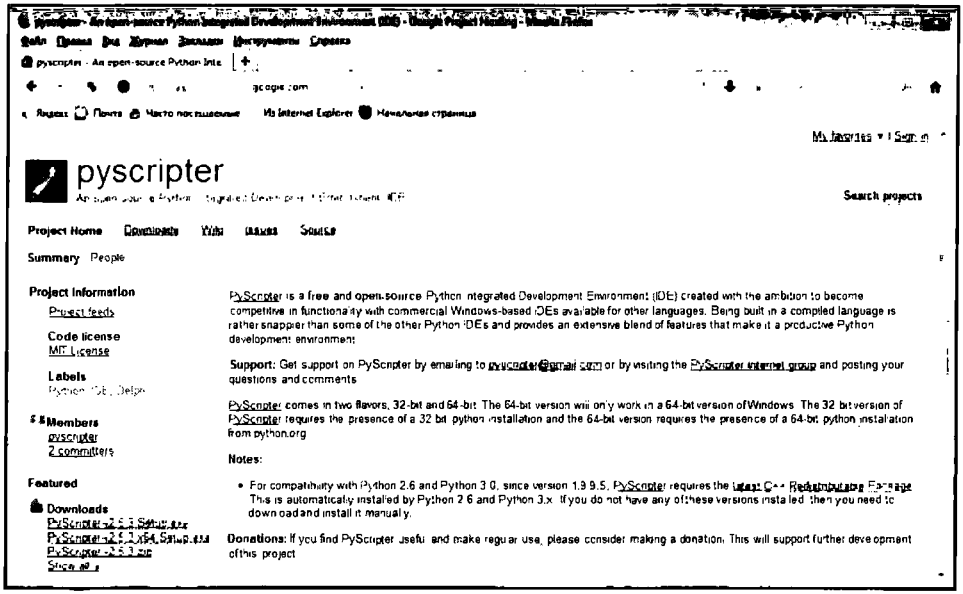


Рис. В.13. Страница поддержки проекта PyScripter

Окно среды разработки с программным кодом показано на рис. В.14.

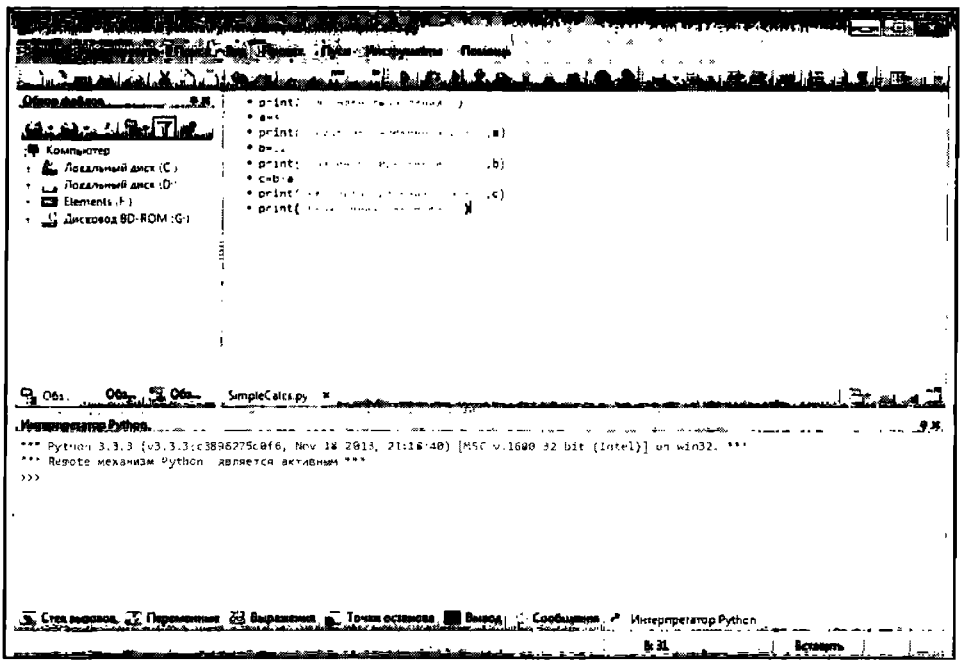


Рис. В.14. Окно среды PyScripter с программным кодом

На заметку

Чтобы создать новый файл с программой, выбираем команду **Новый** в меню **Файл**, а чтобы открыть уже существующий файл выбираем команду **Открыть** из того же меню.

Для запуска программы на выполнение выбираем в меню **Пуск** команду **Пуск** (рис. В.15) или щелкаем кнопку с зеленой стрелкой на панели инструментов.

На рис. В.16 показан результат выполнения программы.

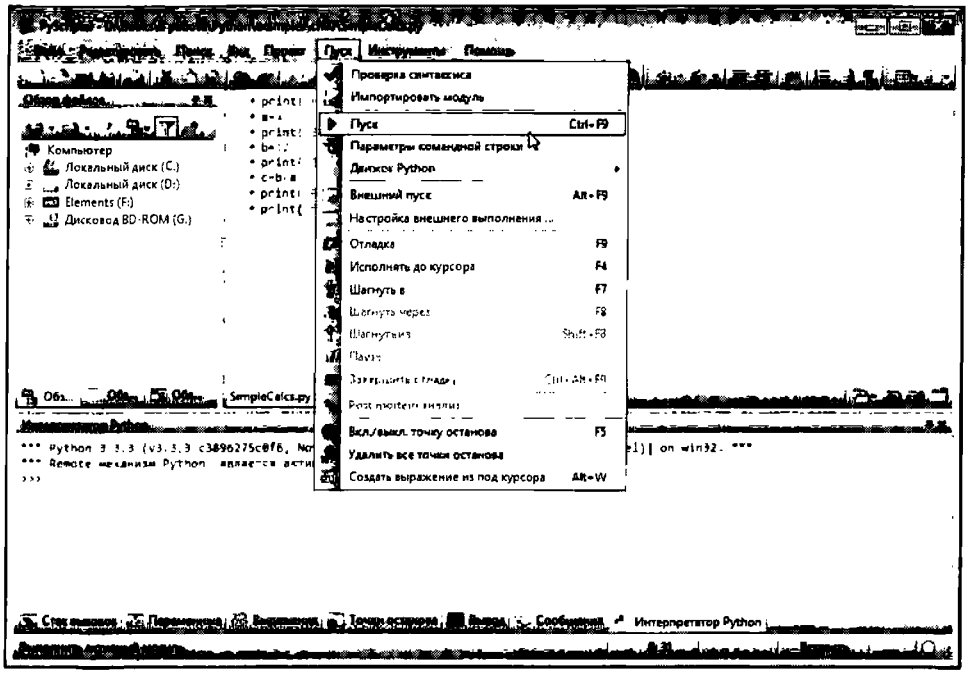


Рис. В.15. Запуск программы на выполнение в среде PyScripter

Результат отображается во внутреннем окне **Интерпретатор Python**, и это достаточно удобно.

На заметку

При желании во внутреннем окне **Интерпретатор Python** можно выполнять отдельные команды: инструкции для выполнения вводятся после символа >>>.

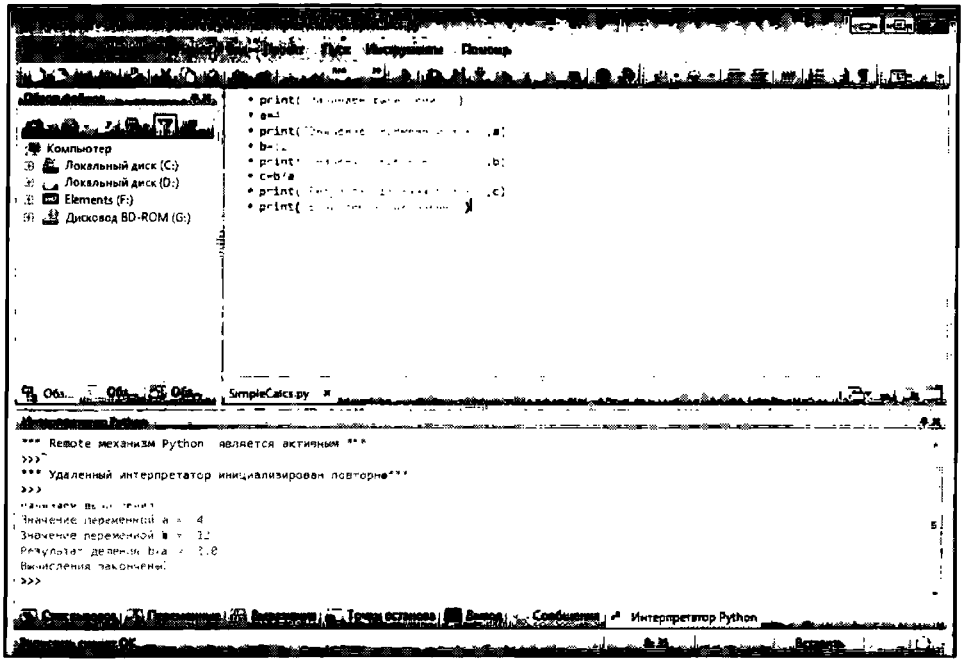


Рис. В.16. Результат выполнения программы в среде PyScripter

Поскольку в ниши планы входит широкое использование среды разработки PyScripter для работы с программными кодами, далее мы более детально обсудим некоторые особенности данного приложения. Также отметим, что если читатель в силу каких-то объективных или субъективных причин предпочтет другую среду разработки (в том числе и одну из перечисленных выше) - нет никаких проблем. Правда, на страницах книги отсутствует возможность дать описание всех (или даже некоторых) наиболее популярных сред разработки для Python: книга, все-таки, посвящена языку программирования, а не программному обеспечению. Да и большинство предлагаемых утилит для работы с программными кодами Python обычно просты в использовании, универсальны в плане методов и приемов работы с ними, а также интуитивно понятны. Хочется верить, что читатель в случае надобности сам сможет справиться с освоением необходимого программного обеспечения.

Работа со средой PyScripter

Будь проклят тот день, когда я сел за баранку этого пылесоса!
из к/ф "Кавказская пленница"

Сразу оговоримся, что полностью описывать приложение PyScripter мы не будем: во-первых, возможности такой нет, а, во-вторых, необходимости, если честно, тоже. Поэтому мы остановимся лишь на тех настройках и режимах, которые критичны и будут (или могут быть) полезны читателю в процессе работы над материалом книги (имеются в виду в первую очередь, конечно, программные коды, которые рассматриваются в книге).

Прежде всего, стоит обратить внимание, что интерфейс среды разработки PyScripter поддерживает различные языки, в том числе и русский. На рис. В.17 показано окно приложения PyScripter, в котором используется русскоязычный интерфейс.

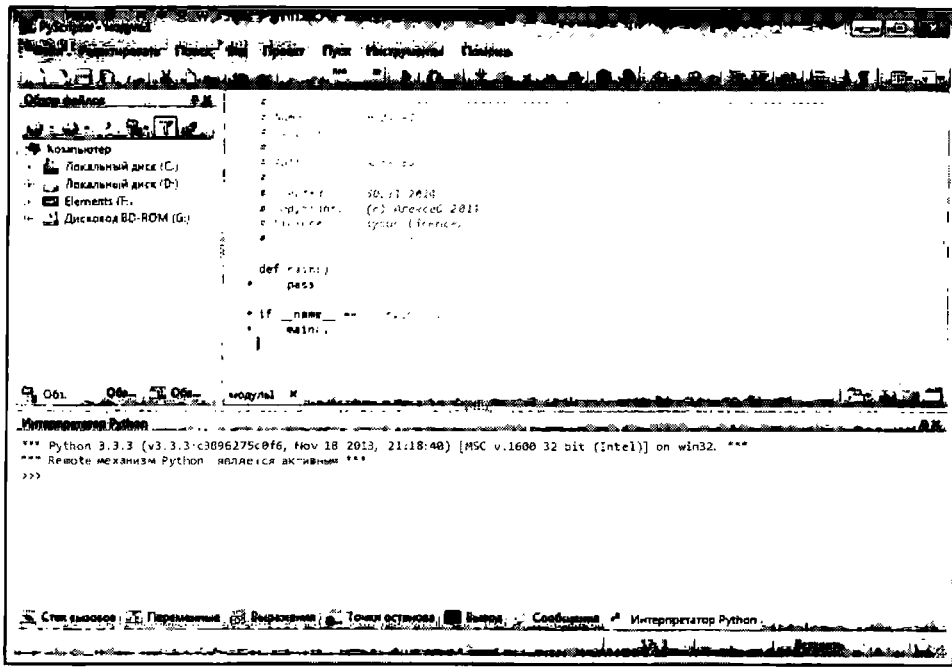


Рис. В.17. Окно приложения PyScripter с русскоязычным интерфейсом и шаблонным кодом в окне редактора кодов

На заметку

По умолчанию при запуске приложения PyScripter во внутреннем окне редактора кодов для нового, автоматически созданного (но еще не сохраненного) файла предлагается шаблонный код, как это можно видеть на рис. В.17. Этот шаблонный код можно удалить и ввести собственный. Также пользователь может изменить настройки приложения, в том числе и содержимое шаблонного кода.

Если мы хотим перейти к англоязычному интерфейсу, то в меню **Вид** следует выбрать подменю **Язык**, а в этом подменю - команду **Английский**, как показано на рис. В.18.

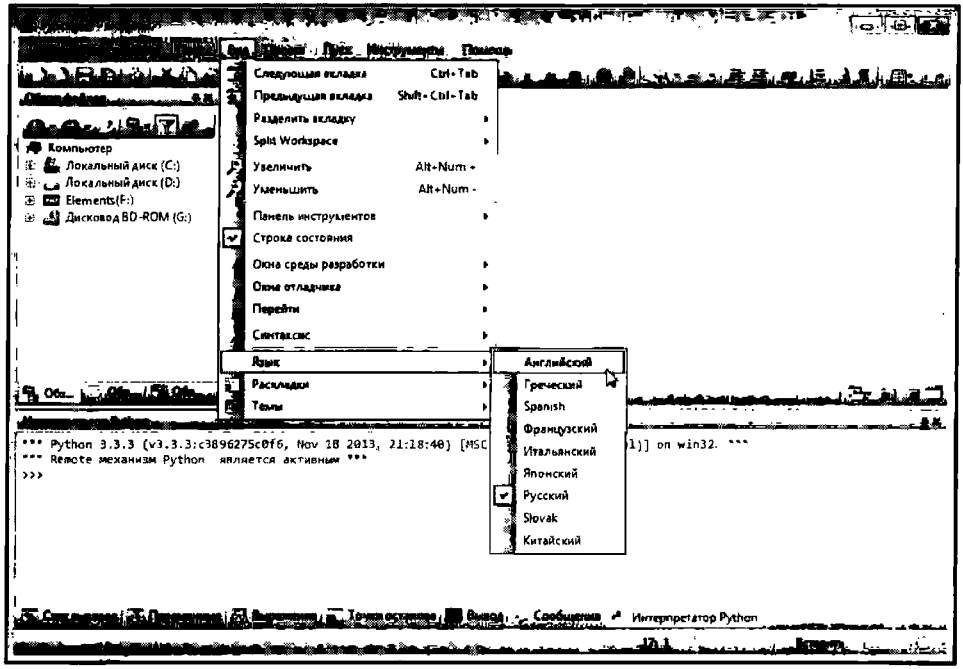


Рис. В.18. Переходим к англоязычному интерфейсу для приложения PyScripter

В результате интерфейс приложения PyScripter станет англоязычным. Чтобы вернуться обратно к русскоязычному интерфейсу, в меню **View** в подменю **Language** выбираем команду **Russian** (рис. В.19).

Хочется верить, что такие стандартные процедуры, как создание, сохранение, открытие и закрытие рабочего документа (файла с программой) у читателя проблем не вызовут. Вместе с тем обращаем внимание на достаточно полезную утилиту: внутреннее окно **Обзор файлов**, непосредственно в котором можно в системе каталогов выбрать тот документ, с которым собираемся работать (рис. В.20).

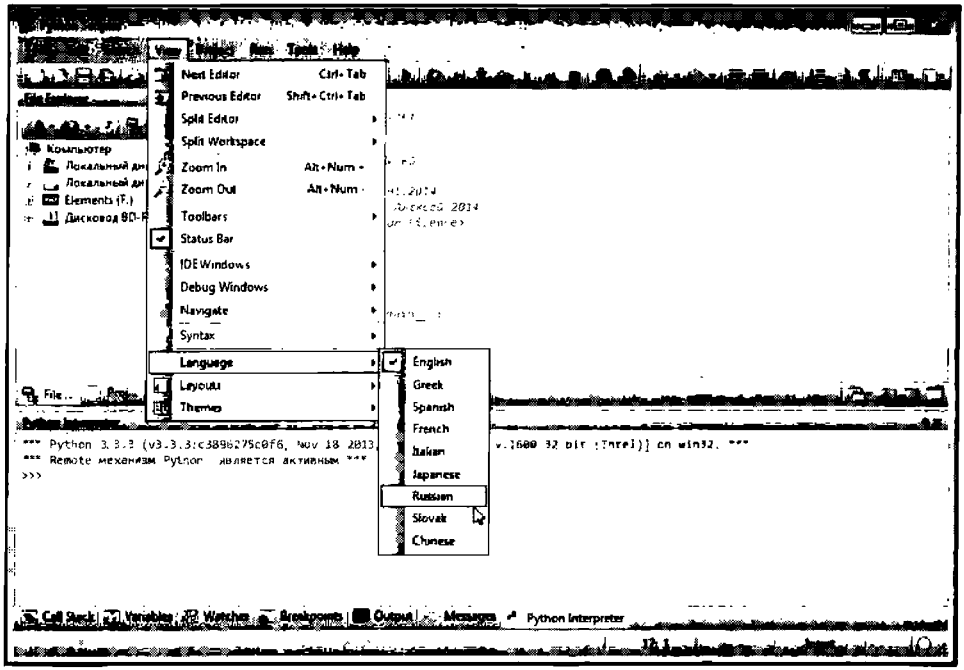


Рис. В.19. Окно приложения PyScripter с англоязычным интерфейсом

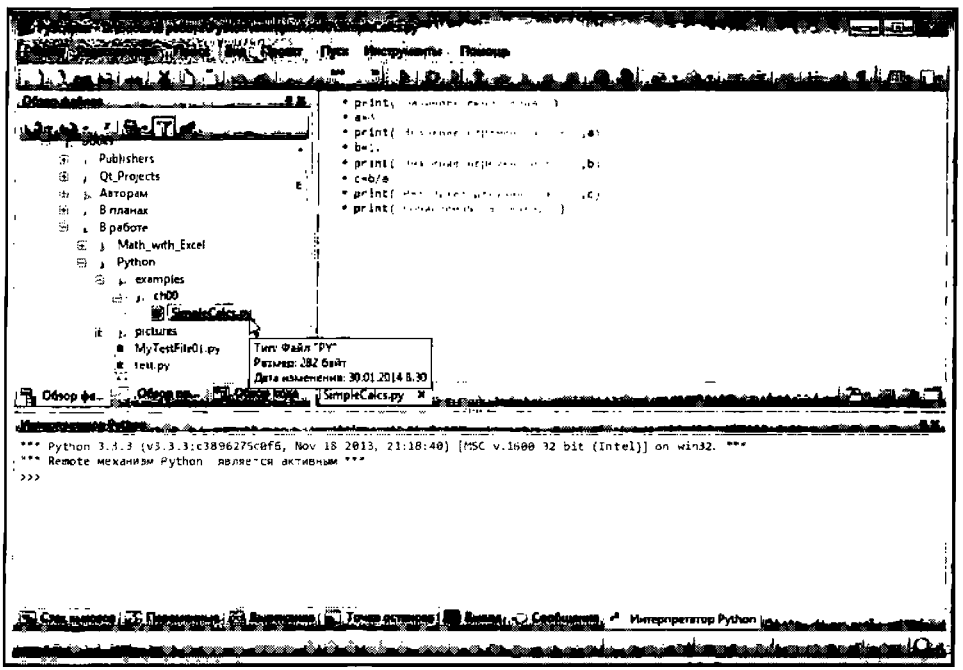


Рис. В.20. Выбрать нужный файл с программным кодом можно непосредственно в окне приложения PyScripter

Среда разработки PyScripter достаточно гибкая в плане настроек. Например, чтобы настроить параметры редактора кодов (такие, скажем, как шрифт или режим выделения синтаксических конструкций) в меню **Инструменты** в подменю **Параметры** выбираем команду **Свойства редактора**, как показано на рис. В.21.

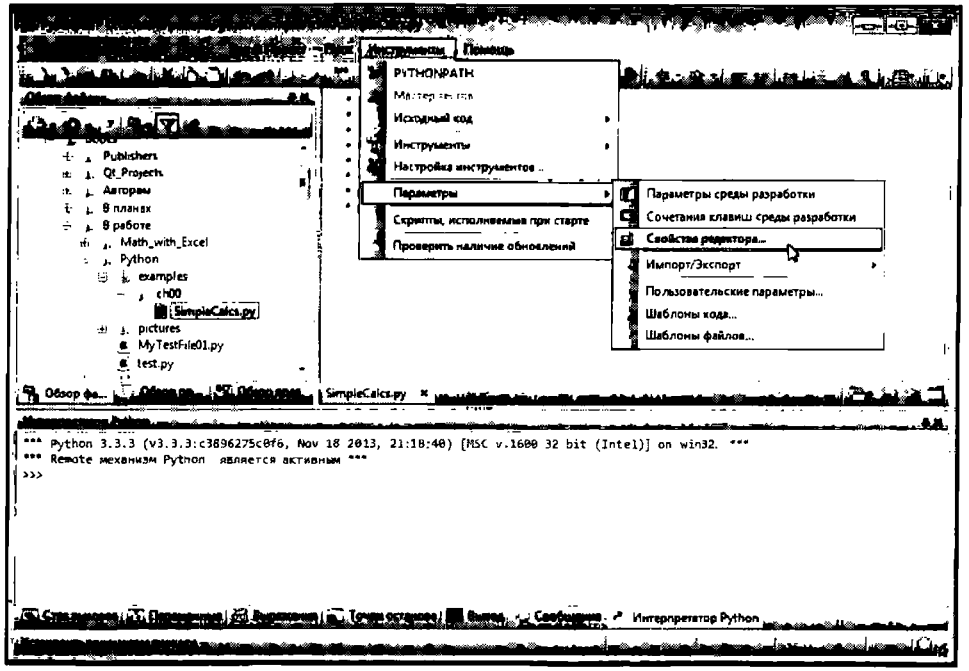


Рис. В.21. Переход в режим настройки параметров редактора программных кодоа

В результате откроется диалоговое окно с названием **Редактор свойств**, в котором, собственно, и выполняются настройки (рис. В.22).

Одно весьма важное замечание касается программных кодов, в которых используется кириллический текст. Такие файлы нужно сохранять в "правильной" кодировке - иначе при последующем открытии в том месте, где был кириллический текст, появятся "каракули". В частности, рекомендуется перед сохранением файла в меню **Редактировать** в подменю **Формат файла** установить кодировку **UTF-8** (рис. В.23).

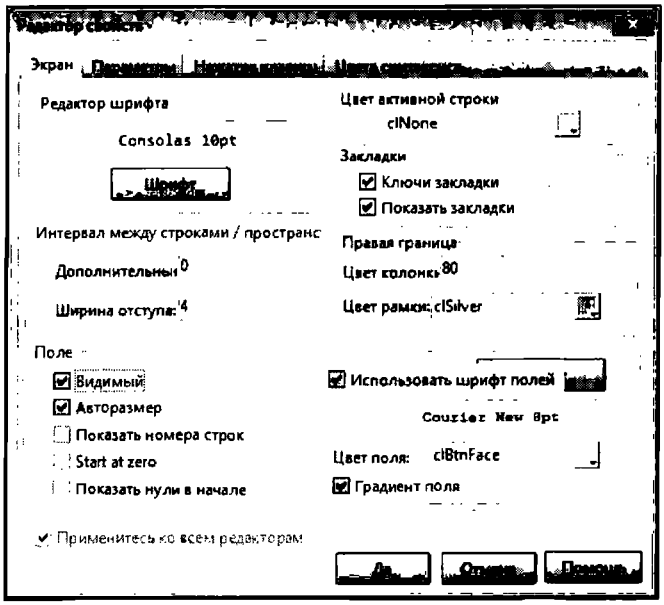


Рис. В.22. Окно настройки параметров редактора программных кодов

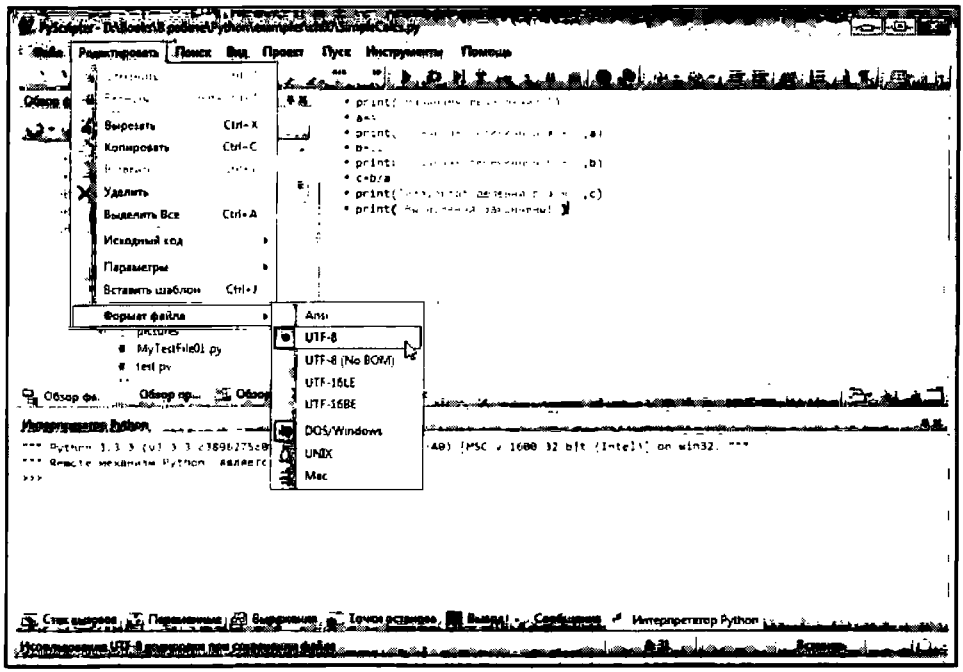


Рис. В.23. При работе с кириллическим текстом для корректной работы необходимо файлы сохранять в "правильной" кодировке

Также ранее мы отмечали, что используемые по умолчанию шаблоны программных кодов, предлагаемые пользователю в различных ситуациях (при создании нового документа) можно изменить (отредактировать). Полезными в этом случае будут команды подменю **Параметры** в меню **Инструменты**. На рис. В.24 показано диалоговое окно **Файлы шаблонов**, с помощью которого редактируются шаблоны.

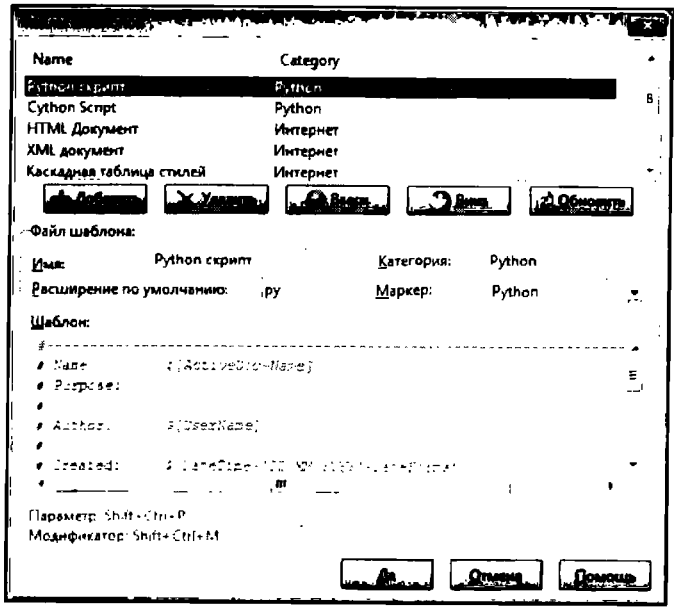


Рис. В.24. Окно настройки шаблонов

Характеризуя ситуацию в целом, следует сказать, что приложение PyScripter настраивается довольно просто, так что на практике проблем с настройкой обычно не возникает даже у неподготовленных пользователей. В случае крайней необходимости, разумеется, всегда можно обратиться к справочной системе среды разработки PyScripter.

На заметку

В книге мы будем рассматривать исходные программные коды на языке Python и результат выполнения этих кодов - в текстовом формате. Поэтому работу со средой разработки отдельно обсуждать не будем. Фактически, задача набора и выполнения программного кода полностью ложится на плечи читателя. Благо, задание это не очень сложное.

Если все же возникнут непредвиденные проблемы с программным обеспечением, напомним, что на худой конец программный код можно набрать

в обычном текстовом редакторе и сохранить в соответствующий файл с расширением `py`. Затем этот файл выполняется с помощью программы-интерпретатора. То есть программу-интерпретатор все же придется установить.

Благодарности

- А что передать мой король?
- Передай твой король мой пламенный привет!
из к/ф "Иван Васильевич меняет профессию"

Автор выражает искреннюю признательность сотрудникам издательства *"Наука и Техника"*, благодаря слаженной и профессиональной работе которых книга стала лучше и интересней. Особая благодарность - *Марку Финкову*, при поддержке которого уже реализованы интересные проекты и, хочется верить, много новых еще будет реализовано.

Примеры и задачи из книг по программированию обычно проходят апробацию на терпеливой, но требовательной аудитории - студентах и слушателях курсов. Им - отдельное спасибо. Ведь непосредственное общение с теми, кто учится программированию, дает бесценный опыт для преподавателя, и это, естественным образом, находит свое отражение в книгах.

Огромнейшей благодарности заслуживают читатели, которые присылали и присылают письма со своими критическими замечаниями и предложениями. Благодаря их неравнодушному отношению воплощены в жизнь многие замечательные идеи. От уважаемого читателя всецело зависит, будет ли продолжен этот процесс.

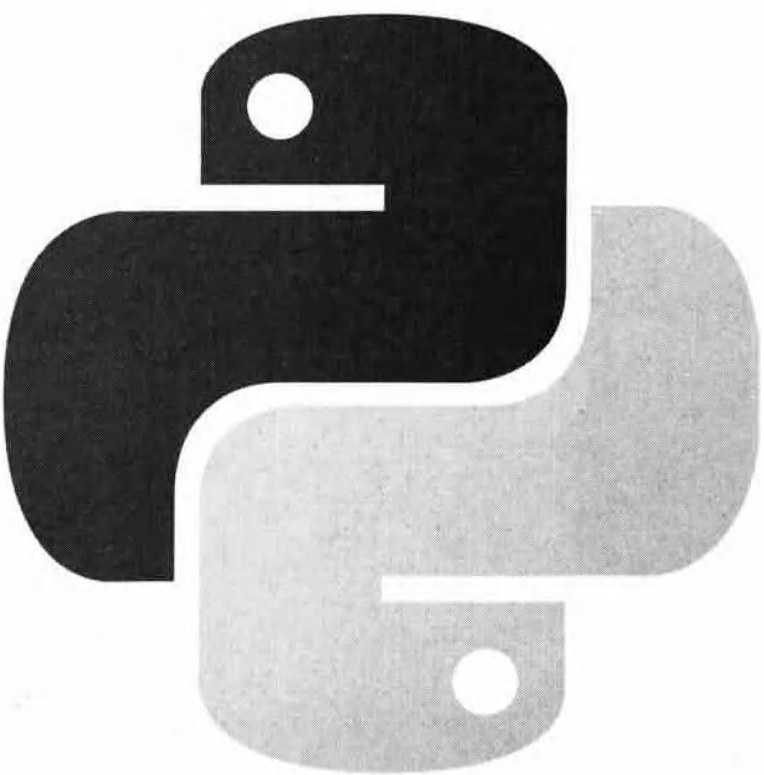
Обратная связь

Я могу пронизать пространство и уйти в прошлое.
из к/ф "Иван Васильевич меняет профессию"

Свои пожелания, замечания и предложения читатели могут направлять по электронной почте *alex@vasilev.kiev.ua* или *vasilev@univ.kiev.ua*. Книги в первую очередь пишутся для читателей. Поэтому крайне важно знать, что в

книге удалось, а что - нет. Конструктивная критика и обратная связь с читателями - очень действенные инструменты, которые позволяют двигаться в правильном направлении.

Некоторая полезная информация по этой книге и другим книгам автора представлена на сайте *www.vasilev.kiev.ua*. Если же нужной читателям информации там нет, но технически ее размещение возможно, имеет смысл написать письмо с предложением добавить на страницу недостающие сведения - адреса электронной почты приведены выше.



Глава 1

Первые программы на языке Python

- Паки, паки... иже херувимы! Ваше сиятельство, смилуйтесь. Между прочим, Вы меня не так поняли.
- Да как же тебя понять, коль ты ничего не говоришь?

- Языками не владею, Ваше благородие.
из к/ф "Иван Васильевич меняет профессию"

Мы приступаем к изучению языка Python. В этой главе познакомимся с базовыми конструкциями, которые позволят нам составлять несложные программные коды. Некоторые моменты сначала будут объясняться на немного поверхностном уровне, чтобы облегчить для новичков процесс перехода от теоретизирования к практическому использованию Python для написания программ.

На заметку

Нередко программы, написанные на языке Python, называют *сценариями*. Мы несколько отойдем от традиции и к термину *сценарий* прибегать не будем.

Начнем мы с того, что обсудим общие принципы организации программного кода, написанного на Python.

Размышляя о программе

Это же вам не лезгинка, а твист!
из к/ф "Кавказская пленница"

Программа в Python - это последовательность команд. Никаких специальных инструкций для формального обозначения начала или окончания кода программы использовать не нужно. Таким образом, при написании программы мы размещаем команды одна за другой - в том порядке, как они должны выполняться. Команда помещается в начале строки. Отступов делать не нужно. В одной строке обычно по одной команде - то есть каждая команда в новой строке. "

На заметку

В данном случае речь идет о "простых" командах, наподобие команды вывода в консольное окно текстового сообщения. Как отмечалось выше, эти команды размещаются в начале строки, пробел перед ними не ставится, и в конце команды тоже ничего ставить не нужно. Немного позже мы познакомимся с управляющими инструкциями - такими, как оператор цикла и условный оператор. В этих синтаксических конструкциях с помощью отступов (пробелов) выделяются составные части выражения для оператора (блок команд тела оператора). Но об этом мы поговорим позже. На данный момент важно запомнить, что пробел в языке Python -важный элемент с точки зрения синтаксиса и использовать его следует с крайней осторожностью. Лишний пробел может вызвать ошибку.

В принципе, это все, что нам пока необходимо знать о структуре программы для того, чтобы приступить к написанию программного кода. Со временем мы познакомимся с различными синтаксическими конструкциями, которые вносят в программный код большую интригу и делают процесс программирования интересным и в некоторых случаях непредсказуемым. Но это будет позже.

Что важно сейчас? Сейчас важно понимать, что команды в программе должны быть корректными как минимум с точки зрения синтаксиса языка Python. Но даже формально правильные команды не гарантируют правильности выполнения программы. Обычно программы пишутся для решения той или иной задачи. Задача решается в соответствии с определенным алгоритмом, который в подавляющем большинстве случаев разрабатывает и реализует программист. То есть в программе реализуется некоторый алгоритм. Если этот алгоритм неправильный, то и программа выдаст не тот результат, которого от нее ожидают. Поэтому по-хорошему написание программы начинается с выработки алгоритма, а уже затем этот алгоритм реализуется в виде команд программы. Мы для простоты исходим из предположения, что алгоритм имеется, и его лишь необходимо "перевести" на язык инструкций Python.

Практически любая программа оперирует с некоторыми данными. Это может быть как реально большая база данных, так и одно-единственное значение - принципиальной разницы здесь нет. Важно то, что данные в программе должны как-то сохраняться. Мы пока что будем "сохранять" данные с помощью *переменных*. В отношении переменных важны два момента:

- у переменной есть имя (или название);
- переменная *ссылается* на некоторое значение.

В принципе, значение, на которое ссылается переменная, мы можем:

- прочитать;

- изменить.

И в том, и в другом случае мы используем, в контексте команды, имя переменной. Поскольку в Python тип переменной явно не указывается (он определяется по значению, на которое ссылается переменная), то предварительно объявлять переменные не нужно. Просто при первом использовании переменной ей сразу присваивается значение.

На заметку

Обычно, когда объясняют назначение переменных, сравнивают их, например, с корзиной или банковской ячейкой. Значение переменной при такой аналогии - это то, что находится в корзине/ячейке. Пока что мы можем думать о переменной именно так. Тем не менее, в Python дела с переменными обстоят несколько иначе. Технически переменная содержит адрес в области памяти, где хранится значение, отождествляемое со значением переменной. Тем не менее очень часто (в простых случаях) внешний эффект такой, как если бы переменная реально содержала свое значение - как корзина или банковская ячейка. Пока что механизм хранения значений переменных не важен. Немного позже, когда данный вопрос станет актуальным, мы расставим все точки над "i".

Для начала мы рассмотрим небольшую программу. В ней, кроме нескольких простых команд, будут использованы *комментарии*. Комментарий - это текст, предназначенный для программиста. Интерпретатором комментарий игнорируется. Для создания комментария используют символ #. Все, что находится справа от символа #, является комментарием.

Пример простой программы

Баранов - в стойло, холодильник - в дом.
из к/ф "Кавказская пленница"

В программе, которую мы рассмотрим далее, реализуется следующий алгоритм:

- выводится текстовое приветствие с просьбой к пользователю указать имя;
- после того, как пользователь вводит имя, оно считывается и записывается в переменную;
- программой выводится еще одно сообщение, а текст сообщения содержит имя, которое на предыдущем шаге ввел пользователь.

Для вывода сообщений в консольное окно используется функция `print()`, а для считывания введенного пользователем текста (имя пользователя) используем функцию `input()`. Также в программе есть комментарии. Полный код программы приведен в листинге 1.1.

Листинг 1.1. Программа с вводом и выводом данных

```
# Выводится сообщение
print("Давайте познакомимся!")
# Считываем введенное пользователем значение.
# Результат записывается в переменную name
name=input("Как Вас зовут? ")
# Выводится новое сообщение
print("Добрый день,",name+"!")
```

Поскольку это первая наша "официальная" программа, исследуем вопрос о том, как она будет выполняться в среде PyScripter и IDLE (на практике между процессом выполнения программы в этих средах имеются некоторые отличия). На рис. 1.1 показано окно среды PyScripter с программным кодом (перед запуском программы на выполнение).

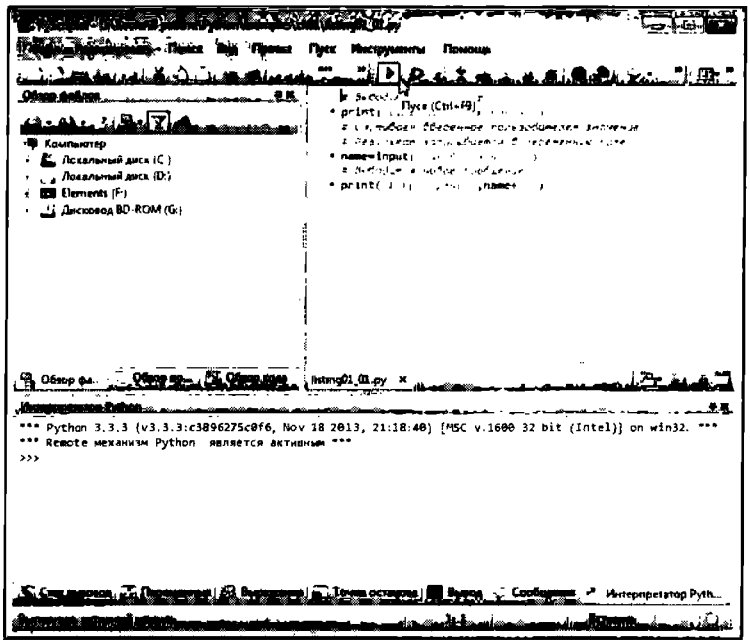


Рис. 1.1. Окно среды PyScripter перед началом выполнения программы

В процессе выполнения программного кода сначала в окне интерпретатора появляется первое сообщение, а затем открывается окно с полем ввода. Перед полем ввода отображается текст `Как Вас зовут?`, а в поле ввода мы указываем имя пользователя (рис. 1.2).

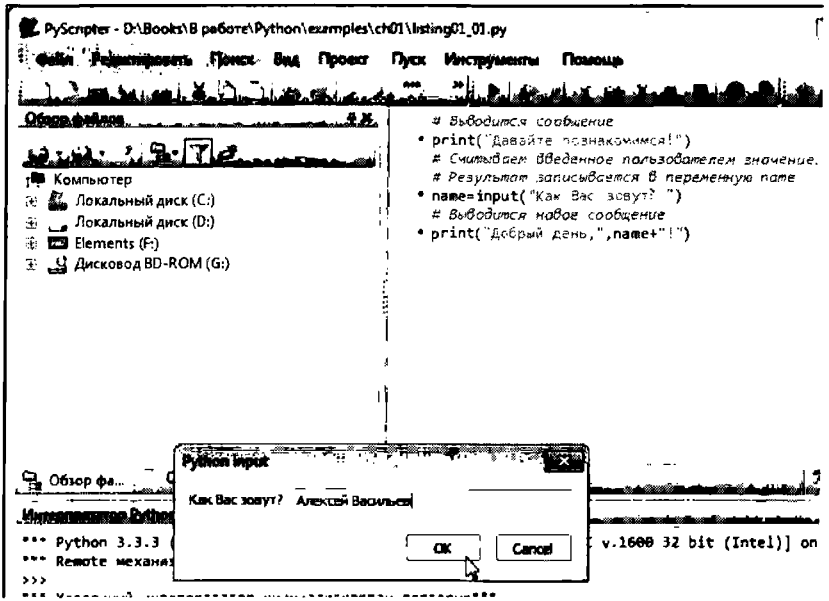


Рис. 1.2. В процессе выполнения программы в среде PyScripter появляется окно с полем ввода

После щелчка на кнопке OK в окне ввода, получаем результат, как на рис. 1.3.

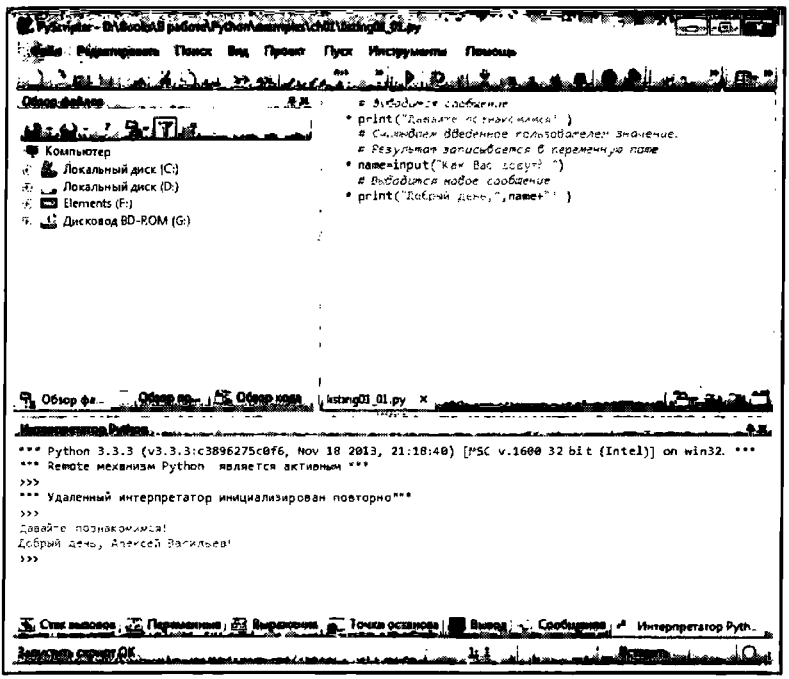


Рис. 1.3. Результат выполнения программы в среде PyScripter

Таким образом, при работе со средой PyScripter ввод выполняется в отдельное окно, которое отображается автоматически. Несколько иначе обстоят дела при использовании среды IDLE. На рис. 1.4 показано окно среды с программным кодом перед началом выполнения программы.

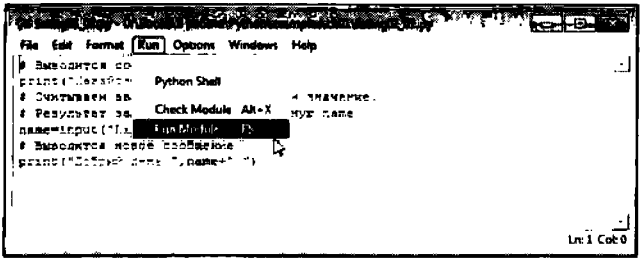


Рис. 1.4. Окно среды IDLE перед началом выполнения программы

После запуска программы на выполнение в окне интерпретатора появляется первое сообщение и, в новой строке, фраза `Как Вас зовут?`, после которой пользователем выполняется ввод текста, как это показано на рис. 1.5.

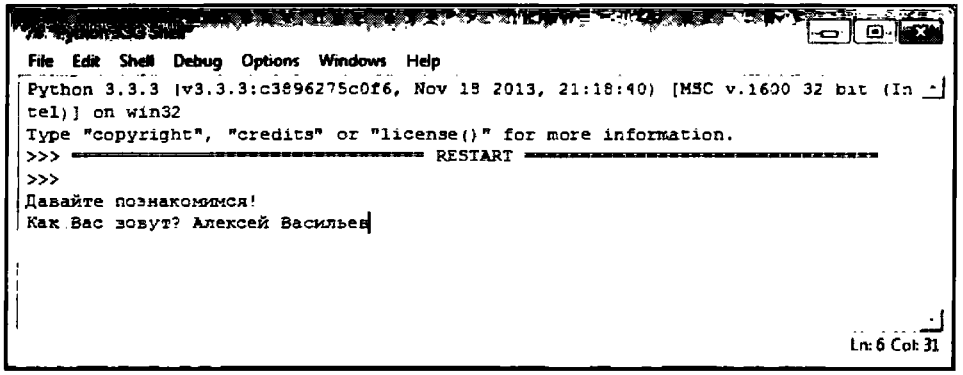


Рис. 1.5. В процессе выполнения программы в среде IDLE ввод текста выполняется в консольном окне

Ввод пользователя подтверждается нажатием клавиши <Enter>. Результат выполнения программы представлен на рис. 1.6.

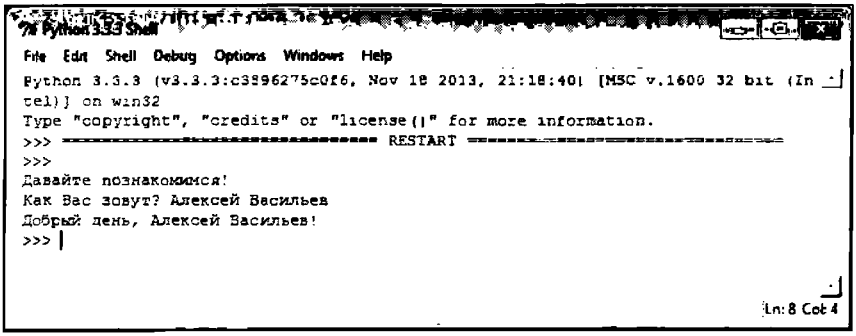


Рис. 1.6. Результат выполнения программы в среде IDLE

Таким образом, результат выполнения программы будет следующим (жирным шрифтом выделен ввод пользователя):

Результат выполнения программы (из листинга 1.1)

```
Давайте познакомимся!
Как Вас зовут? Алексей Васильев
Добрый день, Алексей Васильев!
```

На заметку

Если программа выполняется в среде IDLE, результат будет точно таким, как показано выше (разумеется, с учетом того, какое значение ввел пользователь). В среде PyScripter вторая строка отсутствует. Хотя мы предполагаем использовать среду PyScripter, результат для тех программ, в которых пользователем вводятся данные, будем приводить в виде, как для среды IDLE.

Теперь рассмотрим более детально программный код, выполнение которого приводит к такому результату (см. листинг 1.1):

- Все, что начинается с символа #, является комментарием и на результат выполнения программы не влияет.
- Командой `print("Давайте познакомимся!")` в окне интерпретатора выводится сообщение, определяемое аргументом функции `print()`.
- В команде `name=input("Как Вас зовут? ")` вызывается функция `input()` с текстовым аргументом, а результат вызова функции записывается в переменную `name`. Как следствие в окне интерпретатора (окне с полем ввода) появляется текст, переданный аргументом функции `input()`. То значение, которое пользователь ввел в

окне интерпретатора (поле ввода диалогового окна) возвращается как значение (результат) функции `input()`. Это и есть значение переменной `name`.

- Командой `print("Добрый день,",name+"!")` в окне интерпретатора выводится еще одно сообщение, которое получается объединением (конкатенацией) текста `Добрый день,`, значения переменной `name` и восклицательного знака.

На заметку

В последней команде у функции `print()` два аргумента: `"Добрый день,"` и `name+"!"`. И первый, и второй аргументы - текстовые. Они выводятся в одной строке друг за другом, причем между ними автоматически по умолчанию добавляется пробел. Что касается второго аргумента, то формально он представлен как "сумма" текстовой переменной `name` (точнее, переменной с текстовым значением) и текста `"!"`. Если операция сложения применяется к текстовым значениям, результатом является текст, получающийся объединением соответствующих текстовых значений. Обратите внимание, что пробел в этом случае не добавляется. Чтобы не запутаться с добавлением или не добавлением пробела можно пользоваться таким правилом: функция `print()` печатает свои аргументы через пробел, а при объединении строк пробел не добавляется. Ну и, разумеется, читатель уже заметил, что текстовые значения указываются в двойных кавычках. Это не единственный способ выделения текста в Python (можно, например, использовать одинарные кавычки). Мы в основном будем заключать текстовые значения (литералы) в двойные кавычки.

Далее мы более подробно обсудим некоторые особенности работы с переменными в Python.

Обсуждаем переменные

Бывают такие случаи, когда неплохо и соврать.
из к/ф "Ирония судьбы или с легким паром"

Мы уже имели дело с переменными. Но это было скорее "шапочное" знакомство. Здесь мы уделим переменным немного больше внимания. Надо сказать, они того заслуживают. Тем более что в Python переменные достаточно специфичны.

В первую очередь, что касается названия переменных: в принципе, это может быть практически любое имя (комбинация букв, цифр и символов подчеркивания), которое не совпадает ни с одним из ключевых слов Python. Список ключевых слов Python представлен в таблице 1.1.

Таблица 1.1. Ключевые слова Python

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Ключевые слова не могут быть модифицированы, и попытка использовать переменную с соответствующим именем приведет к ошибке.

На заметку

Заметьте, что в данном случае мы не обсуждаем назначение ключевых слов. Список ключевых слов приведен исключительно для того, чтобы читатель знал, как не следует называть переменные.

Помимо ключевых слов, в Python существует множество встроенных идентификаторов - таких, например, как названия встроенных функций (список встроенных идентификаторов приведен в таблице 1.2).

Таблица 1.2. Встроенные идентификаторы Python

ArithmeticError	SyntaxError	float
AssertionError	SyntaxWarning	format
AttributeError	SystemError	frozenset
BaseException	SystemExit	getattr
BlockingIOError	TabError	globals
BrokenPipeError	TimeoutError	hasattr
BufferError	True	hash
BytesWarning	TypeError	help
ChildProcessError	UnboundLocalError	hex
ConnectionAbortedError	UnicodeDecodeError	id
ConnectionError	UnicodeEncodeError	input
ConnectionRefusedError	UnicodeError	int
ConnectionResetError	UnicodeTranslateError	isinstance
DeprecationWarning	UnicodeWarning	issubclass
EOFError	UserWarning	iter
Ellipsis	ValueError	len

```
EnvironmentError                Warning                 license
Exception                       WindowsError            list
False                           ZeroDivisionError       locals
FileExistsError                 __build_class__         map
FileNotFoundError               __debug__               max
FloatingPointError              __doc__                 memoryview
FutureWarning                   __import__              min
GeneratorExit                   __loader__              next
IOError                         __name__                object
ImportError                     __package__             oct
ImportWarning                   abs                     open
IndentationError                all                     ord
IndexError                      any                     pow
InterruptedError                ascii                   print
IsADirectoryError               bin                     property
KeyError                        bool                    quit
KeyboardInterrupt               bytearray               range
LookupError                     bytes                   repr
MemoryError                     callable                reversed
NameError                       chr                     round
None                            classmethod             set
NotADirectoryError              compile                 setattr
NotImplemented                  complex                 slice
NotImplementedError             copyright               sorted
OSError                         credits                 staticmethod
OverflowError                   delattr                 str
PendingDeprecationWarning       dict                    sum
PermissionError                 dir                     super
ProcessLookupError              divmod                  tuple
ReferenceError                  enumerate               type
ResourceWarning                 eval                    vars
RuntimeError                    exec                    zip
RuntimeWarning                  exit

StopIteration                   filter
```

На заметку

В таблице 1.2 приведены в основном названия встроенных классов исключений и названия встроенных функций. Вообще, узнать, какие идентификаторы на данный момент задействованы в работе интерпретатора (определены в программе) можно с помощью функции `dir()`. Если вызвать эту функцию без аргументов (скажем, воспользоваться командой `print(dir())`), получим список (набор) идентификаторов (имен), которые уже "заняты". Но в этот список не будут входить названия встроенных функций и других встроенных идентификаторов. Получить доступ к

списку встроенных идентификаторов можно через *модуль* `builtins`. Для этого необходимо командой `import builtins` подключить модуль, а затем вызвать функцию `dir()` с аргументом `builtins` (для отображения списка встроенных идентификаторов в области вывода используем команду `print(dir(builtins))`). Подробнее о *подключении модулей* рассказывается в конце этой главы.

Формально мы можем задействовать переменную с именем, совпадающим с тем или иным идентификатором. Однако это может неожиданным образом повлиять на выполнении программы. Подобные ситуации считаются дурным тоном в программировании, и их следует избегать.

Имя переменной не может начинаться с цифры. Также крайне осторожно следует использовать подчеркивание в начале и конце имени переменной (переменные с начальным подчеркиванием и с двойным подчеркиванием в начале и конце имени обрабатываются по специальным правилам).

О том, что для переменных не нужно явно объявлять тип, мы уже знаем. Однако это вовсе не означает, что типов данных в Python нет. Другое дело, что такое понятие как тип отождествляется именно с данными, а не с переменной.

🕮 На заметку

Вообще тип данных важен, по меньшей мере, с двух позиций: тип данных определяет объем памяти, выделяемый для хранения этих данных, а также находится в тесной связи с теми операциями, которые допустимы с данными. Во многих языках программирования переменные объявляется с указанием типа. Вот в этом случае переменную удобно представлять как корзину, на которой написано название переменной, а значение переменной - то, что внутри корзины. Таких корзин в программе может быть много. Каждая корзина отождествляется с какой-то конкретной переменной, в каждой корзине что-то "лежит" (значение соответствующей переменной). Размер корзины определяется типом переменной, с которой мы эту корзину отождествляем. Когда мы считываем значение переменной, то просто "смотрим", что лежит в корзине. Когда меняем значение переменной, то вынимаем из корзины то, что в ней было, и помещаем туда новое содержимое.

В Python все несколько иначе. Переменные не имеют типа. Но тип есть у данных, которые "запоминаются" с помощью переменных. Поэтому концепция обычных корзин теряет свою актуальность. Актуальной становится концепция корзин, к которым привязаны веревочки. Переменная - это веревочка с биркой (название переменной), а корзина -это данные, на которые ссылается переменная. То есть теперь название переменной - это не бирка на корзине, а бирка на веревке. На корзинах бирок нет. Как и ранее, размер корзины определяется характером ее содержимого (тип данных). Но доступ к корзине у нас есть только через веревочку. Веревочки все одинаковые и отличаются лишь названием на бирке. Что мы можем сделать в такой ситуации: мы можем потянуть за веревочку и посмотреть, что находится в корзине, к которой привязана веревочка. Это аналог считывания значения переменной. Также мы можем поменять содержимое корзины или вообще отвязать

от веревочки корзину и привязать веревочку к другой корзине. Это аналог изменения значения переменной. Более того, мы можем привязать несколько веревочек к одной и той же корзине. На языке программирования это означает, что несколько переменных ссылаются на одно и то же значение. И такое в Python тоже возможно.

В Python переменные *ссылаются* на данные, а не содержат их, как в некоторых других языках программирования. Каждая переменная "помнит", в каком месте в памяти находится некоторое значение. Это значение мы отождествляем со значением переменной (хотя на самом деле значением переменной является адрес ячейки памяти с соответствующими данными). Тем не менее, когда мы обращаемся к переменной, то выполняется автоматический переход по ссылке на данные, так что внешне иллюзия такая, как если бы переменная реально содержала определенное значение. Какие следствия из всего сказанного?

Во-первых, описанный выше механизм нередко является краеугольным камнем в понимании того, что происходит при выполнении программного кода и, в частности, при присваивании значений переменным.

Во-вторых, совершенно очевидно, что одна и та же переменная на разных этапах выполнения программы может ссылаться не просто на разные значения, но на значения разных типов (например, сначала ссылаться на текст, а затем на число).

В-третьих, хотя непосредственно у переменных типа нет, мы можем получить доступ через переменную к значению, на которое она ссылается, и узнать его тип. Осталось лишь выяснить, какие типы данных вообще возможны в Python.

🕮 На заметку

Для определения типа значения, на которое ссылается переменная, можно использовать функцию `type()`. Переменная указывается аргументом функции.

Наш опыт работы с переменными пока что ограничивается текстовыми и числовыми значениями (пример во *Вступлении*). Так вот, текстовые значения относятся к типу, который называется `str`. Числовые значения относятся к типу `int` (если это целые числа) или к типу `float` (если это действительные числа в формате значения с плавающей точкой). Приятной неожиданностью для любителей математических расчетов будет то, что в Python есть встроенная поддержка для комплексных чисел. Для этих целей используются значения типа `complex`. Некоторые примеры несложных числовых расчетов приведены далее в этой главе.

Значения логического типа могут принимать два значения (*истина* или *ложь*). В Python логический тип обозначается как `bool`. С логическими выражениями мы познакомимся, когда приступим к изучению условных инструкций и инструкций цикла.

Несколько позже мы познакомимся со *списками*, которые играют роль массивов в Python. Списки относятся к типу `list`. Еще в Python есть *множества* (о них мы тоже поговорим, но не в этой главе). Множества относятся к типу `set`. Существуют и другие типы данных, которые мы будем изучать постепенно, по мере нашего изучения языка Python.

На заметку

В первую очередь имеются в виду такие "разновидности" данных, как *кортежи* (тип `tuple`) и *словари* (тип `dict`). А еще в Python есть следующие типы: `bytes` (неизменяемая последовательность байтов), `bytearray` (изменяемая последовательность байтов), `frozenset` (неизменяемое множество), `function` (функция), `module` (модуль), `type` (класс). Есть также типы для специальных значений `ellipsis` (для элемента `Ellipsis`) и `NoneType` (для значения `None`).

Как отмечалось выше, от типа данных зависит то, какие операции могут с этими данными выполняться. Манипулировать с данными можем или с помощью функций, или операторов. Какие операторы, когда и как используются в Python, мы обсудим в следующем разделе.

На заметку

В завершение этого раздела отметим, что в Python переменные можно не только создавать, но и удалять. Для удаления переменной используется инструкция `del`, после которой, через пробел, указывается имя удаляемой переменной. Если удаляемых переменных несколько, они разделяются запятой.

Основные операторы

Вот что крест животворящий делает!
из к/ф "Иван Васильевич меняет профессию"

Обычно выделяют четыре группы операторов:

- арифметические;
- побитовые;

- операторы сравнения;
- логические операторы.

Арифметические операторы предназначены в первую очередь для выполнения арифметических вычислений. В таблице 1.3 перечислены и кратко описаны основные арифметические операторы языка Python.

Сразу следует отметить, что действие арифметических операторов достаточно точно соответствует "математической природе" этих операторов. При этом мы предполагаем, что речь идет о числовых расчетах.

Таблица 1.3. Арифметические операторы

Оператор	Описание
+	Оператор сложения. Вычисляется сумма двух чисел
-	Оператор вычитания. Вычисляется разность двух чисел
*	Оператор умножения. Вычисляется произведение двух чисел
/	Оператор деления. Вычисляется отношение двух чисел
//	Оператор целочисленного деления. Вычисляется целая часть от деления одного числа на другое
%	Оператор вычисления остатка от целочисленного деления. Вычисляется остаток от деления одного числа на другое
**	Оператор возведения в степень. Результатом является число, получающееся возведением первого операнда в степень, определяемую вторым операндом

🕮 На заметку

Некоторые из перечисленных выше операторов могут применяться не только к значениям числовых типов, но и, например, *тексту* или *спискам*. Эти темы мы пока не затрагиваем и обсудим их позже. Напомним только, что если к одной текстовой строке прибавить другую текстовую строку (с помощью оператора +), то в итоге произойдет конкатенация (объединение) строк: результатом будет текстовая строка, получающаяся объединением "суммируемых" строк. Например, в результате выполнения команды `txt="Язык "+"Python"` переменная `txt` будет ссылаться на текстовое значение `"Язык Python"`.

Если мы попытаемся умножить текстовое значение на целое число (или целое число на текст), то получим текстовую строку, получающуюся повторением (и конкате-

нацией) исходной строки (количество повторений определяется целочисленным операндом в инструкции умножения текста и числа). Например, в результате выполнения команды `txt="Python "*3` переменная `txt` будет ссылаться на текст `"Python Python Python "`, который получается троекратным повторением и конкатенацией исходного текста `"Python "`.

Пример программного кода для выполнения несложных арифметических вычислений приведен в листинге 1.2.

Листинг 1.2. Арифметические операторы

```
a=(5+2)**2-3*2 # Результат 43
b=6-5/2        # Результат 3.5
c=10//4+10%3   # Результат 3
# Результаты вычислений выводим на экран
print("Результаты вычислений:")
print(a,b,c)
```

Результат выполнения этого программного кода представлен ниже:

Результат выполнения программы (из листинга 1.2)

```
Результаты вычислений:
43 3.5 3
```

На заметку

Возможно, некоторых пояснений потребует процедура вычисления значения выражения `10//4+10%3`. Так, значением выражения `10//4` является целая часть от деления `10` на `4` - это число `2`. Значение выражения `10%3` - это число `1` (остаток от деления `10` на `3`). В результате получаем сумму `2` и `1` - то есть число `3`.

Язык Python позволяет создавать очень элегантные программные коды. Здесь мы воспользуемся возможностью, чтобы проиллюстрировать данное утверждение. Рассмотренный выше программный код мы перепишем немного иначе. В частности, воспользуемся функцией `eval()`, которая позволяет вычислять выражения, заданные в текстовом формате. Например, если некоторое алгебраическое выражение, записанное в соответствии с правилами синтаксиса языка Python, заключить в двойные кавычки, то получится текст. Если этот текст теперь указать аргументом функции `print()`, то на экране появится соответствующее алгебраическое выражение. Если же текст (с алгебраическим выражением) передать аргументом функции `eval()`, то соответствующее алгебраическое выражение будет вычислено. Обратимся к листингу 1.3.

Листинг 1.3. Использование функции eval()

```
a="(5+2)**2-3*2" # Текстовое значение
b="6-5/2"        # Текстовое значение
c="10//4+10%3"   # Текстовое значение
# Результаты вычислений выводим на экран.
# Для "вычисления" текстовых выражений
# используем функцию eval()
print("Результатывычислений:")
print(a+" =",eval(a))
print(b+" =",eval(b))
print(c+" =",eval(c))
```

В результате выполнения этого программного кода получаем следующее:

Результат выполнения программы (из листинга 1.3)

```
Результаты вычислений:
(5+2)**2-3*2 = 43
6-5/2 = 3.5
10//4+10%3 = 3
```

На заметку

Результаты вычислений поясним на примере манипуляций с переменной a, которой в качестве значения присваивается выражение "(5+2)**2-3*2". Это текст. Если бы мы не использовали двойные кавычки, было бы обычное арифметическое выражение. Двойные кавычки арифметическое выражение превращают в текст. Поэтому переменная a ссылается на текстовое значение. Если мы передаем переменную a аргументом функции print(), то на экране появится текстовое содержимое, на которое ссылается переменная - как и должно быть. Если мы указываем переменную a аргументом функции eval(), то выполняется попытка вычислить выражение, которое представлено текстом, на который ссылается переменная a. Функция eval() достаточно "умная". Если ее аргумент - обычный текст (набор слов), то результатом будет этот же текст. Если, как в нашем случае, в тексте"спрятано" арифметическое выражение - будет вычислено значение этого выражения. Вообще, чтобы понять, каков будет результат выполнения функции eval(), нужно представить, что выполняется команда, представленная текстом в аргументе функции. Если задуматься, то легко понять, что это простое обстоятельство открывает перед нами грандиозные перспективы. Например, если мы последовательно выполним команды x=3, y=7 и z="x+y", а затем команду print(z+" =",eval(z)), то получим в области вывода сообщение x+y = 10. Почему так? Весь секрет, очевидно, кроется в том, как обрабатывается значение переменной z функциями print() и eval(). В первом случае ничего особенного не происходит - текст "x+y" обрабатывается как текст. Во втором случае результатом будет значение выражения x+y, "спрятанного" в двойных кавычках. Поскольку до этого переменные x и y получили числовые значения, получаем сумму двух числовых значений.

Побитовые (или двоичные) операторы очень близки к арифметическим в том отношении, что тоже предназначены для работы с числовыми значениями. Только в случае побитовых операторов вычисления выполняются на уровне двоичного кода числа. Для эффективного использования побитовых операторов необходимо неплохо разбираться в двоичном представлении чисел. Приведенная далее врезка предназначена для тех читателей, кто с этой темой знаком не очень хорошо.

На заметку

В двоичном коде число представляется в виде последовательности нулей и единиц. Старший бит используется для определения знака числа: ноль соответствует положительному числу, а единица соответствует отрицательному числу.

В обычной жизни для записи чисел мы используем десять цифр: 0, 1, 2, и так далее, до 9 включительно. С помощью этих десяти цифр мы можем записать любое число. Достигается это благодаря позиционному представлению чисел: число записывается в виде последовательности цифр. Важное значение имеет то, на какой позиции какая цифра находится. Допустим, позиционное представление числа имеет вид $\overline{a_n a_{n-1} \dots a_1 a_0}$ (черточка сверху означает, что речь идет о позиционном представлении числа), где числовые параметры a_0, a_1, ..., a_n могут принимать значения от 0 до 9 включительно. Как определить значение такого числа? Достаточно просто. По определению имеет место соотношение $\overline{a_n a_{n-1} \dots a_1 a_0} = a_0 \cdot 10^0 + a_1 \cdot 10^1 + a_2 \cdot 10^2 + \dots + a_n \cdot 10^n$.

Точно такую же схему мы можем использовать для того, чтобы записывать числа с помощью всего двух цифр - цифры 0 и цифры 1. Такое представление называется двоичным кодом. Представим, что некоторое число в позиционном представлении имеет вид $\overline{b_n b_{n-1} \dots b_1 b_0}$, но только теперь параметры b_0, b_1, ..., b_n могут принимать лишь два значения: 0 или 1. Чему равняется число $\overline{b_n b_{n-1} \dots b_1 b_0}$? Ответ такой: $\overline{b_n b_{n-1} \dots b_1 b_0} = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \dots + b_n \cdot 2^n$.

Основные арифметические операции при двоичном представлении чисел выполняются так же легко, как в привычном нам десятичном представлении. Только немного видоизменяются основные правила. Например, имеют

место такие соотношения: 0+0=0, 1+0=1, 0+1=1, 1+1=10, и так далее. Умножение числа на два сводится к дописыванию в позиционном представлении этого числа в конце нуля. Деление на два четного числа означает "зачеркивание" нуля в правой крайней позиции в двоичном коде (нечетные числа на два без остатка не делятся).

Таким образом, любое число мы можем записать как последовательность нулей и единиц и выполнять с этими числами все те операции, которые мы привыкли выполнять с числами в десятичном формате. Но один вопрос остается открытым: как записывать отрицательные числа? Опять же, если мы пишем число на бумаге, мы просто дописываем перед ним знак "минус". В принципе это же мы можем сделать и в случае, если число представлено двоичным кодом. Но проблема в том, что лист бумаги - не компьютер. Для компьютера нужен другой подход. Обычно используют принцип кодирования отрицательных чисел, который называется "дополнение до нуля". Чтобы понять его основную идею, рассмотрим вспомогательный пример.

Допустим, есть некоторое положительное (это важно!) число, которое обозначим через x (икс). А что такое число $-x$ (минус икс)? Ответ может быть таким: это такое число, которое будучи прибавлено к исходному, даст в результате ноль. То есть фактически по определению должно быть так: $x + (-x) = 0$. От этого момента и будем отталкиваться.

Есть такая операция, как побитовая инверсия (в Python этой цели служит оператор ~). Побитовая инверсия состоит в том, что в двоичном представлении числа все нули заменяются на единицы, а все единицы заменяются на нули. Если у нас есть некоторое положительное число x, то число $\sim x$ получается из числа x заменой нулей и единиц на свои антиподы. Спросим себя: а что будет, если мы вычислим сумму чисел x и $\sim x$? Несложно сообразить, что в результате получится число, двоичное представление которого состоит из единиц. Действительно, на той позиции, где у числа x находится 0, у числа $\sim x$ находится 1. Там, где у числа x находится 1, у числа $\sim x$ находится 0. А мы уже знаем, что 0+1=1+0=1. То есть получаем такое:

$$x + (\sim x) = \underbrace{111\ldots11}_{n \text{ единиц}}.$$

А теперь к полученному результату прибавим число 1 ("в игру вступает" правило 1+1=10).

Получится следующее: $x + (\sim x) + 1 = \underbrace{111\ldots11}_{n\text{ единиц}} + 1 = 1\underbrace{000\ldots00}_{n\text{ нулей}}$. И вот теперь мы вспоминаем, что речь идет не просто о вычислениях, а о вычислениях на компьютере. А в компьютере для запоминания чисел (в двоичном коде) выделяется фиксированное количество битов - то есть фиксированное количество позиций (или разрядов) для записи числа. Если для записи числа нужно больше разрядов, чем это отведено в компьютере, старшие разряды будут потеряны - они попросту игнорируются. Теперь представим, что для записи чисел используется nразрядов. Тогда мы сможем записать число x, число $\sim x$, не будет проблем с их суммой $x + (\sim x)$, но вот при записи результата $x + (\sim x) + 1$ уже понадобится на один бит (разряд) больше. Но для этого бита место не выделено. Поэтому в числе $x + (\sim x) + 1 = 1\underbrace{000\ldots00}_{n\text{ нулей}}$ единица в старшем бите будет потеряна, и мы получим код из n нулей. Но это число 0! Таким образом, с учетом практики компьютерных вычислений получаем результат $x + (\sim x) + 1 = 0$. С другой стороны, мы ранее договорились, что $x + (-x) = 0$. Какой из этого вывод? Очень простой: $-x = \sim x + 1$. Другими словами, чтобы получить двоичный код отрицательного числа, нужно взять двоичный код соответствующего положительного числа, по битам инвертировать его, и к полученному значению прибавить единицу. При этом если у положительных чисел старший бит всегда нулевой, то у отрицательных чисел он всегда единичный. Именно старший бит служит признаком положительности/отрицательности числа. Положительные числа из двоичного кода в десятичный переводятся по формуле $\overline{b_n b_{n-1}\ldots b_1 b_0} = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \cdots + b_n \cdot 2^n$. Если число отрицательное, то эту формулу применять нельзя. Чтобы перевести отрицательное число в двоичном коде в привычное нам десятичное представление, необходимо выполнить такие действия:

- по битам инвертировать двоичное представление отрицательного числа;
- прибавить к полученному результату единицу;

- полученный двоичный код перевести к десятичному представлению по формуле $\overline{b_n b_{n-1} \ldots b_1 b_0} = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \cdots + b_n \cdot 2^n$;
- дописать знак "минус".

Чтобы понять, откуда взялись эти правила, достаточно заметить, что с учетом особенностей представления чисел в компьютере имеет место соотношение $-x + \sim(-x) + 1 = 0$. Последующие рассуждения (с точностью до обозначений) повторяют те, что приводились ранее.

Например, мы хотим узнать, каково двоичное представление для числа -5. Начинаем с того, что определим код для числа 5. Несложно проверить, что это такая последовательность битов: 00...000101. После побитового инвертирования получаем код 11...111010. К полученному коду прибавляем 1, получаем код 11...111011. Это и есть двоичный код числа -5.

Напротив, нам хочется узнать, какое число закодировано в виде 11...110111. Поскольку старший (самый крайний слева) бит равен 1, число отрицательное. Чтобы узнать, что же это за число, выполняем побитовое инвертирование кода. Получаем такой результат: 00...001000. Прибавляем единицу, получаем код 00..001001. Это код числа 9. Следовательно, в исходной последовательности битов 11...110111 было закодировано число -9.

Побитовые операторы перечислены и описаны в таблице 1.4.

Таблица 1.4. Побитовые операторы

Оператор	Описание
~	Побитовая инверсия (унарный оператор - у него один операнд). Результатом является число, получающееся заменой нулей на единицы и единиц на нули в побитовом представлении операнда (сам операнд при этом не меняется)
&	Побитовое И. При вычислении результата сравниваются побитовые представления операндов. Если на одной и той же позиции в операндах стоят единицы, то в числе-результате на этой же позиции будет единица. В противном случае (то есть если хотя бы в одной из двух позиций нуль) в числе-результате на соответствующей позиции будет нуль

Оператор	Описание
|	Побитовое ИЛИ. Сравниваются побитовые представления операндов. Если на одной и той же позиции в операндах стоят нули, то в числе-результате на этой же позиции будет нуль. В противном случае (то есть если хотя бы в одной из двух позиций единица) в числе-результате на соответствующей позиции будет единица
^	Побитовое ИСКЮЧАЮЩЕЕ ИЛИ. Результат вычисляется сравнением побитовых представлений операндов. Если на одной и той же позиции в операндах стоят разные значения (у одного числа нуль, а у другого единица), то в числе-результате на этой же позиции будет единица. В противном случае (то есть если на соответствующих позициях в операндах стоят одинаковые числа) в числе-результате на этой позиции будет нуль
<<	Сдвиг влево. Результат вычисляется так: в побитовом представлении первого операнда выполняется сдвиг влево. Количество разрядов, на которые выполняется сдвиг, определяется вторым операндом. Младшие недостающие биты заполняются нулями
>>	Сдвиг вправо. Для вычисления результата в побитовом представлении первого операнда выполняется сдвиг вправо. Количество разрядов, на которые выполняется сдвиг, определяется вторым операндом. Биты слева заполняются значением самого старшего бита (для положительных чисел это нуль, а для отрицательных единица)

Чтобы проиллюстрировать методы использования побитовых операторов, рассмотрим небольшой программный код, представленный в листинге 1.4.

Листинг 1.4. Побитовые операторы

```
a=70>>3
b=~a
c=a<<1
print(a,b,c)
print(7|3,7&3,7^3)
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 1.4)

```
8 -9 16
7 3 4
```

На заметку

На всякий случай приведем пояснения относительно результатов выполнения программного кода. Результат выражения `70>>3` - это число, получаемое сдвигом битового представления числа 70 на 3 позиции вправо (с потерей младших битов). Двоичный код для числа 70 имеет вид 00...0001000110 (три бита, которые "пропадают" после сдвига вправо, выделены жирным шрифтом). После сдвига на 3 позиции вправо получаем 00...0000001000. Это код числа 8. Такой же результат можно было получить проще, если вспомнить, что сдвиг вправо на одну позицию эквивалентен целочисленному делению (делению без остатка) на число 2. Если 70 трижды поделить без остатка на 2, получим 8 (35 после первого деления, 17 после второго деления, и 8 после третьего деления).

Далее, если применить побитовое инвертирование к числу 00...0000001000 (значение переменной `a`), получим бинарный код 11...1111110111, который соответствует отрицательному числу -9. Желающие могут выполнить проверку самостоятельно, однако если вспомнить, что результатом операции `~a+1` является код для значения `-a` (в данном случае это -8), то несложно догадаться, что `~a` соответствует значению `-a-1` (то есть в данном случае это -9).

Значение выражения `a<<1` получаем сдвигом бинарного кода для значения переменной `a` на одну позицию вправо с заполнением младшего бита нулем (соответствует умножению значения переменной `a` на 2). Поскольку значение переменной `a` равно 8, то значение выражения `a<<1` равняется 16.

В следующих выражениях используются числа 7 (бинарный код 00...000111) и 3 (бинарный код 00..000011). Для побитовых операций | (или), &(и), ^ (исключающее или) получаем следующие результаты:

```
  00 ... 000111
| 00 ... 000011
--------------
7 00 ... 000111

  00 ... 000111
& 00 ... 000011
--------------
3 00 ... 000011


  00 ... 000111
^ 00 ... 000011
--------------
4 00 ... 000100
```

В левом нижнем углу жирным шрифтом выделен результат соответствующей операции в десятичной системе счисления.

С *логическими значениями* мы столкнемся при проверке условий в условной инструкции (операторе). Значений у логического типа всего два: `True` (*истина*) и `False` (*ложь*). Для работы со значениями логического типа предназначены специальные операторы, которые принято называть логическими. Используемые в Python *логические операторы* описаны в таблице 1.5.

Таблица 1.5. Логические операторы

Оператор	Описание
`or`	Бинарный оператор (у оператора два операнда). Логическое *ИЛИ*. В общем случае результатом выражения `x or y` является `True`, если значение хотя бы одного из операндов `x` или `y` равно `True`. Если значения обеих операндов `x` и `y` равны `False`, результатом выражения `x or y` будет `False`. В Python выражения на основе оператора `or` вычисляются по упрощенной схеме: если первый операнд `x` интерпретируется как `True`, то `x` возвращается в качестве результата (второй операнд при этом не вычисляется). Если первый операнд `x` интерпретируется как `False`, то в качестве результата возвращается второй операнд `y`
`and`	Бинарный оператор (у оператора два операнда). Логическое *И*. В общем случае результатом выражения `x and y` является значение `True`, если значения обеих операндов `x` и `y` равны `True`. Если значение хотя бы одного из операндов `x` или `y` равно `False`, результатом выражения `x and y` будет `False`. В Python выражения на основе оператора `and` вычисляются по упрощенной схеме: если первый операнд `x` интерпретируется как `False`, то `x` возвращается в качестве результата (второй операнд при этом не вычисляется). Если первый операнд `x` интерпретируется как `True`, то в качестве результата возвращается второй операнд `y`
`not`	Логическое отрицание. Унарный оператор (у оператора один операнд). Результатом выражения `not x` будет значение `True`, если у операнда `x` значение `False`. Результатом выражения `not x` будет значение `False`, если у операнда `x` значение `True`

На заметку

Нередко на практике используется такая операция, как *логическое исключающее или*. Это бинарная операция. Ее результатом является *истина*, если операнды имеют разные значения. Если операнды имеют одинаковые значения, результатом операции исключающего или является значение *ложь*. Другими словами, исключающее или - это проверка на предмет того, различные ли значения у операндов, или нет. С помощью логических операторов `or`, `and` и `not` операция исключающего или для операндов `x` и `y` запишется как `(x or y) and (not (x and y))`.

Помимо непосредственно логических значений могут использоваться и числовые значения (да и не только). Если числовое значение встречается в том месте, где по идее должно быть значение логического типа, начинается интерпретация нелогического выражения как логического. Интерпретация выполняется так: нулевые числовые значения интерпретируются как `False`, а ненулевые значения интерпретируются как `True`.

На заметку

Ситуация даже более интересная, чем может показаться на первый взгляд. Логические значения `True` и `False` можно использовать, соответственно, как числа `1` и `0`. Например, в арифметических расчетах вместо `1` можем использовать `True`, а вместо `0` можем использовать `False`. Однако к текстовому формату (например, будучи переданными аргументами функции `print()`) значения `True` и `False` приводятся не как `1` и `0`, а как `"True"` и `"False"`. Об этой особенности следует помнить.

Кроме того, при проверке условий или вычислении логических выражений могут использоваться не только непосредственно логические значения или числа, но и иные объекты. Как именно выполняется "логическая" интерпретация таких объектов мы узнаем, когда поближе познакомимся с методами ООП.

Программный код, приведенный в листинге 1.5, дает представление о том, как с применением логических операторов вычисляются логические выражения.

Листинг 1.5. Логические операторы

```
a=True
b=nota
print(a,b)
c=a and b
d=aorb
print(c,d)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 1.5)

```
True False
False True
```

На заметку

В данном случае переменная a ссылается на логическое значение True. Значение переменной b - это значение выражения not a. Поскольку a - это True, то значение выражения not a равно False. Поэтому результатом выражения a and b является значение False, а значением выражения a or b является значение True.

Хотя по своей сути логические операторы должны возвращать логические значение, в Python это далеко не всегда так. Результатом выражений на основе операндов or и and является один из операндов соответствующего выражения. А операнды не обязательно должны быть логического типа - достаточно, чтобы они могли интерпретироваться как логические значения. Ситуацию иллюстрирует следующий пример (программный код в листинге 1.6).

Листинг 1.6. Снова логические операторы

```
x=10       # Числовая переменная
y=20       # Числовая переменная
z=x and y # Логическое И
print(z)   # Результат логического И
z=x or y   # Логическое ИЛИ
print(z)   # Результат логического ИЛИ
# Логическое отрицание
print(not x)
```

В данном случае логические операторы используются с числовыми операндами. При выполнении программного кода получаем такой результат:

Результат выполнения программы (из листинга 1.6)

```
20
10
False
```

Переменные x и y ссылаются на целочисленные значения. Поскольку значения ненулевые, то при выполнении логических операций обе перемен-

ные интерпретируются как такие, что имеют значение `True`. Но интерпретироваться как значение `True` и ссылаться на значение `True` - это далеко не одно и то же.

При вычислении выражения `x and y` сначала проверяется значение первого операнда. Поскольку первый операнд `x` (значение 10) интерпретируется как `True`, в качестве результата выражения возвращается второй операнд - то есть `y` (значение `20`). Аналогично, при вычислении выражения `x or y`, поскольку первый операнд `x` интерпретируется как `True`, он же возвращается в качестве результата. Но результатом является реальное числовое значение, на которое ссылается переменная `x` (то есть значение `10`).

Иначе обстоят дела с оператором логического отрицания `not`. Результатом выражения `not x` является значение `False`. То есть в данном случае логический оператор дает вполне "ожидаемый" логический результат.

Операторы сравнения позволяют сравнивать на предмет равенства/неравенства различные значения. Обычно (хотя и не всегда) речь идет о числовых значениях. Результатом операций сравнения являются логические значения: `True`, если соответствующее соотношение верно (если отношение имеет место), и `False`, если неверно (отношение не имеет место). Здесь и далее мы будем говорить об операциях сравнения только в основном для числовых значений. Операторы сравнения языка Python представлены в таблице 1.6.

Таблица 1.6. Операторы сравнения

Оператор	Описание
`<`	Строго меньше. Результатом является `True`, если значение операнда слева от оператора *меньше* значения операнда справа от оператора. Иначе возвращается значение `False`
`>`	Строго больше. Результатом является `True`, если значение операнда слева от оператора *больше* значения операнда справа от оператора. Иначе возвращается значение `False`
`<=`	Меньше или равно. Результатом является `True`, если значение операнда слева от оператора *не больше* значения операнда справа от оператора. Иначе возвращается значение `False`

Оператор	Описание
>=	Больше или равно. Результатом является True, если значение операнда слева от оператора *не меньше* значения операнда справа от оператора. Иначе возвращается значение False
==	Равно. Результатом является True, если значение операнда слева от оператора *равно* значению операнда справа от оператора. Иначе возвращается значение False
!=	Не равно. Результатом является True, если значение операнда слева от оператора *не равно* значению операнда справа от оператора. Иначе возвращается значение False
is	Оператор проверки идентичности объектов. В качестве результата возвращается значение True, если оба операнда ссылаются на один и тот же объект. В противном случае (то есть если операнды ссылаются на разные объекты) возвращается значение False
is not	Оператор проверки неидентичности объектов. В качестве результата возвращается значение True, если операнды ссылаются на разные объекты. В противном случае (то есть если операнды ссылаются на один и тот же объект) возвращается значение False

Думается, операторы сравнения особых комментариев не требуют (может только за исключением операторов проверки идентичности/неидентичности объектов). Небольшие примеры использования операторов сравнения приведены в листинге 1.7.

Листинг 1.7. Операторы сравнения

```
a=100
b=200
print(a<b,a>=b,a==100,b!-199)
```

Результат выполнения кода такой:

Результат выполнения программы (из листинга 1.7)

```
True False True True
```

На заметку

Если со сравнением числовых значений все более-менее ясно, что операторы `is` и `is not` могут быть не совсем понятны читателю. Здесь уместно вспомнить, что переменные в Python не содержат значения, а ссылаются на них. Мы говорим о переменных, которые ссылаются на значения одного типа. Какие при этом возможны варианты? Во-первых, переменные могут ссылаться на разные значения. Во-вторых, переменные могут ссылаться на одинаковые значения (то есть физически эти значения одинаковые, но каждое из них записано в памяти отдельно). В-третьих, переменные могут ссылаться не просто на одинаковые значения, а на одно и то же значение. Вторая и третья ситуации принципиально разные. Причем если мы просто обращаемся к переменной, то фактически получаем то значение, на которое переменная ссылается. При проверке на предмет равенства (оператор `==`) или неравенства (оператор `!=`) значений, на которые ссылаются две переменные, сравниваются значения, возвращаемые по соответствующим ссылкам. При этом с помощью указанных операторов невозможно проверить, реализуются ли эти значения одним объектом или разными объектами. Для выполнения такой проверки используют операторы `is` и `is not`.

В заключение раздела сделаем несколько важных замечаний. Первое касается так называемых *сокращенных форм оператора присваивания.*

Как мы уже знаем, оператором присваивания в Python служит знак равенства =. Также существуют так называемые сокращенные формы оператора присваивания. Речь идет вот о чем. Если необходимо выполнить команду вида `x=x ☺ y`, где через `x` и `y` обозначены некоторые переменные, а через ☺ мы формально обозначили один из арифметических или побитовых операторов, то эту команду можно записать в виде `x ☺ =y`. Например, вместо команды `x=x+y` можно использовать команду `x+=y`.

Что касается самого оператора присваивания (это второе замечание), то в Python разрешено *многократное* и *множественное* присваивание. При многократном присваивании в одном выражении используется несколько операторов присваивания. Примером такой ситуации может быть выражение `x=y=10`, которым переменным `x` и `y` присваивается значение `10`. Множественное присваивание - это когда в левой части от оператора присваивания указано сразу несколько переменных. Примером такой ситуации может быть команда `a,b=1,2`. В данном случае слева от оператора присваивания через запятую указаны переменные `a` и `b`, а справа (тоже через запятую) - значения `1` и `2`. В результате переменная `a` получает значение `1`, а переменная `b` получает значение `2`. Принцип обработки такого рода выражений следующий: сначала вычисляются значения в правой части от оператора присваивания, а затем эти значения присваиваются переменным, указанным слева от оператора присваивания (количество переменных слева и количество значений справа должны совпадать). Поэтому, например, что-

бы переменные a и b "обменялись" значениями, можно воспользоваться командой `a,b=b,a`.

Вместе с тем, многократное и множественное присваивание следует использовать с крайней осторожностью, поскольку при работе с такими данными, как, например, списки, результаты подобных операций могут быть вполне неожиданными. Вся "сложность" ситуации связана в основном с тем, что в Python (как подчеркивалось и еще будет подчеркиваться) переменные не содержат значения, а ссылаются на них. Для некоторых типов данных этот момент является принципиальным. Все подобные "премудрости" мы будем изучать по мере знакомства с различными типами данных.

Третье замечание касается приоритета различных операторов. В сложных выражениях одновременно могут присутствовать самые разные операторы. Такие выражения вычисляются в соответствии с приоритетом операторов. Сначала вычисляются выражения и подвыражения с операторами, которые имеют более высокий приоритет, а уже затем вычисляются выражения и подвыражения с более низким приоритетом. Если несколько операторов имеют одинаковые приоритеты, выражение вычисляется слева направо. Вкратце приоритет операторов в Python следующий (в порядке уменьшения приоритета):

- побитовая инверсия, возведение в степень, знак числа (унарный минус или унарный плюс);
- умножение, деление, целочисленное деление, остаток от деления;
- сложение, вычитание;
- побитовые сдвиги;
- побитовое *И*;
- побитовое *ИСКЛЮЧАЮЩЕЕ ИЛИ*;
- побитовое *ИЛИ*;
- оператор присваивания (включая и сокращенные формы).

На заметку

Выше упоминался знак числа - это унарный оператор, который ставится перед числовым значением и определяет, положительное число или отрицательное. Для положительных чисел знак числа обычно не ставится (такая вот традиция), хотя никто не запрещает написать перед положительным числом знак +. Перед отрицательными числами ставится знак –.

Для изменения порядка вычисления выражения можно использовать круглые скобки. Более того, наличие в правильном месте круглых скобок (даже если острой необходимости в них нет), не только делает код более "надежным", но и значительно повышает его читабельность.

Далее мы более детально обсудим некоторые особенности числовых данных. Числовые типы интересны сами по себе и часто используются при составлении программных кодов. Прочие типы данных, такие, как текст, списки, множества, кортежи и словари, мы рассмотрим в следующих главах книги.

Числовые данные

- А зачем вы мне это говорите?
- Сигнализирую.
из к/ф "Девчата"

Числовые значения могут быть представлены несколькими типами. Целые числа реализуются с помощью типа `int`, действительные нецелые числа (числа в формате с плавающей точкой) реализуются с помощью типа `float`, а комплексные числа реализуются в виде данных типа `complex`.

На заметку

С математической точки зрения множество целых чисел является подмножеством действительных чисел, которое, в свою очередь, является подмножеством комплексных чисел.

Напомним, что комплексные числа - это числа, представимые в виде $z = x + iy$, где действительные числа x и y называются соответственно действительной и мнимой частью комплексного числа, а мнимая единица i по определению такова, что $i^2 = -1$. Операции с комплексными числами (умножение, деление, сложение и вычитание) выполняются как с обычными алгебраическими выражениями, только лишь с поправкой на то, что $i^2 = -1$. Результатом сложения, вычитания, умножения и деления двух комплексных чисел является комплексное число.

Числом, комплексно сопряженным к числу $z = x + iy$, называется число $\bar{z} = x - iy$ (получается заменой в исходном числе i на $-i$).

Модуль $|z|$ комплексного числа Z - это всегда число действительное, равное корню квадратному из произведения этого числа на комплексно сопряженное: $|z| = \sqrt{z \cdot \bar{z}} = \sqrt{x^2 + y^2}$.

Здесь нас в первую очередь будут интересовать приемы создания числовых литералов. О том, как выполняются числовые расчеты, речь пойдет немного позже - по крайней мере, после того, как мы познакомимся с основными управляющими инструкциями (такими, как условный оператор и операторы цикла).

Обычно числовые литералы набираются в десятичной системе - то есть в той системе, которой мы пользуемся в повседневной жизни. Это десять цифр от 0 до 9 включительно, с помощью которых мы и вводим в программном коде нужное нам целое или действительное число. Целые числа набираются в виде последовательности цифр и, если нужно, знака числа. Действительные числа набираются практически так же, но только у них есть целая и дробная части, а в качестве разделителя целой и дробной частей используем точку.

На заметку

Литерал - это фиксированное значение, которое не может быть изменено в программе. Обычно под литералами подразумевают числа и текст, которые используются явно в программном коде. Например, в команде `x=1.5` через `x` обозначена переменная, а `1.5` - это литерал (числовой). Другой пример: в команде `name="Иван Иванович"` текст `"Иван Иванович"` является литералом, а `name` -это переменная.

Но в принципе в программном коде для целых чисел значения можно указывать не только в десятичной системе счисления, но и в двоичной, восьмерично и шестнадцатеричной.

Двоичную систему счисления мы уже обсуждали. В восьмеричной системе числа записываются с помощью восьми цифр: от 0 до 7 включительно. Если в восьмеричной системе число имеет позиционное представление $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$ (здесь параметры a_0, a_1, ..., a_n могут принимать значения 0, 1, ..., 7), то это на самом деле означает, что речь идет о числе, которое в десятичную систему переводится по формуле

$$\overline{a_n a_{n-1} \ldots a_2 a_1 a_0} = a_0 \cdot 8^0 + a_1 \cdot 8^1 + a_2 \cdot 8^2 + \cdots + a_n \cdot 8^n.$$

Аналогично обстоят дела с шестнадцатеричной системой счисления. Только теперь используется, как несложно догадаться, 16 "символов": десять цифр от 0 до 9 и еще шесть букв от a до f включительно. Эти буквы обозначают числа от 10 до 15 (то есть буква a соответствует числу 10, буква b соответствует числу 11, и так далее, вплоть до буквы f, которая соответствует числу 15).

Если в шестнадцатеричной системе число имеет представление $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$ (но теперь параметры a_0, a_1, ..., a_n могут принимать значения 0, 1, ..., 9, a, b, ..., f), в десятичной системе счисления это число вычисляется так: $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0} = a_0 \cdot 16^0 + a_1 \cdot 16^1 + a_2 \cdot 16^2 + \cdots + a_n \cdot 16^n$.

Если литералы задаются не в десятичной системе счисления, то начинаться они должны со специального префикса, который "идентифицирует", к какой системе счисления относится литерал. Для бинарных чисел (литералы в двоичной системе) префикс состоит из нуля и буквы b (большой или маленькой). Например, литерал `0b101` означает число 5. Вполне законной будет, скажем, команда `x=0b101`, которой переменной x присваивается числовое значение 5.

На заметку

Если бы нам понадобилось присвоить переменной x значение `-5` (то есть отрицательное значение), причем набрать литерал в двоичном коде, то соответствующая команда выглядела бы как `x=-0b101`. То есть мы используем знак "минус" в литерале. Все то, о чем мы говорили выше о бинарном кодировании отрицательных чисел в компьютере, о принципе "дополнения до нуля" - в данном случае нам не нужно. Почему? Потому что там речь шла о том, как отрицательные числа представлены в памяти компьютера. Здесь речь идет о формальной записи числа с использованием двоичного кода. То, что мы пишем в программном коде, предназначено для того, кто будет этот код читать - то есть для программиста, пользователя, читателя. Поэтому здесь никто не запрещает нам использовать в числовом литерале знак "минус", даже если этот литерал представляет число в двоичной системе счисления. Компьютер, когда придет его черед взяться за выполнение программного кода, автоматически переведет указанный литерал в "правильное" с точки зрения компьютера представление. Это же замечание (относительно знака числа) касается и прочих систем счисления: восьмеричной и шестнадцатеричной.

Признаком числа, записанного в восьмеричной системе счисления, является префикс `0o` (ноль и большая или маленькая буква o), а числа, записанные в шестнадцатеричной системе, начинаются с префикса `0x` (ноль и большая или маленькая буква x). Например, число `123` в восьмеричной системе

счисления запишется как `0o173`. Это же число в шестнадцатеричной системе представляется литералом `0x7b`.

Комплексные числа вводятся в "естественном" формате, то есть в виде суммы действительной и мнимой части. Признаком мнимости числа является суффикс `j` (большая или маленькая буква), который размещается сразу после числового значения. Другими словами, если после числового литерала указать (без пробелов или иных разделителей) букву `j`, то получится мнимое число. Например, комплексное число $3 + 2i$ запишется в программном коде как `3+2j`. Мнимая единица i в виде программного кода реализуется как `1j`, и так далее. Также для создания комплексного числа можем воспользоваться функцией `complex()`. Первым аргументом функции передается действительная часть комплексного числа, а второй аргумент - мнимая часть комплексного числа. Так, число $3 + 2i$ можем получить с помощью команды `complex (3,2)`, а мнимой единице i соответствует инструкция `complex(0,1)`.

При вводе очень больших или, напротив, очень маленьких (по модулю) чисел удобно воспользоваться представлением числа в виде *мантиссы* и *показателя степени*. В качестве разделителя мантиссы и показателя степени используют букву `e` (большую или маленькую). Например, числовой литерал `1.2e3` обозначает число $1.2 \cdot 10^3$, а числовой литерал `1.2e-5` соответствует числу $1.2 \cdot 10^{-5}$.

Выше мы уже рассматривали арифметические и побитовые операторы. С целочисленными значениями (данные типа `int`) могут использоваться и те, и другие. С действительными значениями (данные типа `float`) используются арифметические операторы. Также для выполнения математических вычислений предназначен целый ряд встроенных функций. Некоторые полезные в работе математические функции представлены в таблице 1.7.

Таблица 1.7. Некоторые математические функции

Функция	Описание
`abs()`	Вычисление модуля числа. Число, для которого вычисляется модуль, указывается аргументом функции. Может использоваться с комплексными числами. Например, результатом каждого из выражений `abs(5.0)`, `abs(-5.0)` и `abs(3+4j)` является значение `5.0`
`bin()`	Функция предназначена для преобразования числа из десятичной системы счисления в двоичную. Исходное число указывается аргументом функции, а результатом является текстовое представление для двоичного кода числа. Например, результатом выражения `bin(9)` будет текст `"0b1001"`
`complex()`	Функция используется для создания комплексных чисел на основе (переданных аргументами функции) действительной и мнимой частей числа, или на основе текстового представления комплексного числа. Например, результатом выражений `complex(3,4)` и `complex("3+4j")` будет комплексное число `3+4j`
`float()`	Функция используется для преобразования числовых значений и текстовых представлений для действительных чисел в числовые значения типа `float`. Например, результатом каждого из выражений `float(5)`, `float("5")`, `float("5.")` и `float("5.0")` является значение `5.0`
`hex()`	Функция предназначена для преобразования числа из десятичной системы счисления в шестнадцатеричную. Аргумент функции - число в десятичной системе счисления. Результат функции - текстовое представление этого числа в шестнадцатеричной системе. Например, результатом выражения `hex(123)` будет текст `"0x7b"`

Функция	Описание
`int()`	Функция для преобразования объекта (например, текста) в целое число. Если аргументом функции передано действительное число (тип `float`), то результат вычисляется отбрасыванием дробной части в действительном числе. Если аргументом указать текстовое представление целого числа, результатом будет само это число. Причем число (в текстовом представлении) может быть не только в десятичной системе, но и в двоичной, восьмеричной или шестнадцатеричной. В этом случае вторым аргументом указывается целое число, определяющее исходную систему счисления (соответственно, 2, 8 или 16). Например, результатом каждого из выражений `int(123.4)`, `int("123")`, `int("0b1111011",2)`, `int("0o173",8)` и `int("0x7b",16)` является значение 123
`max()`	Функция для вычисления максимального значения из набора чисел. Например, результатом выражения `max(-2,4,9,-1)` является значение 9
`min()`	Функция для вычисления минимального значения из набора чисел. Например, результатом выражения `min(-2,4,9,-1)` является значение -2
`oct()`	Функция предназначена для преобразования числа из десятичной системы счисления в восьмеричную. Аргументом указывается число в десятичной системе счисления, а результатом является текстовое представление этого числа в восьмеричной системе. Например, результатом выражения `oct(9)` будет текст `"0o11"`
`pow()`	Функция для возведения в степень числа. Если у функции два аргумента, то результатом является первое число в степени, определяемой вторым числом. Другими словами, результатом выражения `pow(x,y)` является значение `x**y`. Если у функции три аргумента, то после возведения в степень вычисляется остаток от деления на третий аргумент: то есть результатом выражения `pow(x,y,z)` является значение `(x**y)%z`. Например, результатом выражения `pow(2,3)` будет значение 8, а результатом выражения `pow(2,3,5)` будет значение 3

Функция	Описание
`round()`	Функция предназначена для округления действительных значений до целочисленных. Аргументом указывается округляемое значение. Округление выполняется по таким правилам: • если дробная часть округляемого значения *меньше* 0.5, округление выполняется до ближайшего меньшего целого числа; • если дробная часть округляемого значения *больше* 0.5, округление выполняется до ближайшего большего целого числа; • если дробная часть округляемого значения *равняется* 0.5, округление выполняется до ближайшего четного целого числа. Если функции передать второй аргумент, то он будет определять количество знаков в дробной части, до которых будет выполняться округление (то есть в этом случае округление выполняется не до целого числа, а до действительного с количеством разрядов после запятой, определяемым вторым аргументом функции). Например, результатом выражения `round(4.6)` (дробная часть `0.6`) есть значение `5`, у выражения `round(-4.6)` (дробная часть `0.4`) значение `-5`, у выражения `round(4.4)` (дробная часть `0.4`) значение `4`, у выражения `round(-4.4)` (дробная часть `0.6`) значение `-4`, у выражения `round(4.5)` (дробная часть `0.5`) значение `4`, а у выражения `round(-4.5)` (дробная часть `0.5`) - соответственно, `-4`. Для сравнения: результат выражения `round(1.23456,3)` - число `1.235`

С некоторыми из этих функций мы еще будем иметь дело и познакомимся с ними поближе при решении задач в следующих главах книги.

Большое количество математических функций становится доступным после подключения модуля `math`. Модули обсуждаются далее, но сразу отметим, что ничего сложного в подключении модуля нет: для этого в программу достаточно добавить инструкцию `import math`. После этого, например, можем в программном коде использовать тригонометрические функции, такие, как `sin()` (синус), `cos()` (косинус), `tan()` (тангенс) и многие другие. Правда, для указанного способа импортирования модуля, при вызове функций из этого модуля придется перед именем функции указывать название модуля (разделитель имени модуля и имени функции - точка): например, `math.sin()`, `math.cos()` или `math.tan()`. Также в этом модуле определены иррациональные постоянные: $\pi \approx 3.14159265$ (инструкция `math.pi`) и $e \approx 2.7182818$ (инструкция `math.e`).

Немного позже в книге мы рассмотрим примеры, в которых программные коды, написанные на языке Python, используются для решения вычислительных задач.

Подключение модулей

Ну, понимаете, я за кефиром пошел, а тут такие приключения!

из к/ф "Гостья из будущего"

Обычно под *модулем* подразумевается некоторый файл с программой или блоком программного кода. При написании больших программ не всегда удобно весь программный код хранить в одном файле. Поэтому его разбивают на отдельные части или на модули. Возможен и другой вариант: при написании собственной программы мы хотим использовать часть программного кода, который был создан ранее (и находится в каком-то определенном модуле). Чтобы использовать такой код, необходимо *импортировать* соответствующий модуль. Для импортирования модулей используется инструкция `import`, после которой указывается название подключаемого (импортируемого) модуля. Мы будем импортировать встроенные модули Python - то есть те модули, которые являются составной частью среды разработки на языке Python. Например, если мы хотим импортировать математический модуль `math` (модуль содержит различные математические функции и утилиты), то используем инструкцию `import math`. Если им-

портируемых модулей несколько, то после инструкции `import` имена импортируемых модулей перечисляются через запятую.

После того, как модуль подключен, описанные в нем функции, переменные и другие полезные конструкции можно использовать в программном коде. Но при этом каждый раз необходимо явно указывать имя модуля, в котором описана переменная или функция. Используется так называемый *точечный синтаксис*: сначала указываем имя модуля, и, через точку, имя переменной или название функции (со всеми полагающимися аргументами). Например, если мы хотим использовать переменную, которая описана в модуле, то соответствующая инструкция будет выглядеть как `модуль.переменная`. Причем предварительно командой `import` модуль необходимо импортировать модуль. Более конкретно, в модуле `math` есть переменная `pi` со значением постоянной $\pi \approx 3.14159265$. Если мы хотим увидеть значение этой переменной, то сначала командой `import math` подключаем модуль `math`, а затем командой `print (math.pi)` отображаем значение переменной `pi` из модуля `math`.

Вместо того чтобы использовать имя модуля при обращении к его "содержимому", можем для модуля создать "псевдоним". Модуль подключаем командой в формате `import` модуль `as` имя. Другими словами, в инструкции подключения модуля после имени модуля через ключевое слово `as` можно указать идентификатор, который будет использоваться вместо имени модуля. То есть модуль все равно подключается, но когда мы обращаемся к переменным и функциям этого модуля, то указываем не имя модуля, а идентификатор (тот, который после ключевого слова `as`). Например, если мы подключаем `модуль` командой `import модуль as имя`, то обращаться к `переменной` из `модуля` нужно в формате `имя.переменная`. Если вспомнить о модуле `math` и подключить его командой `import math as m`, то распечатать значение переменной `pi` можем командой `print(m.pi)`.

На заметку

Если мы подключили модуль с "псевдонимом", то обращаться к содержимому модуля придется через "псевдоним" - попытка использовать имя модуля приведет к ошибке.

Очевидным образом бросается в глаза неудобство, связанное с необходимостью при обращении к переменным и функциям модуля в явном виде указывать имя модуля (или его "псевдоним"). Но здесь же кроется и главное преимущество: непосредственно в программном коде мы можем определить переменную (функцию или что-то еще) с таким же именем, как и в подключаемом модуле. В этом случае наличие или отсутствие имени мо-

дуля при обращении к переменной (или функции) позволяет однозначно идентифицировать, о какой программной конструкции идет речь.

Вместе с тем, можно подключать не весь модуль, а только некоторые его утилиты (переменные или функции). Скажем, если из модуля нас интересует только одна переменная, то можем воспользоваться командой `from модуль import переменная`. После этого переменную можно использовать без ссылки на имя модуля. Так, чтобы использовать в программном коде переменную `pi` без указания перед ее именем названия модуля `math`, используем инструкцию импорта `from math import pi`.

Если мы хотим подключить все утилиты из некоторого модуля, полезной будет инструкция `from модуль import *` (то есть после инструкции импорт указываем звездочку).

Тернарный оператор

Ничего особенного. Обыкновенная контрабанда.

из к/ф "Бриллиантовая рука"

Часто возникает необходимость выполнять в программном коде различные команды в зависимости от того, выполняется или нет некоторое условие. Вообще такая задача решается с помощью *условного оператора*, с которым мы познакомимся в следующей главе. Вместе с тем, есть "упрощенная" форма условного оператора, которую, по аналогии с языками C++ и Java, можно было бы назвать *тернарным оператором* (хотя, конечно, рассматриваемая далее конструкция резко контрастирует с обычными представлениями об операторе).

На заметку

Обычно операторы бывают *унарные* и *бинарные*. У унарного оператора один операнд, у бинарного оператора два операнда. *Тернарный* оператор - оператор, у которого три операнда. Один из операндов - это проверяемое условие. Еще два операнда - значения, одно из которых возвращается в качестве результата, в зависимости от проверяемого условия.

Тернарный оператор возвращает значение. Причем это значение зависит от истинности или ложности некоторого условия. Шаблон тернарного оператора такой (жирным шрифтом выделены основные инструкции в выражении для тернарного оператора):

```
значение_1 if условие else значение_2
```

Тернарный оператор - достаточно сложная конструкция. Сначала указывается выражение (`значение_1`), которое определяет значение тернарного оператора при истинности `условия`. Между этим выражением и условием указывается ключевое слово `if`. После `условия` указывается ключевое слово `else`, а затем выражение (`значение_2`), определяющее результат тернарного оператора, если `условие` не выполнено.Небольшой пример использования тернарного оператора приведен в листинге 1.8.

Листинг 1.8. Тернарный оператор

```
# Считывается первое число
a=float(input("Введите первое число: "))
# Считывается второе число
b=float(input("введите второе число: "))
# Первое значение
value_1="Первое число больше второго."
# Второе значение
value_2="Второе число не меньше первого."
# Вызывается тернарный оператор
res=value_1 if a>b else value_2
# Отображается результат
print(res)
```

Результат выполнения этого программного кода может быть таким (жирным шрифтом выделен ввод пользователя):

Результат выполнения программы (из листинга 1.8)

```
Введите первое число: 12
введите второе число: 30
Второе число не меньше первого.
```

Или таким:

Результат выполнения программы (из листинга 1.8)

```
Введите первое число: 30
введите второе число: 12
Первое число больше второго.
```

В данном случае ситуация достаточно простая. Сначала пользователем вводятся два числа, которые записываются в переменные `a` и `b`. Для считывания вводимого пользователем значения мы используем функцию `input()`. Но поскольку эта функция в качестве результата возвращает введенное

пользователем значение в текстовом формате, для надежности преобразуем считанное значение к формату числа с плавающей точкой, для чего передаем результат вызова функции `input()` аргументом функции `float()`.

Значение переменной `res` вычисляется на основе тернарного оператора. В тернарном операторе проверяется условие `a<b`. Если условие истинно, то переменной `res` присваивается текстовое значение из переменной `value_1`. Если условие ложно, то переменной `res` присваивается значение переменной `value_2`. Значение переменной `res` отображается в окне вывода с помощью функции `print()`.

Резюме

Только я тебя прошу - говори спокойно, без ораторского нажима.
из к/ф "Безумный день инженера Баркасова"

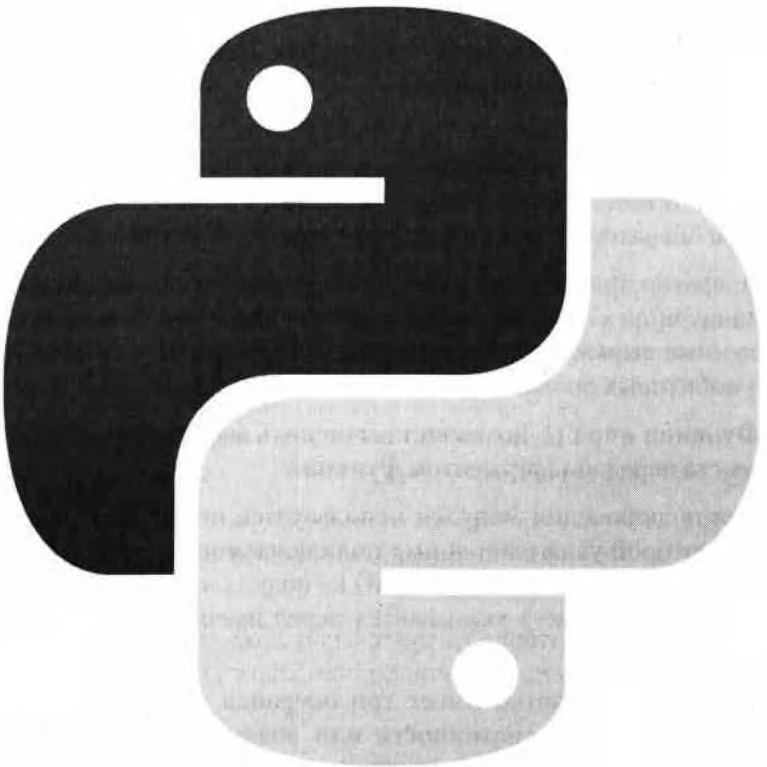
Подведем краткие итоги этой главы.

1. Программа, написанная на Python - это последовательность команд. Для выполнения этих команд используется специальная программа-интерпретатор.
2. В программе могут использоваться переменные. В Python переменная ссылается на значение, а не содержит его, как во многих других языках программирования.
3. В Python существует несколько типов данных. Числовые значения реализуются данными типа `int` (целые числа), `float` (действительные числа) и `complex` (комплексные числа). Для обозначения мнимой части комплексного числа используют букву `j` (большую или маленькую). Бинарные, восьмеричные и шестнадцатеричные литералы вводятся соответственно с префиксами `0b`, `0o` и `0x` (вторая буква после нуля может быть большой или маленькой).
4. Тексту соответствует тип `str`. Текстовые литералы заключаются в двойные (или одинарные) кавычки.
5. С помощью типа `bool` реализуются логические значения. Данные этого типа могут принимать значения `True` (истина) и `False` (ложь). Логические значения являются подвидом целочисленного типа данных, и поэтому могут использоваться в арифметических вычислениях.

6. Тип переменной явно указывать не нужно - он определяется на основе значения, на которое ссылается переменная.
7. Для ввода данных с консоли используют функцию `input()`, а для вывода - функцию `print()`.
8. Для ввода в программный код комментария используют символ `#`. Все, что находится справа от этого символа, интерпретатором игнорируется.
9. Основные операторы Python можно разделить на четыре группы: арифметические, побитовые, логические и операторы сравнения.
10. Арифметические операторы: + (сложение), – (вычитание), * (умножение), / (деление), // (целочисленное деление), % (остаток от целочисленного деления), ** (возведение в степень).
11. Побитовые операторы: ~ (побитовое отрицание), & (побитовое *И*), | (побитовое *ИЛИ*), ^ (побитовое *ИСКЛЮЧАЮЩЕЕ ИЛИ*), << (побитовый сдвиг влево), >> (побитовый сдвиг вправо).
12. Логические операторы: `or` (логическое *ИЛИ*), `and` (логическое *И*), `not` (логическое отрицание).
13. Операторы сравнения: >(больше), <(меньше), <=(не больше), >=(не меньше), ==(равно), !=(не равно). Также к операторам сравнения обычно относят оператор проверки идентичности объектов `is` и оператор проверки неидентичности объектов `is not`.
14. Оператор присваивания имеет сокращенные формы: например, команду вида `x=x+y` можно записать в виде `x+=y`. Такого типа сокращенные выражения можно использовать для всех арифметических и побитовых операторов.
15. Функция `eval()` позволяет вычислить выражение, которое в виде текста передано аргументом функции.
16. Для подключения модулей используется инструкция `import`, после которой указывается имя подключаемого модуля. При использовании переменных (функций) из подключенного модуля имя модуля (через точку) указывается перед именем переменной (функции).
17. Тернарный оператор имеет три операнда и возвращает значение в зависимости от истинности или ложности некоторого условия (один из операндов тернарного оператора).

Глава 2

Управляющие инструкции



Ну, если ты хочешь вращаться, тогда, конечно, вращайся.
из к/ф "Ирония судьбы или с легким паром"

В этой главе речь пойдет об управляющих инструкциях. К управляющим инструкциям в данном случае мы относим *условный оператор* (со всеми его возможными модификациями), а также *операторы цикла* (в языке Python их два). По большому счету, именно управляющие инструкции делают программу действительно программой - то есть целостной конструкцией, а не банальным набором последовательно выполняемых команд. Но обо всем по порядку. В первую очередь рассмотрим условный оператор, причем начнем с самых простых его версий.

Условный оператор

- Будь я вашей женой, я бы тоже уехала.
- Если бы вы были моей женой, я бы повесился.
из к/ф "Иван Васильевич меняет профессию"

Условный оператор (или, как его еще иногда называют, условная инструкция) позволяет в зависимости от истинности или ложности определенного условия выполнять различные блоки программного кода. Общая схема, в соответствии с которой функционирует условный оператор, выглядит так:

- Проверяется некоторое условие: обычно это выражение, значение которого может интерпретироваться как истинное (значение `True`) или ложное (значение `False`).
- Если выражение-условие интерпретируется как истинное, выполняется выделенная специальным образом последовательность команд.
- Если условие интерпретируется как ложное, выполняется другая (но тоже выделенная специальным образом) последовательность команд.
- После того, как одна или другая последовательность команд условного оператора выполнена, управление передается той команде, которая находится после команды вызова условного оператора.

Фактически, речь идет о том, что имеется два набора команд, а решение о том, какой из наборов команд выполнять, принимается по результатам проверки выражения-условия. Таким образом, в программе создается "точка ветвления".

В языке Python условный оператор реализуется в виде программного блока со следующим шаблоном (жирным шрифтом выделены ключевые элементы):

```
if условие:
    команды_1
else:
    команды_2
```

После ключевого слова `if` указывается *условие* - выражение логического типа (или допускающее интерпретацию в терминах `True` и `False`). После *условия* ставится двоеточие (то есть :). Далее следует блок команд, которые выполняются, если *условие* истинно (в шаблоне это *команды_1*). Блок команд выделяется с помощью *отступов* (это стандартный способ выделения блоков команд в языке Python). В принципе, количество отступов может быть произвольным - главное, чтобы для каждой команды блока делалось одно и то же количество отступов. Но общепринятым является *правило делать 4 отступа* (4 пробела).

После окончания блока команд (выполняемых при истинном *условии*) следует ключевое слово `else` (и двоеточие :). Ключевое слово `else` должно быть на том же уровне (такте же количество отступов или пробелов), что и ключевое слово `if`. Далее - еще один блок команд (в шаблоне обозначены как *команды_2*), которые выполняются, если *условие* ложно.

На заметку

В Python несколько команд могут находиться в одной строке. В этом случае команды разделяются точкой с запятой. Также команды можно размещать в одной троке с ключевыми словами `if` и `else`. Но это не тот стиль, которого нужно придерживаться.

Схема выполнения условного оператора иллюстрируется на рис. 2.1.

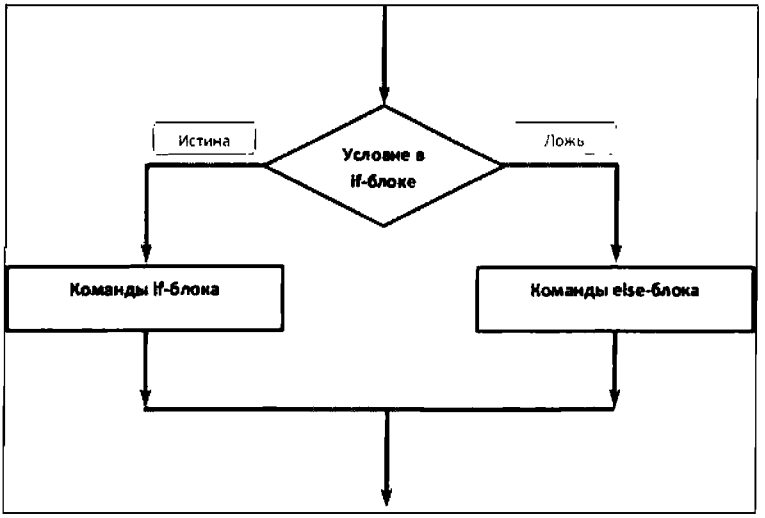


Рис. 2.1. Схема выполнения условного оператора

Пример использования условного оператора приведен в листинге 2.1.

Листинг 2.1. Условный оператор

```
# Пользователь вводит значение
res=eval(input("Введите что-нибудь: "))
# Используем условный оператор для проверки
# типа введенного пользователем значения
if type(res)==int:
    # Если целое число
    print("Вы ввели целое число!")
else:
    # Если что-то другое
    print("Это точно не целое число!")
# После выполнения условного оператора
print("Работа завершена!")
```

При запуске программы на выполнение сначала появляется запрос для ввода пользователем некоторого значения. Введенное пользователем значение считывается и запоминается. Затем проверяется тип введенного пользователем значения: если точнее, проверяется, не целочисленное ли значение ввел пользователь? Для этого используется условный оператор. Если пользователь ввел целое число, выводится соответствующее сообщение. Если то, что ввел пользователь, не является целым числом, тоже появляется со-

общение, но уже другое. Наконец, после завершения выполнения условного оператора появляется сообщение об окончании работы программы.

На заметку

Напомним, что при работе со средой IDLE ввод пользователем значения осуществляется через окно интерпретатора. При работе со средой PyScripter для ввода значения отображается специальное окно с полем ввода.

Понятно, что результат выполнения программного кода зависит от ввода пользователя. Например, если пользователь вводит целое число, результат может быть таким, как показано ниже (жирным шрифтом выделено вводимое пользователем значение):

Результат выполнения программы (из листинга 2.1)

```
Введите что-нибудь: 12
Вы ввели целое число!
Работа завершена!
```

Если пользователь вводит не целое число или вовсе не число, результат будет несколько иным (жирный шрифт - то, что вводит пользователь):

Результат выполнения программы (из листинга 2.1)

```
Введите что-нибудь: 12.0
Это точно не целое число!
Работа завершена!
```

Разница в том, что в первом случае пользователь (по нашему хотению) вводит значение `12`, и это целое число. Во втором случае вводится число `12.0`, и это уже число действительное - наличие десятичной точки превращает целочисленный литерал в литерал для числа с плавающей точкой (тип `float`).

Что касается непосредственно программного кода (см. листинг 2.1), то некоторые команды лучше прокомментировать. Так, мы уже знакомы с функцией `input()`. Она возвращает текстовое представление тех данных, которые вводит пользователь. То есть даже если пользователь вводит число, оно все равно будет считано в текстовом формате (это называется текстовое представление числа). Пикантность ситуации в том, что у нас нет гарантии в том, что пользователь ввел число. В принципе этом может быть какой-угодно текст. Поэтому мы применяем хитрость: результат функции `input()` (а это, напомним, введенное пользователем текстовое значение) передаем аргументом функции `eval()`. Функция `eval()` пытается "вычислить" выражение, "спрятанное" в тексте. Если там спрятано целое число, результатом будет это число. Если действительное число - результатом

будет оно (действительное число). Если просто какой-то текст, то он текстом и останется.

На заметку

При вводе текста его нужно будет заключить в двойные кавычки. Если этого не сделать, возникнет ошибка, причем возникнет она на этапе выполнения функции `eval()`. То есть на запрос ввести значение следует реагировать аккуратно: или вводим число, или текст - но текст обязательно в кавычках!

Результат запоминается в переменной `res`. Дальше в игру вступает условный оператор. Функцией `type()` вычисляем тип введенного пользователем значения. В частности, в условном операторе проверяется выражение `type(res)==int`, которое имеет значение `True` если тип значения, на которое ссылается переменная `res`, равен `int` (то есть если это целое число).

Если пользователь действительно ввел целое число, выполняется команда `print("Вы ввели целое число!")`. В противном случае (если не выполнено условие `type(res)==int`) выполняется команда `print("Это точно не целое число!")`. На этом блок условного оператора заканчивается. Команда `print("Работа завершена!")` выполняется уже после завершения условного оператора.

Также хочется обратить внимание на одно немаловажное обстоятельство: если пользователь введет с формальной точки зрения не число, но выражение, которое возвращает целочисленный результат (например, `1+2**3-4`), эффект будет такой, как если бы пользователь ввел целое число:

Результат выполнения программы (из листинга 2.1)

```
Введите что-нибудь: 1+2**3-4
Вы ввели целое число!
Работа завершена!
```

Описанный выше условный оператор обычно называют `if`-оператором или `if-else`-оператором. Вместе с тем, этот оператор может использоваться и в несколько ином виде. Так, допускается использование *упрощенной формы* оператора без `else`-блока. Шаблон использования условного оператора в этом случае такой (жирным шрифтом выделены ключевые элементы):

```
if условие:
    команды
```

Как и в предыдущем случае, сначала проверяется *условие*, которое указано после ключевого слова `if`. Если *условие* истинно, выполняется блок

команд. После этого управление передается команде, размещенной после условного оператора. Если *условие* ложно, то работа условного оператора на этом заканчивается и выполняется следующая команда после условного оператора. Как выполняется условный оператор в упрощенной форме, иллюстрирует схема на рис. 2.2.

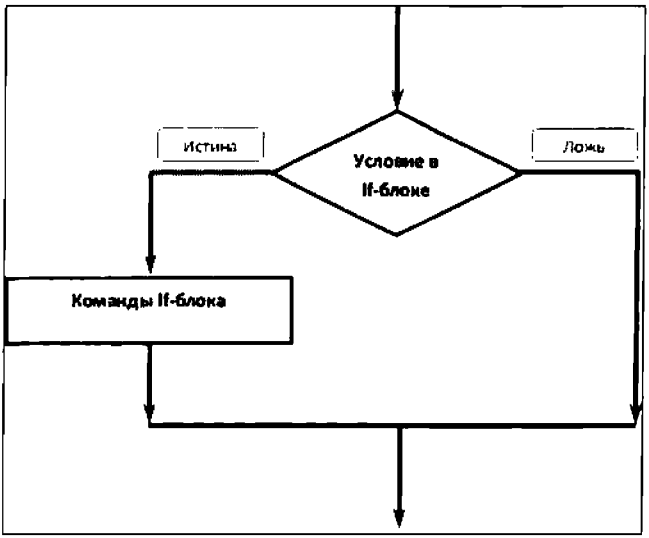


Рис. 2.2. Схема выполнения условного оператора в упрощенном варианте без else-блока

Небольшой пример использования упрощенной версии условного оператора приведен в листинге 2.2.

Листинг 2.2. Упрощенная форма условного оператора

```
# Пользователь вводит значение
res=eval(input("Введите что-нибудь: "))
# Тип значения запоминаем в переменной
resType=type(res)
# Используем условные операторы (упрощенная форма)
# для проверки типа введенного пользователем значения
if resType==int:
    # Если целое число
    print("Это целое число!")
if resType==float:
    # Если действительное число
    print("Это действительное число!")
if resType!=int and resType!=float:
    # Если не число
```

```
    print("Наверное, это текст!")
# После выполнения условных операторов
print("Работа завершена!")
```

Ситуация очень похожа на предыдущую, но есть некоторые отличия. Главное состоит в том, что теперь отслеживаются два типа данных: `int` и `float`. Проверка выполняется с помощью трех условных операторов - каждый в упрощенной форме, следуют один за другим. В первом условном операторе проверяется условие `resType==int` (предварительно переменной `resType` присвоено значение `type(res)`). Это условие истинно, если переменная `res` ссылается на целочисленное значение. Если это действительно так, выполняется команда `print("Это целое число!")`. Если условие ложно, ничего не выполняется (имеется в виду, что ничего не выполняется первым условным оператором).

Во втором условном операторе проверяется условие `resType==float`. На случай истинности такого условия предусмотрена команда `print("Это действительное число!")`. Если условие ложно - на данном этапе ничего не происходит.

В третьем условном операторе проверяется условие `resType!=int and resType!=float`. Условие состоит в том, что тип значения, на которое ссылается переменная `res`, не совпадает с типом `int` и, одновременно, не совпадает с типом `float`. Фактически, это условие выполняется, если не выполнено ни одно из двух условий, которые использовались в предыдущих условных операторах. Так вот, в этом случае выполняется команда `print("Наверное, это текст!")`.

Таким образом, имеется три условных оператора, в каждом из которых проверяется некоторое условие. Условия такие, что выполняться может одно и только одно из них. При истинности того или иного условия выводится соответствующее сообщение. Ниже приведен пример того, как может выглядеть результат выполнения программы, если пользователь вводит целое число (жирным шрифтом, как всегда, выделен ввод пользователя):

Результат выполнения программы (из листинга 2.2)

```
Введите что-нибудь: 12
Это целое число!
Работа завершена!
```

Если ввести действительное число, результат будет выглядеть следующим образом:

Результат выполнения программы (из листинга 2.2)

```
Введите что-нибудь: 12.0
Это действительное число!
Работа завершена!
```

Наконец, если ввести текст, получим такой результат:

Результат выполнения программы (из листинга 2.2)

```
Введите что-нибудь: "Изучаем Python"
Наверное, это текст!
Работа завершена!
```

Еще раз обращаем внимание, что текст нужно вводить в кавычках. И еще раз напоминаем, что связано это с тем, как мы обрабатываем введенное пользователем значение - мы данное значение передаем аргументом функции `eval()`.

Что можно сказать о рассмотренном выше программном коде? Во-первых, он рабочий (то есть выполняется). Во-вторых, он выполняется правильно: если мы вводим целое число, если мы вводим действительное число, и если мы вводим нечто другое (текст) - в каждом из этих случаем появляется "персональное", предназначенное именно для этого случая сообщение.

Однако есть одно "но". Состоит оно в том, что с одной стороны, и мы это отмечали, выполняться может только одно из трех проверяемых в условных операторах условий, а с другой стороны, даже если какое-то условие выполнено, оставшиеся условия все равно будут проверяться. Более конкретно, представим, что в первом же условном операторе условие оказалось истинным.

Данное обстоятельство означает, что два других условия однозначно ложные, но они все равно будут проверены. А это - лишние вычисления. И хотя на времени выполнения столь малого по объему программного кода такая избыточность практически не скажется, некоторая неудовлетворенность от ситуации все же остается. В данном случае разумнее использовать несколько вложенных условных операторов. И на этот случай в Python имеется специальная версия условного оператора.

Одна очень полезная модификация условного оператора позволяет последовательно проверять несколько условий. Если смотреть в корень, то это на самом деле система вложенных условных операторов. Используется следующий шаблон (жирным шрифтом выделены ключевые элементы):

```
if условие_1:
    команды_1
elif условие_2:
    команды_2
elif условие_3:
    команды_3
...
elif условие_N:
    команды_N
else:
    команды
```

В данном случае начало традиционное: после ключевого слова `if` указывается условие для проверки (обозначено *условие_1*). Если условие истинно, выполняются команды `if`-блока (обозначены как *команды_1*). Если выражение *условие_1* ложно, проверяется условие, указанное после ключевого слова `elif` (в шаблоне условие обозначено как *условие_2*, после условия ставится двоеточие `:`).

Если это, второе, условие истинно, выполняются команды для данного `elif`-блока (команды этого блока обозначены как *команды_2*), и на этом работа условного оператора завершается. Если же и выражение *условие_2* ложно, проверяется условие в следующем `elif`-блоке, и в случае его истинности - команды этого блока.

Если при переборе `elif`-блоков ни одного истинного условия не обнаружено, выполняются команды `else`-блока, который размещается в шаблоне самым последним. На рис. 2.3 представлена схема выполнения условного оператора с проверкой нескольких условий.

В листинге 2.3 приведен программный код, в котором идея предыдущего примера реализована через один условный оператор с проверкой нескольких условий.

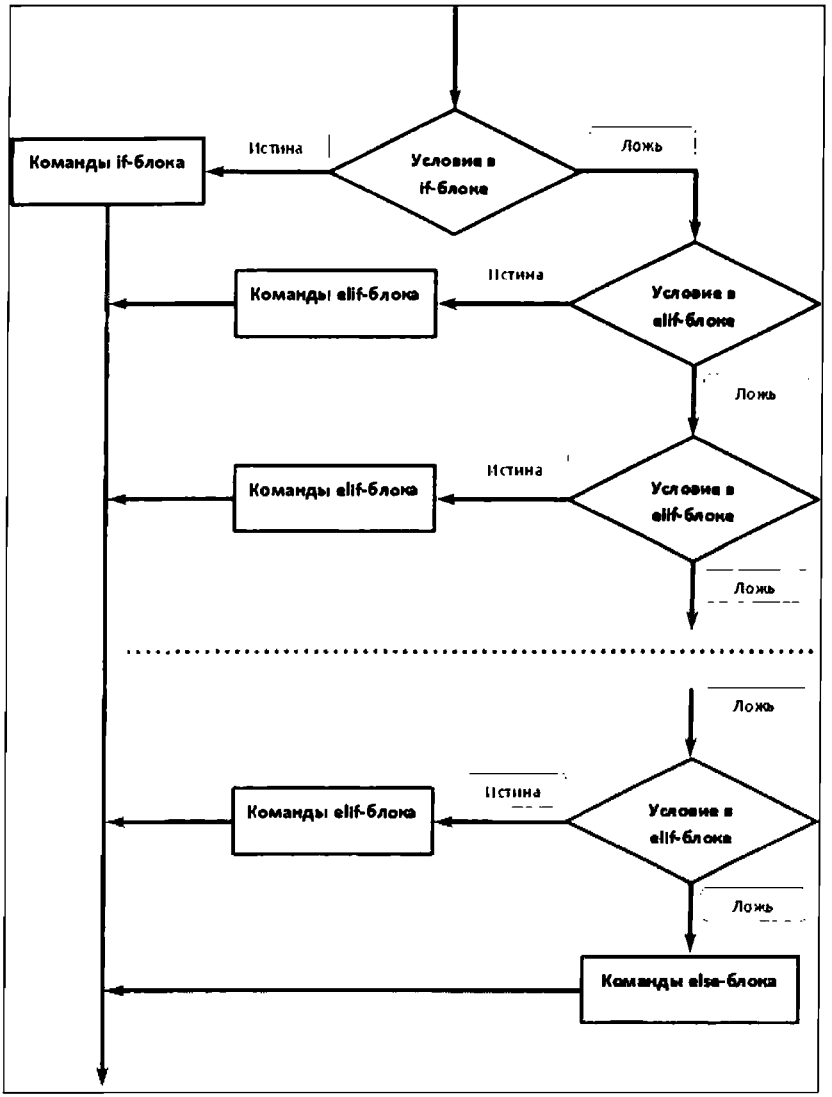


Рис. 2.3. Схема выполнения условного оператора с проверкой нескольких условий

Листинг 2.3. Условный оператор с проверкой нескольких условий

```
# Пользователь вводит значение
res=eval(input("Введите что-нибудь: "))
# Тип значения запоминаем в переменной
resType=type(res)
# Используем условные операторы (упрощенная форма)
```

```
# для проверки типа введенного пользователем значения
if resType==int:
    # Если целое число
    print("Это целое число!")
elif resType==float:
    # Если действительное число
    print("Это действительное число!")
else:
    # Если не число
    print("Наверное, это текст!")
# После выполнения условных операторов
print("Работа завершена!")
```

Результат выполнения этого программного кода точно такой же, как и в предыдущем примере, поэтому останавливаться на этом не будем. Однако алгоритм здесь немного иной. Так, первым делом проверяется условие `resType==int`. Если условие истинно, выводится соответствующее сообщение и работа условного оператора завершается. Это означает, что сразу будет выполнена команда `print("Работа завершена!")` в конце программы, а все "внутренние" условия (точнее, оно там осталось одно) проверяться не будут.

Условие `resType==float` будет проверяться только в том случае, если ложно условие `resType==int`. А что касается `else`-блока, то он выполняется и вовсе лишь в том случае, если не выполнены первые два условия.

Оператор цикла while

Казань брал, Астрахань брал, Ревель брал...
Шпака не брал.
из к/ф "Иван Васильевич меняет профессию"

Операторы цикла позволяют многократно выполнять предопределенный набор или группу команд. В языке Python есть два оператора цикла. Сначала мы познакомимся с оператором цикла, в котором используется `while`-инструкция. Поэтому обычно такой оператор цикла называют *оператором цикла* `while`, или `while`-оператором.

Шаблон у этого оператора цикла по-спартански тривиальный (жирным шрифтом выделены ключевые элементы):

```
while условие:
    команды
```

Принцип работы оператора цикла очень простой: первым делом, как доходит очередь до выполнения этого оператора, проверяется условие, которое указано после ключевого слова while. Если условие истинно, выполняются команды в теле оператора цикла. После выполнения команд снова проверяется условие. Если условие истинно, выполняются команды, а затем проверяется условие, и так далее - до тех пор, пока при проверке условия не окажется, что оно ложно. На этом выполнение оператора цикла заканчивается и выполняется команда, следующая после условного оператора. Схематически процесс выполнения оператора цикла проиллюстрирован диаграммой на рис. 2.4.

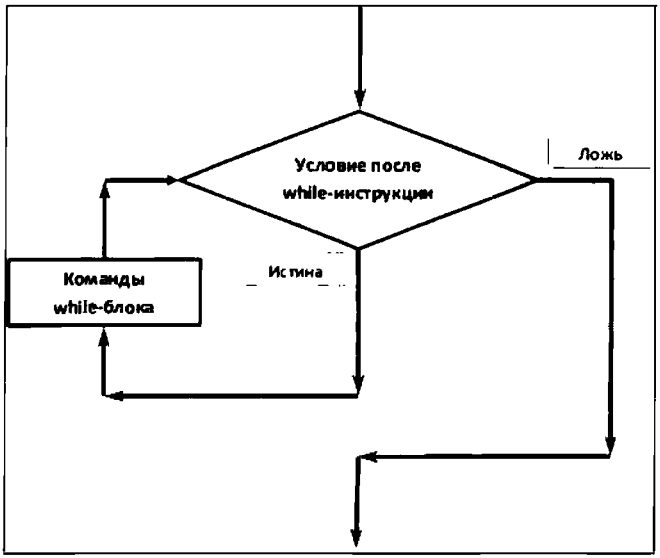


Рис. 2.4. Схема выполнения оператора цикла

У оператора цикла имеется и "расширенная" версия с ключевым словом else. Шаблон для оператора цикла в этом случае таков (жирным шрифтом выделены ключевые элементы):

```
while условие:
        команды_1
    else:
        команды_2
```

Кроме while-блока в этом случае есть еще и else-блок: после ключевого слова else (и двоеточия) указывается блок команд, которые выполняются только в том случае, если условие после инструкции while является ложным. В частности, в самом начале выполнения оператора цикла проверяет-

ся условие после инструкции while. Если условие ложно, выполняются команды в else-блоке и на этом работа оператора цикла завершается. Если условие истинно, выполняются команды while-блока. После этого проверяется условие. Если условие ложно, выполняется else-блок и завершается работа оператора цикла. Если условие истинно, выполняются команды while-блока и снова проверяется условие, и так далее. Схема выполнения такой "расширенной" версии оператора цикла иллюстрируется диаграммой на рис. 2.5.

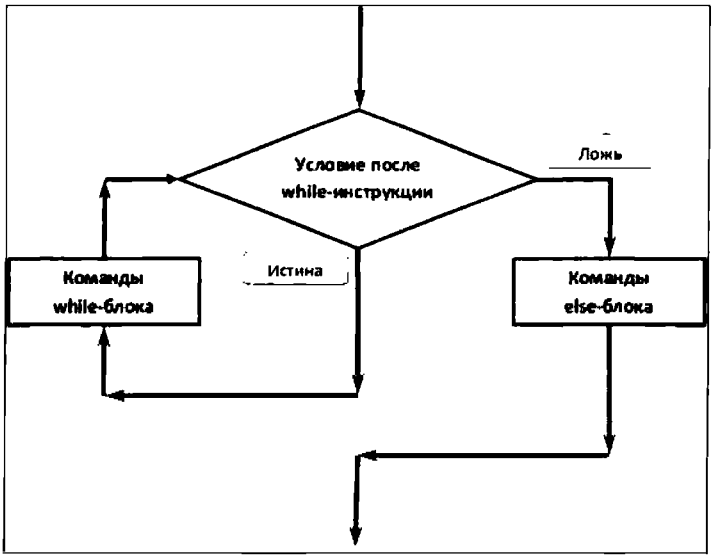


Рис. 2.5. Схема выполнения оператора цикла с else-блоком

Если внимательно проанализировать эту схему, то несложно сообразить, что команды else-блока выполняются один и только один раз, причем на завершающей стадии выполнения оператора цикла. Такого же эффекта можно добиться, если эти команды разместить вне оператора цикла сразу после него. Поэтому в else-блоке было бы мало пользы, если бы не два *ключевых слова* (две инструкции): break и continue. Инструкция break завершает работу оператора цикла без выполнения else-блока. Инструкция continue завершает выполнение текущего цикла и позволяет перейти сразу к проверке условия в while-блоке.

Далее мы проиллюстрируем использование оператора цикла while на нескольких несложных примерах. Начнем с примера, в котором вычисляется сумма натуральных чисел. Соответствующий программный код приведен в листинге 2.4.

Листинг 2.4. Сумма натуральных чисел и оператор цикла

```
print("Сумма натуральных чисел")
n=100 # Количество слагаемых
# Формируем текст для отображения результата
text="1+2+...+"+str(n)+" ="
# Итерационная переменная для оператора цикла
i=1
# Переменная для записи суммы
s=0
# Оператор цикла для вычисления суммы
while i<=n:
    # Добавляем слагаемое к сумме
    s=s+i
    # Изменяем итерационную переменную
    i=i+1
# Отображаем результат
print(text,s)
```

Результат выполнения этого программного кода представлен ниже:

Результат выполнения программы (из листинга 2.4)

```
Сумма натуральных чисел
1+2+...+100 = 5050
```

Основу программного кода составляет оператор цикла, при выполнении которого осуществляются основные вычисления. Перед выполнением оператора цикла командой `print("Сумма натуральных чисел")` выводится сообщение о том, что мы собираемся вычислять сумму натуральных чисел. Далее объявляется несколько переменных.

Переменная `n` (со значением `100`) определяет количество слагаемых в сумме. В данном случае мы будем вычислять сумму `100` натуральных чисел: то есть сумму от `1` до `100`. Также нам понадобится текстовое значение, которое мы выведем в консоли при отображении результата вычислений. С этой целью переменной `text` присваивается значение `"1+2+...+"+str(n)+" ="`. Это текстовое значение. Получается оно объединением (конкатенацией) трех текстовых фрагментов: `"1+2+...+"`, `str(n)` и `" ="`.

Для объединения использован оператор сложения. Вопросы могут возникнуть со вторым "фрагментом", который в данном случае представлен инструкцией `str(n)`. Результатом этого выражения является текстовое представление числового значения, на которое ссылается переменная `n`. Для пе-

ревода числового значения в текстовое мы использовали функцию `str()`, которая для подобных целей и предназначена.

Также мы определяем итерационную (используется в операторе цикла для "индексации" циклов) переменную `i` с начальным значением `1`. Это первое слагаемое, которое мы добавим к сумме. Результат (значение суммы) будем записывать в переменную `s`, у которой начальное значение нулевое. После этого выполняется оператор цикла. Начинается он с инструкции `while`, после которой указано условие `i<=n`. Это означает, что оператор будет выполняться до тех пор, пока значение переменной `i` (итерационная переменная, она же определяет слагаемое, прибавляемое к сумме) не превышает значение переменной `n` (последнее слагаемое в сумме). В теле оператора цикла выполняется две команды. Командой `s=s+i` добавляем очередное слагаемое к сумме, а затем командой `i=i+1` на единицу увеличиваем значение итерационной переменной. После выполнения оператора цикла командой `print(text,s)` отображаем в консольном окне результат вычислений.

На заметку

Вместо команды `s=s+i` можно было использовать эквивалентную ей команду `s+=i`, а вместо команды `i=i+1` - соответственно, команду `i+=1`.

Специфика оператора цикла такова, что в командах цикла должна быть заложена принципиальная возможность для изменения (в результате выполнения этих команд) проверяемого за каждый цикл условия. Иначе можем получить бесконечный цикл. Хотя, с другой стороны, даже формально бесконечный цикл еще не означает, что код составлен неправильно. В листинге 2.5 приведен пример программного кода с бесконечным, на первый взгляд, оператором цикла (который, конечно же, на самом деле бесконечным не является).

Листинг 2.5. Оператор цикла с break-инструкцией

```
print("Сумма натуральных чисел")
n=100 # Количество слагаемых
# Формируем текст для отображения результата
text="1+2+...+"+str(n)+" ="
# Итерационная переменная для оператора цикла
i=1
# Переменная для записи суммы
s=0
# Оператор цикла для вычисления суммы
while True:
    # Добавляем слагаемое к сумме
    s+=i
```

```
    # Изменяем итерационную переменную
    i+=1
    if i>n:
        break
# Отображаем результат
print(text,s)
```

Результат выполнения этого программного кода абсолютно такой же, как и в предыдущем примере. То есть в данном случае мы имеем дело с практически той же задачей, и решаем ее теми же методами. Просто мы немного иначе организовали оператор цикла. А именно, после ключевого слова `while` в качестве условия указано значение `True`. Это константа. Какие бы команды в теле оператора цикла ни выполнялись, условие не изменится. Поэтому чтобы выйти из оператора цикла в нужный момент (завершить работу оператора цикла) в теле оператора мы разместили условный оператор, в котором проверяется условие `i>n`. Если условие истинно, выполняется инструкция `break`. Выполнение этой инструкции приводит к завершению оператора цикла, что нам, собственно, и нужно.

На заметку

В первом примере с оператором цикла условие `i<=n` являлось условием продолжения работы оператора. Во втором примере условие `i>n` в условном операторе является условием завершения оператора цикла. Несложно заметить, что если истинно условие `i<=n`, то ложно условие `i>n`, и наоборот.

Следующий пример, который мы рассмотрим - задача о вычислении площади фигуры, ограниченной двумя кривыми. Здесь, в этом примере, мы используем вложенные операторы цикла (то есть один оператор цикла вызывается в теле другого оператора цикла).

Что касается непосредственно решаемой задачи, то нам предстоит вычислить площадь фигуры, которая ограничена двумя кривыми, уравнения которых $y(x) = x$ и $y(x) = x^2$. Графики этих кривых представлены на рис. 2.6.

Это прямая линия и парабола. Кривые пересекаются в двух точках: в точке $x = 0, y = 0$ и в точке $x = 1, y = 1$ (точки определяются как решение уравнения $x = x^2$). Получается такой своеобразный "лепесток", площадь которого нам и предстоит вычислить.

На заметку

Задача имеет точное решение. А именно, площадь S указанной фигуры $S = \int_0^1 (x - x^2)dx = \frac{1}{6} \approx 0.166667$. Однако мы используем несколько иной

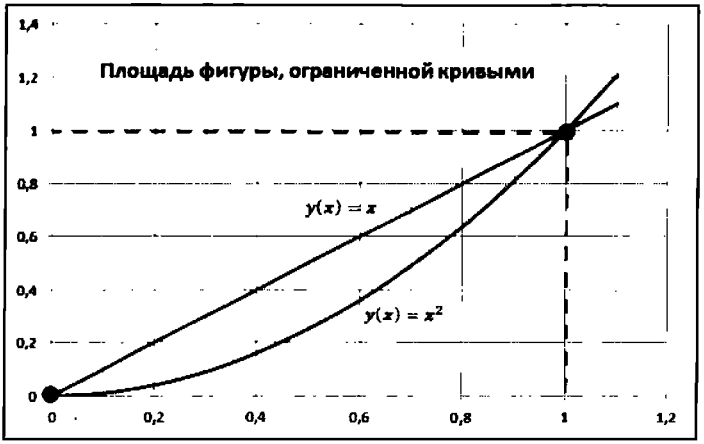


Рис. 2.6. Вычисление области фигуры, ограниченной двумя кривыми

подход. Метод, который описывается далее и применяется нами для вычисления площади фигуры, имеет отношение к теории вероятностей и математической статистике (обычно такой подход называют методами Монте-Карло).

При вычислении площади фигуры мы будем исходить из следующих соображений. Во-первых, замечаем, что область, ограниченная кривыми, полностью попадает в единичный квадрат с левой нижней вершиной в точке начала координат и правой верхней вершиной в точке с единичными координатами. Если мы случайным образом выберем точку внутри этого квадрата, то она с некоторой вероятностью попадает в область, что ограничена кривыми (то есть попадет внутрь "лепестка"). Теория вероятностей утверждает, что эта самая вероятность равна отношению площадей "лепестка" и квадрата. У квадрата с единичной стороной площадь равна единице. Поэтому площадь "лепестка" (которую нам необходимо вычислить) равняется вероятности, с которой случайным образом выбранная точка попадает внутрь "лепестка". Это будет "во-вторых".

В третьих, если взять очень много случайных точек, равномерно распределенных по квадрату, и вычислить отношение количества точек внутри "лепестка" к общему количеству точек, в идеале получим оценку, близкую к вероятности попадания случайно выбранной точки внутрь "лепестка".

Мы всю эту процедуру немного модифицируем и поступим следующим образом. Вместо того чтобы генерировать случайные точки, покроем весь квадрат равноотстоящими узловыми точками. Посчитаем, сколько их попало внутрь "лепестка" (то есть области, ограниченной кривыми), и поделим на

общее количество точек. Это и будет результат. Программный код, в котором реализован такой подход, представлен в листинге 2.6.

Листинг 2.6. Вычисление площади фигуры и операторы цикла

```
# Количество равных интервалов, на которые делятся
# стороны единичного квадрата
n=500
# "Цена деления" - расстояние между соседними точками
dz=1/n;
# Количество точек, которые попадают внутрь области
pts=0
# Начальное значение индекса, определяющего столбец точек
i=0
# Внешний оператор цикла. Перебираем столбцы точек
while i<=n:
    # x-координата точки
    x=dz*i
    # Начальное значение второго индекса для точек столбца
    j=0
    # Внутренний оператора цикла. Перебираем точки
    # в одном столбце
    while j<=n:
        # y-координата точки
        y=dz*j
        # Условный оператор: проверяем, попала ли точка
        # внутрь области
        if y<=x and y>=x**2:
            # Еще одна точка внутри области
            pts=pts+1
        # Значение второго индекса увеличиваем на единицу
        j=j+1
    # Значение первого индекса увеличиваем на единицу
    i=i+1
# Вычисляем площадь фигуры
s=pts/(n+1)**2
# Отображаем результат
print("Площадь фигуры:",s)
```

Результат, который приведен ниже, достаточно близок к точному решению:

Результат выполнения программы (из листинга 2.6)

```
Площадь фигуры: 0.16694355799379285
```

Чтобы понять логику вычислений, имеет смысл мысленно представить, как мы разбиваем каждую из сторон квадрата на определенное количество интервалов. Количество этих интервалов записывается в переменную n (зна-

чение 500). Границы интервалов будем называть *узловыми точками.* Через каждую узловую точку на сторонах квадрата проводим горизонтальные и вертикальные линии. Точки пересечений этих линий - это именно те точки, которые нам нужны. Каждую такую точку можно "идентифицировать" с помощью двух индексов. Первый индекс определяет узловую точку по горизонтали, а второй - узловую точку по вертикали. На пересечении линий, проходящих через эти узловые точки, находится "идентифицируемая" точка внутри квадрата. Если мы зафиксируем первый индекс, и будем брать разные значения для второго индекса, то все соответствующие точки будут находиться на одной вертикальной прямой. Про такие точки будем говорить, что они находятся в одном столбце. Если зафиксировать второй индекс и брать разные значения первого индекса, то все соответствующие точки будут находиться на одной горизонтальной прямой. Про такие точки можем говорить, что они формируют ряд точек. В каждом ряду и в каждом столбце размещено ровно n+1 точек (если учитывать и те точки, что находятся на координатных осях).

Расстояние (по горизонтали или по вертикали) между двумя соседними узловыми точками равняется, очевидно, единице, деленной на количество интервалов, на которые разбивалась каждая из сторон квадрата: такая "цена деления" записывается в переменную dz (значение 1/n). В переменную pts (начальное значение 0) будем записывать количество точек, которые попали внутрь "лепестка".

Начальное значение индекса i, определяющего столбец точек, должно быть нулевым (нулевой индекс соответствует точке на координатной оси). Через первый индекс, напомним, "нумеруются" столбцы точек. Когда запускается внешний оператор цикла, в нем проверяется условие i<=n. Поэтому цикл выполняется до тех пор, пока значение индексной переменной i не превысит значение переменной n.

В теле оператора цикла командой x=dz*i вычисляется координата x (вдоль горизонтальной оси - *абсцисса*) для точек, находящихся в данном столбце (столбец, напомним, определяется значением индекса i). Также поскольку далее мы планируем перебирать точки в столбце, командой j=0 устанавливаем начальное нулевое значение для второго индекса, (определяющего положение внутренних точек). После этого выполняется второй, внутренний оператор цикла. В нем проверяется условие j<=n - то есть второй индекс j будет увеличиваться (об этом мы узнаем позже) до тех пор, пока он не превысит граничное значение n.

В теле внутреннего оператора цикла командой y=dz*j вычисляется координата y для точки в столбце (координата вдоль вертикальной оси - *ордината*). С помощью условного оператора проверяем, попадает ли точка внутрь

"лепестка", и если так, то командой `pts=pts+1` на единицу увеличиваем значение переменной `pts` (в которую, напомним, записывается количество точек, попадающих внутрь "лепестка").

На заметку

В условном операторе проверяется условие `y<=x and y>=x**2`. Это и есть условие попадания точки с координатами x (переменная `x`) и y (переменная `y`) внутрь области, ограниченной кривыми $y = x$ и $y = x^2$. Чтобы точка попадала в эту область, необходимо чтобы одновременно выполнялись два условия. Во-первых, точка должна находиться ниже прямой $y = x$, а это имеет место, если $y \le x$ (нестрогое неравенство - если мы допускаем, чтобы точка могла оказаться не только ниже прямой, но и непосредственно на прямой). Во-вторых, точка должна находиться выше параболы $y = x^2$ (или на самой параболе - такой вариант мы тоже допускаем). Соответствующее условие выглядит как $y \ge x^2$. Следовательно, должно выполняться соотношение $x^2 \le y \le x$. Если перевести это на язык Python, то получим `y<=x and y>=x**2`. Кстати, вместо инструкции `y<=x and y>=x**2` вполне законно можно было использовать выражение `x**2<=y<=x`. Такого типа выражения в Python допустимы.

Перед завершением внутреннего оператора цикла командой `j=j+1` второй индекс увеличивается на единицу. Это последняя команда внутреннего оператора цикла. Внутренний оператор цикла является предпоследней "командой" внешнего оператора цикла. А последняя команда внешнего оператора цикла - это инструкция `i=i+1`, которой на единицу увеличивается значение первого индекса.

После выполнения внешнего оператора цикла, переменная `pts` содержит значение количества точек, которые попадают внутрь "лепестка". Общее количество точек, как несложно догадаться, равняется `(n+1)**2` (в каждом из $n + 1$ столбцов по $n + 1$ точек - всего $(n + 1)^2$ точек). Поэтому если мы поделим одно значение на другое, получим площадь "лепестка". Соответствующее значение вычисляется командой `s=pts/(n+1)**2`. Наконец, командой `print("Площадь фигуры:",s)` отображаем результат.

Кроме оператора цикла `while`, в языке Python есть еще один оператор цикла, который, отталкиваясь от ключевого слова, входящего в выражение для этого оператора, называют оператором цикла `for`.

Оператор цикла for

По сравнению с оператором цикла `while`, оператор цикла `for` можно было бы назвать более "специализированным". С одной стороны, оператор очень

гибкий и эффективный. С другой стороны, его успешное использование может вызвать некоторые трудности идеологического, так сказать, характера. Проще говоря, при работе с оператором `for` (в зависимости от режима его использования) возникает много всяких нюансов. Поэтому мы пока что рассмотрим самые простые и наиболее понятные схемы применения оператора цикла `for`, а уже по ходу книги будем расширять и совершенствовать наши знания и навыки в данном вопросе.

Шаблон использования этого оператора такой (жирным шрифтом выделены ключевые элементы):

```
for элемент in последовательность:
    команды
```

После ключевого слова `for` указывается некий `элемент` (если проще, то название переменной), который последовательно принимает значения из `последовательности` (упорядоченный набор значений). Между `элементом` и `последовательностью` - ключевое слово `in`. Завершается данная синтаксическая конструкция двоеточием `:`. Затем идет блок `команд` оператора цикла. В операторе цикла `команды` выделяются отступами - традиционно, четырьмя.

На заметку

Нередко (но не всегда) в качестве последовательности в операторе цикла `for` используются *списки*. Со списками мы еще не знакомы. Основные сведения о списках (равно как и об иных типах данных, содержащих коллекции значений и подпадающих под определение `последовательности`) представлены в следующих главах. Здесь нас списки интересуют только в контексте использования их в операторе цикла `for`. Важно знать, что список - это набор упорядоченных элементов (оформленных соответствующим образом). Причем элементы могут быть разного типа. Чтобы создать список, достаточно в квадратных скобках через запятую перечислить элементы списка. Например, конструкция `[1,2,3,4,5]` представляет собой список из пяти натуральных чисел.

В качестве `последовательности` можно использовать текстовые значения. В этом случае перебираются буквы в тексте.

Выполняется оператор цикла следующим образом: `элемент` (переменная, указанная перед ключевым словом `in`) принимает первое значение в `последовательности`, указанной после ключевого слова `in`. После этого выполняются `команды` оператора цикла. Затем `элемент` принимает второе значение из `последовательности`, и снова выполняются `команды` оператора цикла. И так далее, пока не будет завешен перебор всех значений в `последовательности`.

На заметку

В блоке команд оператора цикла можно использовать инструкции `break` и `continue`. Эффект такой же, как и при использовании этих инструкций в операторе цикла `while`: выполнение инструкции `break` приводит к завершению оператора цикла, а выполнение инструкции `continue` приводит к завершению текущего цикла и переходу к следующему.

В операторе цикла `for` может использоваться `else`-блок.

Шаблон оператора цикла `for` с `else`-блоком выглядит следующим образом (жирным шрифтом выделены ключевые элементы):

```
for элемент in список:
    команды_1
else:
  команды_2
```

Команды в `else`-блоке выполняются, фактически, по завершении оператора цикла, как мы его понимали бы без `else`-блока. Весь "выигрыш" от `else`-блока в том, что если работа оператора цикла завершается вследствие выполнения инструкции `break`, то команды в блоке `else` не выполняются.

Работу оператора цикла `for` проиллюстрируем на несложных примерах. В первую очередь рассмотрим пример, в котором вычисляется сумма натуральных чисел. Выше мы такой пример рассматривали, но использовали там оператор цикла `while`. Здесь прибегнем к услугам оператора цикла `for`. Еще одно важное новшество - использование функции `range()`, с помощью которой мы будем создавать *виртуальную* последовательность натуральных чисел. Чтобы создать такой объект, как виртуальная последовательность чисел, аргументами функции `range()` передаются первое число виртуальной последовательности и последнее число последовательности плюс один. Например, если нас интересует последовательность чисел от 1 до 5 включительно, то для создания такой последовательности можем воспользоваться инструкцией `range(1,6)`. Таким образом, последнее число в последовательности на единицу меньше второго аргумента функции `range()`.

На заметку

Если функции `range()` передать только один аргумент, то сформированная последовательность будет начинаться с нуля. Если функции `range()` передать три аргумента, то третий аргумент будет определять шаг арифметической прогрессии: первый аргумент определяет начальное числовое значение в последовательности, а каждое следующее получается прибавлением шага прогрессии (третий ар-

гумент) - но не больше, чем значение второго аргумента. Можно также сказать, что если третий аргумент не указан, то по умолчанию это значение равно единице, а если не указан первый аргумент, то его значение по умолчанию нулевое.

Еще одно замечание касается термина *виртуальный*. Почему мы говорим о числовой последовательности, сформированной функцией `range()`, как о виртуальной? Все дело в том, что на самом деле функцией возвращается некоторый объект, который допускает с собой такое обращение, как если бы это была последовательность чисел (впоследствии мы более подробно познакомимся с подобными объектами - когда будем в рамках концепции ООП обсуждать итераторы, функции-генераторы и итерационные объекты). Но если мы присвоим результат функции `range()` некоторой переменной и затем захотим проверить значение этой переменной, последовательности мы не увидим - будет формальное обозначение результата с ссылкой на название функции `range()`.

Чтобы наглядно увидеть, что же "спрятано" внутри, можно воспользоваться функцией формирования списков `list()`. Аргументом этой функции передается результат вызова функции `range()`. Например, результатом выражения `list(range(1,6))` является список `[1,2,3,4,5]`. Однако обращаем внимание читателя, что для использования функции `range()` в операторе цикла `for` никаких дополнительных действий по "визуализации" виртуальной последовательности предпринимать не нужно.

В листинге 2.7 представлен программный код, в котором сумма натуральных чисел вычисляется с помощью оператора цикла `for`.

Листинг 2.7. Вычисление суммы чисел

```
print("Сумма натуральных чисел")
n=100 # Количество слагаемых
# Формируем текст для отображения результата
text="1+2+...+"+str(n)+" ="
# Переменная для записи суммы
s=0
# Оператор цикла для вычисления суммы
for i in range(1,n+1):
    # Добавляем слагаемое к сумме
    s=s+i
# Отображаем результат
print(text,s)
```

При выполнении данного программного кода мы получаем следующий результат:

Результат выполнения программы (из листинга 2.7)

```
Сумма натуральных чисел
1+2+...+100 = 5050
```

По сравнению с примером из листинга 2.4 изменения минимальные и касаются они, в основном, оператора цикла. Так, поскольку в операторе цикла `for` переменная `i` пробегает значения из последовательности чисел от `1` до `n` (последовательность создается выражением `range(1,n+1)`), то отпала необходимость присваивать переменной `i` начальное значение. Также в операторе цикла нет нужды в команде увеличения на единицу значения итерационной переменной `i`: перебор значений выполняется автоматически. Поэтому без всяких дополнительных команд переменная `i` пробегает значения от `1` до `n` включительно.

На заметку

Сумму можно вычислить с помощью встроенной функции `sum()`. Аргументом функции передается список элементов, сумма которых вычисляется. Так, сумму натуральных чисел от `1` до `n` (если значение этой переменной задано) можем вычислить командой `sum(range(1,n+1))`. Здесь аргументом функции `sum()` передается виртуальная последовательность `range(1,n+1)` - так тоже можно делать.

Следующий пример иллюстрирует, как в операторе цикла `for` в качестве последовательности для перебора значений используется текст. Если более конкретно, то мы берем в качестве основы текст, и затем с помощью оператора цикла `for` распечатываем текст по буквам с небольшими текстовыми вставками. Теперь обратимся к программному коду в листинге 2.8.

Листинг 2.8. Текст в операторе цикла

```
# Текст для оператора цикла
txt="Python"
# Переменная для нумерации букв
i=1
# Оператор цикла
for s in txt:
    # Формируем вспомогательный текст
    t=str(i)+"-я буква:"
    # Выводим сообщение
    print(t,s)
    # Изменяем номер буквы
    i=i+1
# Команда после завершения оператора цикла
print("Работа программы завершена!")
```

В переменную txt записывается базовый текст, который мы планируем "перебирать" по буквам в операторе цикла. Также мы объявляем с начальным единичным значением переменную i. Мы при выводе букв из текста txt собираемся указывать номер каждой буквы, и для этого нам понадобилась отдельная переменная. Но отдельно подчеркнем: хотя в операторе цикла эта переменная и используется, перебор элементов последовательности (букв в тексте) выполняется с помощью другой перемешной - мы ее назвали s. Инструкция s in txt в операторе цикла после ключевого слова for означает, что переменная s будет последовательно принимать буквенные значения на основе текста переменной txt. Поэтому о переменной s мы можем думать как об очередной букве в тексте txt. Причем перебираются буквы строго в той последовательности, как они находятся в тексте.

В операторе цикла командой t=str(i)+"-я буква:" мы формируем и записываем в переменную t вспомогательный текст, который получается объединением текстового представления порядкового номера i буквы в тексте и текста "-я буква:". Для перевода числового значения в текстовое использована функция str().

В результате выполнения команды print(t,s) в строке вывода вспомогательный текст с номером буквы и сама буква. После этого командой i=i+1 на единицу увеличивается номер для следующей буквы.

После завершения оператора цикла командой print("Работа программы завершена!") выводится финальное сообщение. Ниже приведен результат выполнения программы:

Результат выполнения программы (из листинга 2.8)

```
1-я буква: P
2-я буква: y
3-я буква: t
4-я буква: h
5-я буква: o
6-я буква: n
Работа программы завершена!
```

Следующий пример иллюстрирует использование инструкции break и блока else в операторе цикла for. В программе проверяется тип элементов списка. Мы ищем текстовые элементы. Для этого в программе используется оператор цикла. В операторе цикла последовательно перебираются элементы списка, и их тип проверяется на предмет того, текст это или нет. Если в списке нет текстовых элементов, программой выводится сообщение соответствующего содержания. Но если при переборе элементов текст, все же, встречается, то выполнение оператора цикла заканчивается и програм-

ма сообщает пользователю, что в списке среди элементов есть текст. Программный код, в котором реализована данная идея, представлен в листинге 2.9.

Листинг 2.9. Оператор цикла с else-блоком

```
# Начинаем проверку списка
print("Проверяем содержимое списка:")
# Список для проверки. Текста не содержит.
# При проверке альтернативного списка следует
# пометить как комментарий следующую строку
myList=[1,3+2j,True,4.0]
# Альтернативный список с текстом.
# При проверке этого списка следует
# отменить комментарий для
# следующей строки (и удалить пробел)
# myList=[1,3+2j,"Python",4.0]
# Отображаем содержимое списка
print("Список:",myList)
# Оператор цикла для проверки типа элементов списка
for s in myList:
    # Проверяем элемент на "текстовость"
    if type(s)==str:
        # Если элемент текстовый
        print("В списке есть текстовые элементы!")
        # Завершается выполнение оператора цикла
        break
# Блок else оператора цикла.
# Выполняется только если не выполнялась
# инструкция break
else:
    # Отображается сообщение об отсутствии в
    # списке текстовых элементов
    print("В списке нет текстовых элементов!")
# Сообщение о завершении проверки
print("Проверка закончена.")
```

В программе командой `myList=[1,3+2j,True,4.0]` объявляется список `myList`, который мы и будем проверять на наличие в нем текстовых элементов (в данном случае, очевидно, таких не имеется, но никто не запрещает нам изменить список). Чтобы было видно, какой именно список проверяется, командой `print("Список:",myList)` выводим содержимое списка в консольное окно. После этого запускается на выполнение оператор цикла. Переменная `s` пробегает набор значений элементов из списка `myList`. Для каждого значения переменной `s` в условно операторе выпол-

няется проверка выражения `type(s)==str`. Выражение истинно, если тип значения, на которое ссылается переменная `s`, равен `str` (то есть если переменная ссылается на текстовое значение). В этом случае командой `print("В списке есть текстовые элементы!")` выводится сообщение о наличии в списке текстовых элементов, а затем инструкцией `break` завершается работа оператора цикла. Если выражение `type(s)==str` ложно, то ничего этого не происходит и начинает проверяться следующий элемент списка.

Если при переборе элементов списка ни один из них не оказался текстовым, основная часть оператора цикла заканчивается "ничем": ничего не происходит. Но это еще не окончание оператора цикла. У него есть `else`-блок, который "вступает в игру" если в теле оператора цикла не выполнялась инструкция `break`. В `else`-блоке выполняется всего одна команда `print("В списке нет текстовых элементов!")`. Таким образом, если `break`-инструкция не выполнялась (а это означает, что в списке нет текстовых элементов), то появится сообщение из `else`-блока. Если инструкция `break` выполнялась, сообщение из `else`-блока не появится. Зато в этом случае появится сообщение из условного оператора (выполняется перед `break`-инструкцией).

Результат выполнения программы для случая, когда в проверяемом списке нет текстовых элементов, показан ниже:

Результат выполнения программы (из листинга 2.9)

```
Проверяем содержимое списка:
Список: [1, (3+2j), True, 4.0]
В списке нет текстовых элементов!
Проверка закончена.
```

Если в списке текстовые элементы есть, результат выполнения программы может быть следующим:

Результат выполнения программы (из листинга 2.9)

```
Проверяем содержимое списка:
Список: [1, (3+2j), 'Python', 4.0]
В списке есть текстовые элементы!
Проверка закончена.
```

На заметку

Для удобства в исходном программном коде есть две команды объявления списка `myList`, но только одна выполнена в виде комментария. В первом случае список не содержит текстовых элементов, во втором - содержит. При необходимости одну команду можно выделить комментарием, в другой комментирование отменить.

Также обращаем внимание и напоминаем читателю, что текст в Python можно выделять как двойными, так и одинарными кавычками. При выводе содержимого списка с текстовым элементом текст отображается в одинарных кавычках, хотя в программном коде для этой цели мы использовали двойные кавычки.

В следующем примере мы решим задачу о поиске совпадающих элементов в двух списках. При решении мы воспользуемся двумя операторами цикла `for`, причем в одном операторе цикла будет вызываться другой - то есть речь идет о вложенных операторах цикла. Рассмотрим программный код в листинге 2.10.

Листинг 2.10. Совпадение элементов в списках

```
print("Поиск совпадающих элементов.")
# Первый список
A=[2,False,9.1,2-1j,"hello",5,"Python"]
# Второй список
B=[5,3,"HELLO",7,12.5,"Python",True,False]
# Отображаем содержимое 1-го списка
print("1-й список:",A)
# Отображаем содержимое 2-го списка
print("2-й список:",B)
print("Совпадают:")
# Индекс для нумерации элементов 1-го списка
i=0
# Внешний оператор цикла.
# Перебираем элементы 1-го списка
for a in A:
    # Новый индекс элемента из 1-го списка
    i=i+1
    # Индекс для нумерации элементов 2-го списка
    j=0
    # Внутренний оператор цикла.
    # Перебираем элементы 2-го списка
    for b in B:
        # Новый индекс элемента из 1-го списка
        j=j+1
        # Условный оператор. Проверяем
        # равенство элементов
        if a==b:
            # Если элементы совпадают
            txt=str(i)+"-й элемент из 1-го списка и "
            txt=txt+str(j)+"-й элемент из 2-го списка"
            print(txt)
# Завершение программы
```

```
print("Поиск закончен.")
```

В программе объявляются два списка, A и B, их элементы указываются явно. Содержимое списков отображается в консольном окне. Кроме списков, мы используем две переменные. Переменная `i` предназначена для запоминания индексов элементов из первого списка, а переменная `j` служит той же цели, но только по отношению ко второму списку. Переменная `i` срезу получает начальное нулевое значение. Это происходит как раз перед началом выполнения внешнего оператора цикла, в котором переменная `a` пробегает значения из списка A. То есть во внешнем цикле перебираются элементы первого списка.

В операторе цикла командой `i=i+1` задается значение порядкового номера для элемента первого списка, который предполагается сравнивать с элементами второго списка. Также переменная `j` получает нулевое значение (эта переменная определяет порядковый номер элемента во втором списке). Перебор элементов второго списка осуществляется во внутреннем операторе цикла. В этом внутреннем операторе цикла переменная `b` последовательно принимает значения из списка B.

Во внутреннем операторе цикла командой `j=j+1` на единицу увеличивается значение переменной `j`, так чтобы первому элементу в списке соответствовал первый порядковый номер. После этого в условном операторе проверяется условие `a==b`. Выполнение этого условия означает, то элементы списков совпадают. В этом случае двумя командами `txt=str(i)+"-й элемент из 1-го списка и "` и `txt=txt+str(j)+"-й элемент из 2-го списка"` формируется текст для отображения, а командой `print(txt)` этот текст отображается в консольном окне (окне вывода).

Таким образом, если элементы списков совпадают, появляется сообщение о том, какие (по порядковому номеру) элементы в первом и втором списках совпадают. Результат выполнения этой программы выглядит следующим образом:

Результат выполнения программы (из листинга 2.10)

```
Поиск совпадающих элементов.
1-й список: [2, False, 9.1, (2-1j), 'hello', 5, 'Python']
2-й список: [5, 3, 'HELLO', 7, 12.5, 'Python', True, False]
Совпадают:
2-й элемент из 1-го списка и 8-й элемент из 2-го списка
6-й элемент из 1-го списка и 1-й элемент из 2-го списка
7-й элемент из 1-го списка и 6-й элемент из 2-го списка
Поиск закончен.
```

На заметку

Выше в программном коде текст (переменная `txt`) для отображения в консольном окне при совпадении элементов списков формировался в несколько этапов по очень простой причине: одна команда выглядела бы достаточно громоздко.

Из приведенного примера видно, что при сравнении текстовых переменных имеет значение не только буквенный состав текста, но и регистр букв (большие или маленькие).Также стоит заметить, что в нашей программе не предусмотрена обработка случая, когда совпадений в списках нет. Ничего страшного в этом случае не произойдет, но результат будет выглядеть не очень эстетично.

Хотя условные операторы и операторы цикла представляют собой достаточно мощное средство и позволяют решать многие нетривиальные задачи, в Python существует еще одна синтаксическая конструкция, которая в известном смысле может быть отнесена к управляющим инструкциям - во всяком случае, один из возможных вариантов ее использования сводится к организации точек ветвления в программе. Речь идет об обработке *исключительных ситуаций.*

Обработка исключительных ситуаций

- Не ходи туда, там тебя ждут неприятности.
- Но как же туда не ходить? Они же ждут!
из м/ф "Кто сказал "мяу"

Какой бы витиевато-изощренный программный код ни создавался, "обойти" все "опасные места" и предусмотреть все возможные варианты развития ситуации далеко не всегда удается. Особенно эта проблема актуальна, когда в программе часть данных считывается из файла или вводится в программу пользователем уже в процессе выполнения программы. В этих случаях велика вероятность получения программой некорректных данных. В таких случаях при выполнении программы могут возникать ошибки. Самое главное и наиболее неприятное следствие возникновения ошибки - это преждевременное "аварийное" завершение программы. В Python существует механизм, который позволяет программе выполняться даже после того, как возникла ошибка. Все это называется общим и емким термином: *обработка исключительных ситуаций.*

Ошибки, или исключительные ситуации, бывают разные. Обычно выделяют три типа ошибок:

- Ошибки, связанные с неправильным синтаксисом команд. Другими словами, это ошибки, связанные с неправильно набранным кодом. Такие ошибки выявляются просто и без особых усилий: при попытке за-

пуска на выполнение программы с неправильными командами, скорее всего, будет выведено сообщение о месте ошибки и причине ее возникновения.

- Ошибки, связанные с "неправильным" алгоритмом выполнения программы. То есть ошибки, которые связаны с тем, что программа как таковая составлена неправильно (хотя с формальной точки зрения все команды корректные). Это очень "коварные" ошибки, поскольку обычно никаких предупреждающих сообщений не появляется, программа работает, а результат - неправильный. Возможное решение проблемы: тестирование программы на модельных примерах или задачах, для которых известен правильный результат.
- Ошибки времени выполнения, возникающие в процессе выполнения программы и связанные с некорректностью переданных в программу данных, недоступностью ресурсов и пр.

Понятно, что далеко не для каждой ошибки можно выполнить обработку так, чтобы программа продолжала выполняться. Однако в некоторых случаях такая обработка может быть выполнена. Собственно о таких потенциально "отлавливаемых" ошибках, их перехвате и обработке речь будет идти далее.

Общая идея, заложенная в основу метода обработки исключительных ситуаций, такая: программный код, в котором теоретически может возникнуть ошибка, выделяется специальным образом - образно говоря, "берется на контроль". Если при выполнении этого программного кода ошибка не возникает, то ничего особенного не происходит. Если при выполнении "контролируемого" кода возникает ошибка, то выполнение кода останавливается и автоматически создается *исключение* - объект, содержащий описание возникшей ошибки. С практической точки зрения мы можем думать об исключении как о некотором сообщении, которое генерируется интерпретатором в силу возникшей ошибки при выполнении кода или из-за некоторых других обстоятельств, близких по своей природе к ошибке выполнения кода. Хотя ошибка и исключение - это не одно и то же (исключение является следствием ошибки), мы, если это не будет приводить к недоразумениям, обычно будем отождествлять эти понятия.

Для обработки исключительных ситуаций в языке Python используется конструкция `try-except`. Существуют разные вариации использования этой конструкции. Мы начнем с наиболее простых ситуаций.

Можем поступить следующим образом: после ключевого слова `try` и двоеточия размещается блок программного кода, который мы подозреваем на предмет возможного возникновения ошибки. Этот код будем называть

основным или *контролируемым*. По завершении этого блока указывается ключевое слово `except` (с двоеточием), после которого идет еще один блок программного кода. Этот код будем называть *вспомогательным* или *кодом обработки исключения*. То есть шаблон такой (жирным шрифтом выделены ключевые элементы):

```
try:
    # основной код
except:
    # вспомогательный код
```

Если при выполнении основного кода в блоке `try` ошибка не возникла, то вспомогательный программный код в блоке `except` выполняться не будет. Если при выполнении основного кода в блоке `try` возникла ошибка, то выполнение кода `try`-блока прекращается, и выполняется вспомогательный код в блоке `except`. После этого управление передается следующей команде после конструкции `try-except`.

Пример использования конструкции `try-except` в описанном выше формате приведен в листинге 2.11. В этом примере речь идет о решении линейного уравнения вида . Это уравнение имеет очевидное решение , но это при условии, что параметр отличен от нуля. Поскольку параметры и вводятся пользователем, то, как говорится, возможны варианты.

Листинг 2.11. Обработка исключительной ситуации

```
print("Решаем уравнение ax = b")
# Начало try-блока (основной код)
try:
    # Определяем первый параметр. Возможна ошибка
    # при преобразовании текста в число
    a=float(input("Введите a: "))
    # Определяем второй параметр. Возможна ошибка
    # при преобразовании текста в число
    b=float(input("Введите b: "))
    # Решение уравнения. Возможна ошибка
    # при делении на ноль
    x=b/a
    # Отображается значение для корня уравнения.
    # Команда выполняется, если до этого не возникли
    # ошибки
    print("Решение уравнения: x =",x)
# Начало except-блока (вспомогательный код)
except:
    # Команда выполняется только если ранее
    # при выполнении основного кода возникла ошибка
```

```
    print("Вы ввели некорректные данные!")
# Команда выполняется после блока try-except
print("Спасибо, работа программы завершена.")
```

Результат выполнения этого программного кода зависит от того, какое значение вводит пользователь - в том числе при некорректно введенном значении может возникнуть ошибка, и такие ситуации обрабатываются в программном коде. Более конкретно, в начале выполнения программы пользователю предлагается ввести значения для переменных `a` и `b` (команды `a=float(input("Введите a: "))` и `b=float(input("Введите b: "))`). В данном случае введенные пользователем значения преобразуются к числовому формату.

На заметку

Для преобразования текстового представления числа к числовому значению (в формате с плавающей точкой) использована функция `float()`.

Понятно, что если пользователь введет неправильное значение (то есть не число), то возникнет ошибка. Поэтому соответствующие команды размещены в блоке `try`. В этом же блоке есть команда `x=b/a`, которой на основе значений переменных `a` и `b` присваивается значение переменной `x`. До этой команды "очередь" дойдет, только если будут успешно выполнены предыдущие по присваиванию значений переменным `a` и `b`. Но даже когда все прошло гладко на предыдущих этапах, нас вполне может ждать новый сюрприз: если переменной `a` присвоено нулевое значение (никто не запрещает пользователю так поступить), то при выполнении команды `x=b/a` возникнет ошибка деления на ноль. Поэтому данная команда также помещена в блок `try`. В этом блоке есть еще одна команда - речь об инструкции `print("Решение уравнения: x =",x)`, которой отображается значение для корня уравнения.

Помимо блока `try`, в программном коде имеется блок `except`. В этом блоке всего одна команда `print("Вы ввели некорректные данные!")`. Блок `except` "вступает в игру" только если в блоке `try` возникла ошибка. Другими словами, команды в блоке `try` выполняются одна за другой, и пока не возникает ошибка, все происходит так, как если бы они были самыми обычными командами, без всякого блока `try`. Если ошибка так и не возникла, по завершении выполнения блока `try` команды в блоке `except` не выполняются, а сразу начинают выполняться команды после конструкции `try-except`. В данном случае это команда `print("Спасибо, работа программы завершена.")`. Если на каком-то этапе в блоке `try` появилась ошибка, выполнение команд блока `try` прекращается, и начинают вы-

полняться команды (в нашем примере она одна) в блоке except. Затем выполняются команды после конструкции try-except (то есть последняя команда в программном коде выполняется в любом случае - независимо от того, была ошибка в try-блоке, или нет).

Ниже показано, как может выглядеть результат выполнения программы в случае, если все данные пользователем вводятся корректно (здесь и далее ввод пользователя выделен жирным шрифтом):

Результат выполнения программы (из листинга 2.11)

```
Решаем уравнение ax = b
Введите a: 5
Введите b: 8
Решение уравнения: x = 1.6
Спасибо, работа программы завершена.
```

Если вместо числового значения ввести нечто другое, можем получить совсем иной результат:

Результат выполнения программы (из листинга 2.11)

```
Решаем уравнение ax = b
Введите a: text
Вы ввели некорректные данные!
Спасибо, работа программы завершена.
```

Практически такой же результат получаем, если для переменной a указать нулевое значение:

Результат выполнения программы (из листинга 2.11)

```
Решаем уравнение ax = b
Введите a: 0
Введите b: 3
Вы ввели некорректные данные!
Спасибо, работа программы завершена.
```

В принципе, схема обработки исключительных ситуаций, описанная выше, во многом проста и удобна. Но и она не лишена недостатков. В первую очередь сразу бросается в глаза, что разные ошибки обрабатываются одинаково. То есть, какая бы ни возникла ошибка, реакция программы будет одна и та же.

Более того, ситуация может быть еще более неоднозначной. Допустим, мы случайно (или не очень) ошиблись в написании названия, например, функции float() в команде a=float(input("Введите a: "). При запуске на

выполнение программного кода получим такую последовательность сообщений:

```
Решаем уравнение ax = b
Вы ввели некорректные данные!
Спасибо, работа программы завершена.
```

Почему так происходит? Потому что в использованной нами схеме перехватываются все ошибки и потому что программный код Python выполняется интерпретатором построчно (без предварительной компиляции). Когда черед доходит до выполнения команды с неизвестной функцией (неправильное название функции), естественно возникает ошибка (класса `NameError`), и эта ошибка перехватывается в блоке `except`. Поэтому мы даже не сможем значение для переменной `a` ввести.

С другой стороны, если речь идет не о неправильном названии функции (например, `float()`), а о некорректном синтаксисе команды (допустим, двойные кавычки в тексте не указаны), то это уже неправильный синтаксис команды, о чем появится сообщение сразу после запуска программы. Обычно такие ошибки автоматически выделяются в окне редактора кодов еще при наборе. Хотя, конечно, многое зависит от возможностей редактора.

Более утонченная обработка исключительных ситуаций подразумевает и более индивидуальный, так сказать, подход. Речь идет о том, чтобы обработка ошибок базировалась на типе или характере ошибки. Разумеется, это возможно. Причем описанная выше схема обработки исключительных ситуаций претерпевает минимальные изменения. Если более конкретно, то ситуация выглядит так: в конструкции `try-except` после блока `try` указывается несколько `except`-блоков, причем для каждого блока явно указывается тип ошибки, который обрабатывается в этом блоке. Ключевое слово, определяющее тип ошибки, указывается после ключевого слова `except` соответствующего блока.

🕮 На заметку

Вообще дела обстоят так. При возникновении ошибки генерируется исключение и создается объект, описывающий ошибку. Поэтому тип ошибки - это на самом деле *класс*, на основе которого создается объект исключения (более точное название, принятое в Python - *экземпляр исключения*). Другими словами, "типология" ошибок базируется на иерархии классов. Каждый класс соответствует какой-то ошибке. Работа с классами и объектами (экземплярами класса) обсуждается немного позже. Да и особенности использования классов и объектов здесь нас мало интересуют. Во всяком случае, не будет большой проблемой, если мы станем интерпретировать название класса ошибки просто как некоторое ключевое слово, определяющее тип, к которому относится ошибка.

В этом случае шаблон программного кода, содержащий обработку ошибок разного типа, имеет такой вид (жирным шрифтом выделены ключевые элементы):

```
try:
    # основной код
except Тип_ошибки_1:
    # вспомогательный код
except Тип_ошибки_2:
    # вспомогательный код
...
except Тип_ошибки_N:
    # вспомогательный код
```

Этот код выполняется следующим образом. Выполняются команды try-блока. Если возникла ошибка, то выполнение команд try-блока прекращается и начинается последовательный просмотр except-блоков на предмет совпадения типа ошибки, которая возникла, и типа ошибки, указанного после ключевого слова в except-блоке. Как только совпадение найдено, выполняются команды соответствующего except-блока, после чего управление переходит к команде после конструкции try-except.

Совпадение типов при поиске нужного except-блока для обработки ошибки не обязательно должно быть "буквальным". Дело вот в чем.

Мы уже знаем, что тип ошибки - это на самом деле класс, который описывает эту ошибку. Но у классов могут быть подклассы (производные классы). Это примерно так, как если бы у типа были подтипы. Или, другим словами, для некоторых категорий ошибок существует более детальная классификация по сравнению с остальными ошибками. Например, класс ZeroDivisionError, соответствующий ошибке деления на ноль, является подклассом класса ArithmeticError (арифметическая ошибка). А еще у класса ArithmeticError есть подклассы FloatingPointError (ошибка при операциях с плавающей точкой) и OverflowError (ошибка переполнения). Так вот, при обработке исключительных ситуаций перехватываются не только ошибки определенного (указанного в except-блоке) класса, но и ошибки подклассов этого класса. Например, если мы в except-блоке укажем класс ошибки ZeroDivisionError, то будут перехватываться ошибки типа ZeroDivisionError (деление на ноль). Но если в except-блоке указать класс ArithmeticError, то в этом блоке будут перехватываться и обрабатываться ошибки типа ZeroDivisionError, FloatingPointError и OverflowError.

Если при переборе except-блоков совпадение (по типу ошибки) не найдено, выполнение кода прекращается и появляется сообщение об ошибке. Правда, могут использоваться вложенные конструкции try-except. В этом случае необработанная ошибка может перехватываться внешним except-блоком (но такая возможность должна быть предусмотрена в программе).

Если при выполнении try-блока ошибок не было, коды в except-блоках не выполняются.

В листинге 2.12 приведен пример использования нескольких except-блоков с явным указанием типа ошибки. Фактически, там решается та же задача, что и в предыдущем случае, но только теперь использован несколько иной способ обработки исключительных ситуаций.

Листинг 2.12. Обработка ошибок разных типов

```
print("Решаем уравнение ax = b")
# Начало try-блока (основной код)
try:
    # Определяем первый параметр. Возможна ошибка
    # при преобразовании текста в число
    a=float(input("Введите a: "))
    # Определяем второй параметр. Возможна ошибка
    # при преобразовании текста в число
    b=float(input("Введите b: "))
    # Решение уравнения. Возможна ошибка
    # при делении на ноль
    x=b/a
    # Отображается значение для корня уравнения.
    # Команда выполняется, если до этого не возникли
    # ошибки
    print("Решение уравнения: x =",x)
# Начало except-блока (вспомогательный код)
except ValueError:
    # Команда выполняется если пользователь
    # ввел некорректное значение
    print("Нужно было ввести число!")
except ZeroDivisionError:
    # Команда выполняется при попытке
    # деления на ноль
    print("Внимание! На ноль делить нельзя!")
# Команда выполняется после блока try-except
print("Спасибо, работа программы завершена.")
```

По большому счету, изменилась, по сравнению с предыдущим примером, только та часть кода, что связана с except-блоками. Более точно, таких

блоков теперь два. После первого ключевого слова except указано название ValueError. К классу ValueError относятся ошибки, возникающие при рассогласовании типов (например, функции нужен аргумент числового типа, а передается текст). Во втором except-блоке обрабатываются ошибки класса ZeroDivisionError. К этому классу относятся ошибки, возникающие при попытке выполнить деление на ноль.

Как видим, для каждого except-блока предлагается свой программный код (в обоих случаях выводится сообщение с информацией о том, что же произошло). При выполнении кода в блоке try если произошла ошибка, тип этой ошибки сначала сопоставляется с классом ошибки ValueError в первом except-блоке. Если совпадение есть, то выполняется код этого блока. Если совпадения нет, проверяется совпадение типа ошибки с классом ошибки ZeroDivisionError во втором except-блоке. Если есть совпадение - выполняется код данного except-блока. Если же и здесь совпадения нет, выполнение программы завершается и появляется автоматически сгенерированное интерпретатором сообщение об ошибке.

В случае, когда все данные введены корректно, результат выполнения программы будет таким же, как и ранее. Ниже показано, как будет выглядеть результат выполнения программы, если при вводе для переменной b указано некорректное значение (жирным шрифтом выделен ввод пользователя):

Результат выполнения программы (из листинга 2.12)

```
Решаем уравнение ax = b
Введите a: 3
Введите b: number
Нужно было ввести число!
Спасибо, работа программы завершена.
```

Реакция программы на попытку деления на ноль будет такой:

Результат выполнения программы (из листинга 2.12)

```
Решаем уравнение ax = b
Введите a: 0
Введите b: 5
Внимание! На ноль делить нельзя!
Спасибо, работа программы завершена.
```

Стоит также обратить внимание на следующее обстоятельство: обрабатываются в данном случае только ошибки классов ValueError и ZeroDivisionError. Если бы теоретически возникла ошибка какого-то иного типа, то она бы перехвачена не была.

Что касается классов исключений, то кроме перечисленных выше, наибольший интерес с практической точки зрения могут представлять такие:

- Исключение `IndexError`: возникает, когда индекс (например, для элемента списка) указан неправильно (выходит за границы допустимого диапазона). Списки обсуждаются в одной из следующих глав.
- Исключение `KeyError`: возникает при неверно указанном ключе словаря. Словари обсуждаются в одной из следующих глав.
- Исключение `NameError`: возникает, если не удается найти локальное или глобальное имя (переменную) с некоторым названием.
- Исключение `SyntaxError`: связано с наличием синтаксических ошибок.
- Исключение `TypeError`: связано с несовместимостью типов, когда для обработки (например, в функции) требуется значение определенного типа, а передается значение другого типа.

Некоторые из этих исключений мы будем использовать (явно или неявно) в примерах. Более подробно классы исключений и методы работы с ними будут обсуждаться после того, как мы познакомимся с классами и объектами. В рассмотренном выше примере мы создавали для разных типов ошибок разные `except`-блоки. Иногда приходится несколько иную задачу: для нескольких типов ошибок создавать один `except`-блок. В этом случае после ключевого слова `except` в круглых скобках через запятую перечисляются те типы ошибок, для которых выполняется обработка в данном блоке.

В инструкции `try-except` помимо блоков `try` и `except` могут также использоваться блоки `else` и `finally`. Общий шаблон инструкции в этом случае такой (жирным шрифтом выделены ключевые элементы):

```
try:
    # основной код
except Тип_ошибки_1:
    # вспомогательный код
except Тип_ошибки_2:
    # вспомогательный код
...
except Тип_ошибки_N:
    # вспомогательный код
else:
    # код для случая, если ошибки не было
finally:
    # код, который выполняется всегда
```

Программный блок с ключевым словом `else` размещается после последнего `except`-блока и содержит программный код, который выполняется только в том случае, если при выполнении основного кода в `try`-блоке ошибок не было.

Программный код, размещенный в блоке `finally`, выполняется в любом случае, независимо от того, возникла ошибка при выполнении кода `try`-блока, или нет.

На заметку

Ошибки можно не только перехватывать, но и генерировать искусственно. Делается это с помощью инструкций `raise` или `assert`. Как это делается и зачем, мы обсудим в последней главе - после того, как познакомимся с классами и объектами. Там же будет описано, как можно создавать пользовательские классы исключений.

Резюме

Ну, скажи что-нибудь! Ты же у нас самый сообразительный.
из к/ф "Ирония судьбы или с легким паром"

1. Условный оператор позволяет выполнять различные блоки кода в зависимости от истинности или ложности некоторого условия.
2. В условном операторе после ключевого слова `if` указывается условие, которое проверяется при выполнении оператора. Если условие истинно, выполняется блок команд после условия. Если условие ложно, выполняется блок команд после ключевого слова `else`.
3. В условном операторе `else`-блок не является обязательным. Также в условном операторе могут использоваться `elif`-блоки, что позволяет проверять в условном операторе последовательно несколько условий и выполнять для каждого из них отдельный блок команд.
4. Оператор цикла `while` позволяет многократно выполнять предопределенный набор команд.
5. После ключевого слова `while` в операторе цикла указывается условие. Оператор выполняется, пока условие истинно. Условие проверяется в начале выполнения оператора, и затем каждый раз после выполнения группы команд оператора цикла.
6. Оператор цикла `for` удобен в том случае, если необходимо произвести перебор элементов некоторой последовательности. После ключевого слова `for` указывается переменная для выполнения перебора элементов последовательности, которая, в свою очередь, указывается после

ключевого слова `in`. Переменная последовательно принимает значения элементов последовательности, и для каждого такого значения выполняется блок команд оператора цикла.

7. В качестве последовательности, которая указывается в операторе цикла `for` можно, кроме прочего, указывать текст, списки или виртуальную числовую последовательность, созданную с помощью функции `range()`.

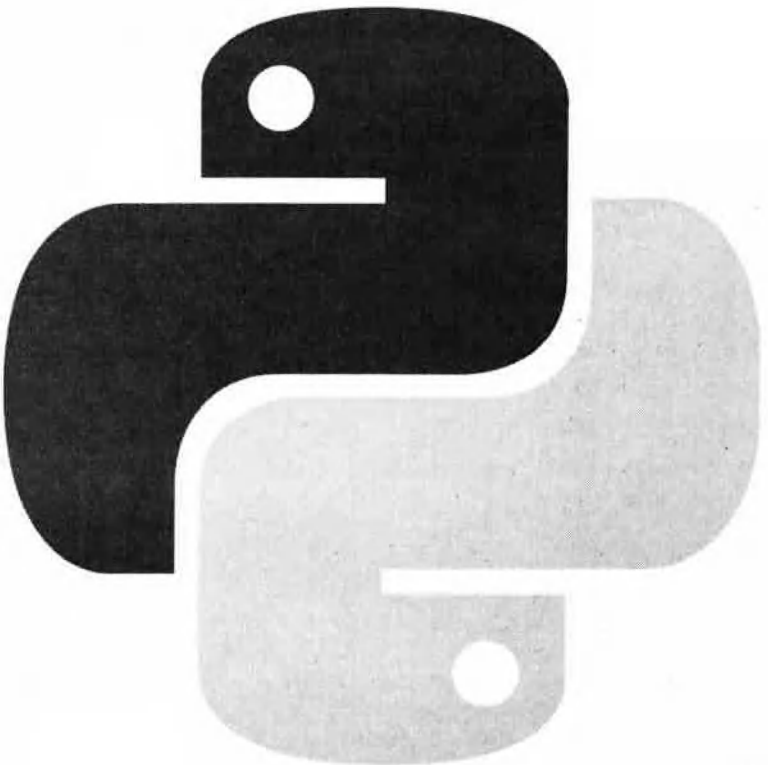
8. Инструкции `break` и `continue` используются в операторах цикла `while` и `for` соответственно для прекращения выполнения оператора цикла или для прекращения выполнения текущего цикла.

9. В операторах цикла (`while` и `for`) может использоваться `else`-блок, который выполняется по завершении работы оператора и при условии, что завершение работы оператора цикла не связано с выполнением инструкции `break`.

10. Механизм обработки исключительных ситуаций базируется на использовании инструкции `try-except`. Если при выполнении программного кода, помеченного ключевым словом `try`, возникает ошибка, она может быть перехвачена и обработана в одном из блоков, отмеченных инструкцией `except`. Для каждого `except`-блока после ключевого слова `except` можно указать тип ошибки (исключения), которая обрабатывается этим блоком.

Глава 3

Функции



Замечательная идея! Что ж она мне самому в голову не пришла?
из к/ф "Ирония судьбы или с легким паром"

Эта глава посвящена созданию функций пользователя в Python. С некоторыми функциями мы уже сталкивались ранее, но это были встроенные функции. Здесь мы узнаем, как создавать собственные функции. Также мы освоим основные приемы по использованию функций в программных кодах.

Когда мы говорим о функции, то имеем в виду фрагмент программного кода, который имеет название, и который по этому названию мы можем вызвать в любом (или практически любом) месте программы. Функции - очень полезный и эффективный инструмент программирования, поскольку позволяет значительно сократить объем программного кода, сделать его понятнее и эффективнее. Короче говоря, без функций писать эффективные программы практически нереально. Поэтому с функциями нужно быть на "ты", особенно если речь идет о программировании в Python.

Создание функции

- Мы не будем полагаться на случай. Мы пойдём простым логическим ходом.
- Пойдем вместе.
из к/ф "Ирония судьбы или с легким паром"

Перед тем, как функцию использовать, ее необходимо описать. Что подразумевает описание функции? Как минимум, это тот программный код, который следует выполнять при вызове функции, и, разумеется, название функции. Также функции при выполнении могут понадобиться некоторые параметры, которые передаются функции при вызове и без которых выполнение функции невозможно. Эти параметры мы будем называть *аргументами функции.*

На заметку

Нередко разделяют понятие *параметра* функции и *аргумента* функции. О параметрах обычно говорят при описании функции. Об аргументах идет речь, когда уже описанная функция вызывается в программном коде. Мы, чтобы не усложнять себе жизнь во всех этих случаях будем использовать термин *аргумент*.

Итак, для описания функции в Python необходимо:

- указать (задать) имя функции;
- описать аргументы функции;
- описать программный код функции.

Описание функции начинается с ключевого слова `def`. После него указывают название функции (это должен быть уникальный идентификатор, желательно со смысловой нагрузкой, состоящий из латинских букв, можно также цифр и символа подчеркивания - но с цифры имя функции начинаться не может). Затем перечисляются аргументы функции. Для аргументов просто указываются названия. Тип аргументов не указывается. Аргументы перечисляются после имени функции в круглых скобках. Если аргументов несколько, они разделяются запятыми. Даже если аргументов у функции нет (такое тоже бывает), круглые скобки все равно указываются. Заканчивается вся эта конструкция двоеточием, и далее с новой строки следует блок команд, которые собственно и определяют программный код функции. Весь шаблон объявления функции выглядит следующим образом (жирным шрифтом выделены ключевые элементы):

```
def имя_функции(аргументы):
    команды
```

То есть все достаточно просто и прозрачно.

На заметку

Для функции не указывается тип результата (хотя функция, разумеется, может возвращать результат). Это может стать сюрпризом для тех, кто изучал и знаком с другими языками программирования. Но это сюрприз только на первый взгляд. Если вспомнить, что в языке Python для переменных тип не указывается и определяется по значению, на которое ссылается переменная, то все выглядит вполне логичным: и отсутствие идентификатора типа для результата функции, и аргументы, для которых тип тоже не указывается.

Функция может возвращать результат, а может не возвращать результат (в последнем случае имело бы смысл говорить о *процедуре* - но в Python все называется одним словом *функция*). Если функция не возвращает результат, то ее можно рассматривать как некоторую последовательность ко-

манд, которые выполняются в том месте и в то время, что и вызов функции. Если функция возвращает результат (а это может быть значение практически любого типа), то инструкцию вызова функции можно использовать в выражениях так, как если бы это была некоторая переменная. В теле функции чтобы определить то значение, которое является результатом функции, обычно используют инструкцию `return`. Выполнение этой инструкции, во-первых, приводит к завершению выполнения программного кода функции, а во-вторых то значение, которое указано после инструкции `return`, возвращается функцией в качестве результата.

На заметку

Вызывается функция просто - указывается имя функции и аргументы, которые ей передаются. Ну и, разумеется, вызывать функцию можно только после того, как функция объявлена, но никак не раньше.

Далее мы рассмотрим простой пример, в котором объявляется несколько простых функций. После объявления эти функции вызываются в программном коде для выполнения несложных действий. Соответствующий программный код представлен в листинге 3.1.

Листинг 3.1. Объявление функций

```
# Функция без аргументов
def your_name():
    # Отображается сообщение
    print("Добрый день!")
    # Запоминается введенный пользователем текст
    name=input("Как Вас зовут? ")
    # Результат функции
    return name
# Функция с одним аргументом
def say_hello(txt):
    # Отображается сообщение
    print("Здравствуйте,",txt+"!")
# Вызываем функцию и результат записываем в переменную
my_name=your_name()
# Вызываем функцию с аргументом
say_hello(my_name)
```

В результате выполнения этого программного кода получаем следующий результат (жирным шрифтом выделено введенное пользователем значение):

Результат выполнения программы (из листинга 3.1)

```
Добрый день!
Как Вас зовут? Алексей Васильев
Здравствуйте, Алексей Васильев!
```

Код достаточно простой, но мы его все же прокомментируем. В программе объявляется две функции. Функция `your_name()` не имеет аргументов. При выполнении этой функции сначала командой `print("Добрый день!")` отображается приветствие. Затем пользователю предлагается ввести свое имя. Введенное пользователем текстовое значение запоминается в переменной `name`. Вся команда выглядит как `name=input("Как Вас зовут? ")`.

После этого с помощью инструкции `return name` значение переменной `name` возвращается в качестве результата функции `your_name()`. Таким образом, у этой функции нет аргументов, но зато она возвращает результат. И ее результат - это то, что вводит пользователь (предполагается, что имя пользователя).

Еще одна функция `say_hello()` нужна для отображения приветствия. У функции один аргумент (обозначен как `txt`). Текст сообщения, которое отображается при вызове функции, формируется с учетом значения аргумента `txt`, переданного функции. Результат функция не возвращает. В теле функции всего одна команда `print("Здравствуйте,",txt+"!")`, которой в консольное окно выводится сообщение.

На заметку

При вычислении выражения `txt+"!"` мы неявно предполагаем, что аргумент `txt` ссылается на текстовое значение. Если такой уверенности нет, то лучше перестраховаться и воспользоваться выражением `str(txt)+"!"`, в котором выполняется явное приведение аргумента `txt` к текстовому типу.

Описанные функции используем следующим образом. Сначала командой `my_name=your_name()` вызываем функцию `your_name()` и результат вызова записываем в переменную `my_name`. Затем командой `say_hello(my_name)` вызываем функцию `say_hello()`, передав ей аргументом переменную `my_name`. Результат этих действий таков, как показано выше.

Функции для математических вычислений

Это великая победа дедуктивного метода.
из к/ф "Гостья из будущего"

Очень удобно создавать функции для выполнения математических расчетов. Особенно естественным такой подход представляется при реализации в программе математических функций. Причина в том, что концепция математической функции достаточно близка концепции функции в программировании. Многие наиболее популярные и часто используемые математические функции представлены в математическом модуле math. Но часто приходится описывать математические функции самостоятельно. Как иллюстрацию рассмотрим программный код в листинге 3.2, в котором реализована математическая функция для вычисления экспоненты exp (x).

🕮 На заметку

При вычислениях мы используем следующее выражение (основанное на ряде Тейлора для соответствующей функции): $exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots$. Восклицательный знак означает вычисление факториала: по определению $n!$ означает произведение натуральных чисел от 1 до n, то есть $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$

Листинг 3.2. Математические функции

```
# Функция для вычисления экспоненты
def my_exp(x,n):
    s=0 # Начальное значение суммы ряда
    q=1 # Начальное значение добавки
    # Оператор цикла для вычисления ряда
    for k in range(n+1):
        s+=q        # Добавка к сумме
        q*=x/(k+1) # Новая добавка
    # Результат функции
    return s
# Проверяем результат вызова функции
x=1 # Аргумент для экспоненты
# Оператор цикла для многократного
# вызова функции вычисления экспоненты
for n in range(11):
    # Отображаем результат вызова
    # функции экспоненты
    print("n =",n,"->",my_exp(x,n))
```

Функция для вычисления экспоненты называется `my_exp()` и у нее два аргумента: через `x` обозначен непосредственно аргумент для вычисления экспоненты, а второй аргумент `n` определяет количество слагаемых в сумме для экспоненты.

На заметку

Точное выражение для экспоненты $\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$ - это бесконечный ряд. При вычислении значения для экспоненты на практике используется приближенное выражение $\exp(x) \approx \sum_{k=0}^{n} \frac{x^k}{k!}$, в котором бесконечный ряд заменен на конечную сумму. Чем больше слагаемых в сумме, тем точнее выражение для экспоненты. В рассматриваемом примере аргументами функции, которую мы описываем для вычисления экспоненты, являются параметры x и n.

В теле функции мы используем несколько переменных. В переменную `s` мы будем записывать сумму ряда для экспоненты. Начальное значение этой переменной равно нулю. Еще одна переменная `q` с начальным единичным значением нужна нам для того, чтобы записывать в эту переменную значение добавки, которая на каждой новой итерации прибавляется к сумме ряда.

Сумма ряда вычисляется с помощью оператора цикла. Индексная переменная `k` пробегает значения от `0` до `n` включительно (выражением `range(n+1)` возвращается виртуальная последовательность в диапазоне от `0` до `n` включительно). В операторе цикла всего две команды: командой `s+=q` сумма ряда `s` увеличивается на значение добавки `q`, а командой `q*=x/(k+1)` вычисляется значение добавки к ряду для следующего цикла.

На заметку

При вычислении суммы $\exp(x) \approx \sum_{k=0}^{n} \frac{x^k}{k!}$ на k-й итерации добавка к сумме ряда равняется $\frac{x^k}{k!}$, а для следующей итерации эта добавка должна быть $\frac{x^{k+1}}{(k+1)!}$. Обозначим $q_k = \frac{x^k}{k!}$ и $q_{k+1} = \frac{x^{k+1}}{(k+1)!}$. Несложно заметить, что $\frac{q_{k+1}}{q_k} = \frac{x}{k+1}$. Поэтому чтобы вычислить добавку для следующего цикла по значению добавки на текущем цикле (переменная `q`) нужно текущее значение добавки умножить на величину $\frac{x}{k+1}$.

После завершения работы оператора цикла сумма ряда для экспоненты вычислена, и переменная `s` ссылается на это значение. Поэтому командой `return s` значение переменной `s` возвращается как результат функции.

В программе мы проверяем результат вызова функции вычисления экспоненты для аргумента `x=1` но при различных значениях второго аргумента функции. Для перебора значений (в диапазоне от `0` до `10`) второго аргумента функции `my_exp()` запускаем оператор цикла. В каждом цикле выпол-

няется команда `print("n =",n,"->",my_exp(x,n))`. Как следствие в окне вывода отображается последовательность значений второго аргумента и значения функции, вычисленное по соответствующему количеству слагаемых в сумме ряда. Результат выполнения программы имеет следующий вид:

Результат выполнения программы (из листинга 3.2)

```
n = 1 -> 1
n = 2 -> 2.0
n = 3 -> 2.5
n = 4 -> 2.6666666666666665
n = 5 -> 2.708333333333333
n = 6 -> 2.7166666666666663
n = 7 -> 2.7180555555555554
n = 8 -> 2.7182539682539684
n = 9 -> 2.71827876984127
n = 10 -> 2.7182815255731922
```

Как и следовало ожидать, с увеличением количества слагаемых точность вычислений возрастает (напомним, "точное" значение равняется $e \equiv \exp(1) \approx 2.718281828459045$).

На заметку

Для вычисления экспоненты в модуле `math` есть функция `exp()`.

Значения аргументов по умолчанию

Лучше бы я упал вместо тебя.
из к/ф "Бриллиантовая рука"

Для аргументов функций нередко задают значения по умолчанию. То есть для аргументов функции допускается указать значение, которое будет использовано, если аргумент функции явно не указан. Можно и иначе: если при вызове функции аргумент не указан, то вместо него используется значение по умолчанию.

Чтобы задать значение аргумента по умолчанию, в описании функции после имени этого аргумента через знак равенства указывается значение по умолчанию. У функции, как мы знаем, возможно наличие нескольких аргументов. Часть этих аргументов может иметь значения по умолчанию, а остальные - нет. Важно помнить, что в описании функции при перечисле-

нии аргументов сначала указываются аргументы без значений по умолчанию, а аргументы со значениями по умолчанию размещаются в конце.

На заметку

Данный способ передачи функции аргументов (сначала без значений по умолчанию, а затем со значениями по умолчанию) - вынужденная мера. В противном случае при вызове функции, если список переданных аргументов неполный, проблематично было бы определить, какие именно аргументу переданы функции.

Примеры функций со значениями аргументов по умолчанию приведены в программном коде листинге 3.3.

Листинг 3.3. Значения аргументов по умолчанию

```
# 1-я функция с одним аргументом.
# У аргумента есть значение по умолчанию
def print_text(txt="Значение аргумента по умолчанию."):
    print(txt)
# 2-я функция с двумя аргументами.
# У второго аргумента есть значение по умолчанию
def show_args(a,b="Второй аргумент не указан."):
    print(a,b)
# 3-я функция с двумя аргументами.
# У аргументов есть значения по умолчанию
def my_func(x="1-й аргумент x.",y="2-й аргумент y.",):
    print(x,y)
# Проверяем работу 1-й функции.
# Функции передан один аргумент
print_text("Аргумент указан явно.")
# Функции аргументы не передаются
print_text()
# Проверяем работу 2-й функции.
# Функции переданы два аргумента
show_args("Первый аргумент.","Второй аргумент.")
# Функции передан один аргумент
show_args("Первый аргумент.")
# Проверяем работу 3-й функции.
# Функции аргументы не передаются
my_func()
# Функции передан один аргумент
my_func("Один из аргументов.")
# Функции передан один аргумент.
# Переданный аргумент идентифицирован явно
my_func(y="Один из аргументов.")
```

В результате выполнения программного кода получаем следующий результат:

Результат выполнения программы (из листинга 3.3)

```
Аргумент указан явно.
Значение аргумента по умолчанию.
Первый аргумент. Второй аргумент.
Первый аргумент. Второй аргумент не указан.
1-й аргумент x. 2-й аргумент y.
Один из аргументов. 2-й аргумент y.
1-й аргумент x. Один из аргументов.
```

Мы объявляем три функции. Все три функции действуют по одной схеме: значения аргументов выводятся на экран. Мы подразумеваем, что все три функции оперируют с текстовыми аргументами. Некоторые из этих аргументов имеют значения по умолчанию. Так, у функции `print_text()` один аргумент, и у этого аргумента есть значение по умолчанию. Мы вызываем эту функцию без аргумента (в этом случае используется значение аргумента по умолчанию), а также вызываем эту функцию с одним аргументом (в этом случае используется переданное аргументом значение).

У функции `show_args()` два аргумента, и у второго аргумента есть значение по умолчанию. Поэтому если вызвать функцию с одним аргументом, то для второго аргумента используется значение по умолчанию. Если передаются два аргумента, то для обоих аргументов используются те значения, что переданы.

У функции `my_func()`, как и в предыдущем случае, два аргумента, но теперь у обоих аргументов есть значения по умолчанию. Если мы вызовем эту функцию с двумя аргументами, то интриги не будет - те значения, что переданы аргументами, будут использованы при выполнении кода функции. Если функции при вызове передан один аргумент, то возникает вопрос: какой это аргумент - первый или второй? Ведь значение по умолчанию есть и у первого, и у второго аргумента. Поэтому теоретически переданное функции значение может быть как первым, так и вторым аргументом.

По умолчанию считается, что функции передается первый аргумент, а для второго используется значение по умолчанию. Но у нас есть еще одна возможность: мы можем явно указать, для какого аргумента явно указано значение. В этом случае при вызове функции указывается не только значение аргумента, но и его *имя* (то, которое указывалось при описании функции). Используется формат `имя_аргумента=значение`.

Примером может быть команда `my_func(y="Один из аргументов.")`. В этом случае для первого аргумента используется значение по умолчанию, а переданное функции значение - это значение второго аргумента.

Таким образом, передача аргументов при вызове функции возможна:

- без указания имени аргументов, при этом значения аргументов передаются строго в той последовательности, как эти аргументы были описаны в функции (такой способ передачи аргументов называют *позиционным*);
- с явным указанием имени аргументов, при этом порядок перечисления аргументов не имеет значения (такой способ называется передачей аргументов *по ключу* или *по имени*, а сами аргументы иногда называют *именованными*).

Если при вызове функции часть аргументов передается по ключу, а часть - позиционным способом, то в команде вызова сначала указываются позиционные аргументы, а затем те, что передаются по ключу. Стоит также особо обратить внимание, что способ описания аргументов (при определении функции) не зависит того, как предполагается передавать аргументы при вызове функции (по ключу или позиционно).

На заметку

Если у аргументов некоторой функции есть значения по умолчанию, то эта функция может вызываться с разным количеством аргументов (подробности зависят от специфики описания функции). В языке Python можно объявлять функции с переменным количеством аргументов. Этот вопрос более детально мы обсудим позже. Здесь же сделаем общее вводное замечание.

Предположим, нам нужно описать функцию такую, чтобы ее можно было вызывать с разным количеством аргументов. Здесь важно, что количество аргументов не ограничено - мы наперед не знаем, сколько в принципе будет передаваться аргументов функции. В этом случае мы описываем функцию с одним аргументом, но перед этим аргументом ставим звездочку *. Такой аргумент отождествляется со списком, элементы которого формируются реальными аргументами, что переданы функции при вызове. Другими словами, наш "звездный" аргумент в теле функции обрабатывается как список. Но при вызове функции аргументы передаются, как обычно. Например, функция может быть описана так:

```
def get_sum(*nums):
    s=0
    for a in nums:
        s+=a
    return s
```

Данная функция в качестве результата будет возвращать сумму чисел, переданных аргументами функции. Скажем, значением выражения `get_sum(1,3,5,2)` будет число 11, а значение выражения `get_sum(-2,4)` - число 2. Списки обсуждаются в следующей главе. Функции с переменным числом аргументов описываются в последней главе.

Функция как аргумент

Я вся такая внезапная, такая противоречивая вся...
из к/ф "Покровские ворота"

Чтобы понять следующий прием, важно уточнить некоторые особенности, касающиеся объявления функций в Python. А именно, при объявлении функции на самом деле создается объект типа `function`. Ссылка на этот объект записывается в переменную, которая указана в описании функции после инструкции `def`. Другими словами, имя функции (без круглых скобок) можно использовать как переменную (которая ссылается на функцию). Вопрос только в том, что с этой переменной можно сделать. Вариантов здесь довольно много, но нас интересует в первую очередь возможность присвоить имя функции другой переменной, а затем вызвать эту функцию через данную переменную так, как если бы это было имя функции. При присваивании переменной в качестве значения имени функции эта переменная становится ссылкой на функцию (наравне с именем функции). Небольшой пример, иллюстрирующий эту возможность, представлен в листинге 3.4.

Листинг 3.4. Ссылка на функцию

```
# Исходная функция
def my_func(txt):
    print("Функция my_func:",txt)
# Переменной присваивается имя функции
new_func=my_func
# Вызываем функцию через переменную
new_func("вызов через new_func.")
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 3.4)

```
Функция my_func: вызов через new_func.
```

Идея здесь реализована очень простая. Сначала мы описываем функцию `my_func()` (у функции один аргумент), а затем командой `new_func=my_func` переменной `new_func` в качестве значения присваи-

вается имя функции `my_func`. В результате и идентификатор (имя функции) `my_func`, и переменная `new_func` ссылаются на один и тот же объект - объект функции. Это означает буквально следующее: о переменной `new_func` мы можем думать как об имени функции, причем функции той же самой, что и функция `my_func()`. Поэтому кода выполняется команда `new_func("вызов через new_func.")`, это все равно, как если бы мы вызвали функцию `my_func()` с тем же точно аргументом.

Описанное свойство функций (возможность присваивать имя функции переменной) имеет ряд полезных применений, в том числе перед нами открывается возможность передавать имя функции аргументом другой функции. Скажем откровенно, возможность нетривиальная и позволяет легко (и где-то даже красиво) решать сложные задачи. Классической иллюстрацией могут быть некоторые математические задачи, в которых функциональная зависимость играет роль внешнего фактора: например, при вычислении интегралов, решении дифференциальных или алгебраических уравнений.

Далее мы рассмотрим несколько иллюстраций к тому, как одна функция передается аргументом другой функции. Начнем с простого примера, в котором описывается функция для решения алгебраических уравнений методом последовательных приближений.

На заметку

Речь идет о решении уравнения вида $x = f(x)$ относительно переменной x, при условии, что функция $f(x)$ известна (задана). Суть метода последовательных приближений состоит в том, что задано начальное приближение x_0 для корня уравнения, на основании которого вычисляется первое приближение для корня $x_1 = f(x_0)$. На основании первого приближения x_1 вычисляют второе приближение $x_2 = f(x_1)$, и так далее. Общая рекуррентная формула для вычисления x_{n+1} приближения на основе приближения x_n имеет вид $x_{n+1} = f(x_n)$. Именно эту схему мы собираемся реализовать в программе.

Понятно, что далеко не любое уравнение можно решить подобным методом. Существуют определенные критерии применимости метода последовательных приближений. В частности, функция $f(x)$ должна быть такой, чтобы на всей области поиска корня выполнялось соотношение $f'(x)| < 1$, то есть модуль производной функции должен быть меньше единицы.

Задача состоит в том, чтобы не просто решить какое-то конкретное уравнение, а написать функцию, которая бы позволяла решать указанным методом различные уравнения (разумеется, при условии, что метод вообще для них применим). Программный код с решением этой задачи представлен в листинге 3.5.

Листинг 3.5. Метод последовательных приближений

```
# Описание функции для решения уравнения
def solve_eqn(f,x0,n):
    # Начальное приближение для корня
    x=x0
    # Оператор цикла для вычисления
    # приближений для решения
    for k in range(1,n+1):
        x=f(x) # Итерационная формула
    # Результат функции
    return x
# Функция, определяющая 1-е уравнение
def eqn_1(x):
    # Значение функции
    return (x**2+5)/6
# Функция, определяющая 2-е уравнение
def eqn_2(x):
    # Значение функции
    return (6*x-5)**0.5
# Решаем 1-е уравнение
x=solve_eqn(eqn_1,0,10)
# Отображаем результат
print("1-е уравнение: x =",x)
# Решаем 2-е уравнение
x=solve_eqn(eqn_2,4,10)
# Отображаем результат
print("2-е уравнение: x =",x)
```

В программе мы описываем функцию `solve_eqn()` с тремя аргументами. Наши предположения относительно аргументов такие:

- первый аргумент `f` является ссылкой на функцию, определяющую решаемое уравнение (имеется в виду функция $f(x)$ в уравнении $x = f(x)$);
- второй аргумент `x0` определяет начальное приближение для корня уравнения;
- третий аргумент `n` функции - количество итераций, по которым вычисляется корень.

В теле функции `solve_eqn()` командой `x=x0` переменной `x` в качестве начального присваивается нулевое приближение для корня уравнения. После этого запускается оператор цикла, в котором осуществляется `n` итераций (циклов). За каждый цикл выполняется команда `x=f(x)`, в которой мы ис-

пользуем вызов функции, переданной через идентификатор f аргументом функции solve_eqn(). Каждое выполнение команды x=f(x) приводит к вычислению нового (следующего) приближения для корня уравнения. По завершении оператора цикла переменная x командой return x возвращается как результат функции solve_eqn().

Также мы определяем две функции от одного аргумента, которые задают уравнения для решения.

На заметку

Мы решаем квадратное уравнение $x^2 - 6x + 5 = 0$, которое имеет два корня: $x = 1$ и $x = 5$. Для применения метода последовательных приближений это уравнение нужно переписать. Причем для писка разных корней нужно использовать разные представления. Связано это с необходимым условием сходимости метода. Так, для поиска корня $x = 1$ уравнение представим в виде $x = \frac{x^2+5}{6}$. В этом случае уравнение задается функцией $x = \frac{x^2+5}{6}$. Производная от этой функции $f'(x) = \frac{x}{3}$ в точке предполагаемого корня $x = 1$ по модулю меньше единицы. Функция $x = \frac{x^2+5}{6}$ в программном коде реализуется функцией eqn_1(). Имя этой функции будем указывать первым аргументом функции solve_eqn(). Начальное приближение в этом случае должно попадать в интервал $-3 < x < 3$. Второй корень уравнения $x = 5$ не попадает в этот интервал, поэтому для его вычисления необходимо изменить способ записи уравнения. Теперь представим уравнение в виде $x = \sqrt{6x-5}$. Данное уравнение задается функцией $f(x) = \sqrt{6x-5}$. Производная от этой функции $f'(x) = \frac{3}{\sqrt{6x-5}}$ по модулю меньше единицы при $x > \frac{7}{3} \approx 2.66667$. Зависимость $f(x) = \sqrt{6x-5}$ в программе реализована через функцию eqn_2(). Имя этой функции передадим первым аргументом функции solve_eqn(). Вторым аргументом нужно указать числовое значение, большее 7/3.

Хотя в данном случае речь фактически идет об одном уравнении, записанном в разном виде, мы, чтобы не путаться, будем говорить о двух уравнениях.

Функция eqn_1(), определяющая первое уравнение, содержит всего одну команду return (x**2+5)/6, которой на основе значения аргумента x вычисляется результат функции. В данном случае определяется функциональная зависимость $f(x) = \frac{x^2+5}{6}$ и решается, соответственно, уравнение $x = \frac{x^2+5}{6}$ (речь идет о корне $x = 1$).

Еще одна функция, которая называется eqn_2() также предназначена для определения уравнения для решения. Значение функции возвращается командой return (6*x-5)**0.5, где x - это аргумент функции. Та-

ким образом, рассматриваем функцию уравнения $f(x) = \sqrt{6x-5}$, уравнение $x = \sqrt{6x-5}$ (и ищем решение $x = 5$).

Для решения первого уравнения мы используем команду `x=solve_eqn(eqn_1,0,10)`. Переменной `x` в качестве значения присваивается результат вызова функции `solve_eqn()`, первым аргументом которой передано имя `eqn_1` функции, определяющей решаемое уравнение, второй аргумент - начальное (в данном случае нулевое) значение для корня уравнения, а третий аргумент (значение `10`) определяет количество итераций, по которым вычисляется приближение для корня уравнения (в принципе, чем больше итераций - тем выше точность решения).

Командой `print("1-е уравнение: x =",x)` найденное решение для корня уравнения отображается в консольном окне.

По той же схеме ищется решение для второго уравнения, только теперь первым аргументом функции `solve_eqn()` передается идентификатор `eqn_2`, а начальное значение для корня (второй аргумент) равно `4`. Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 3.5)

```
1-е уравнение: x = 0.9999925289581152
2-е уравнение: x = 4.992839487820055
```

Как несложно заметить, найденные нами приближенные решения достаточно близки к точным значениям.

Далее рассмотрим задачу о решении дифференциального уравнения первого порядка $y'(x) = f(x, y(x))$ с начальным условием $y(x_0) = y_0$ (*задача Коши*). Здесь через x обозначена независимая переменная, $y(x)$ - неизвестная функция от независимой переменной, $y'(x) \equiv \frac{dy}{dx}$ - производная от функции $y(x)$ по аргументу x, а $f(x, y)$ - известная (заданная) функция двух аргументов (функцию $f(x, y)$ будем называть *функцией уравнения*). Нам необходимо найти такую функцию $y(x)$, чтобы дифференциальное уравнение $y'(x) = f(x, y(x))$ превратилось в тождество, и, кроме этого, в точке $x = x_0$ функция $y(x)$ принимала значение y_0 (числовые параметры x_0 и y_0 считаем заданными).

Для решения уравнения в числовом виде используется наиболее простая схема Эйлера.

На заметку

Если вкратце, то суть схемы Эйлера состоит в следующем. Интервал значений аргумента от x_0 (точка, в которой задано начальное условие) до x (точка, в которой нужно вычислить значение функции $y(x)$) разбивается на n (достаточно большое число) одинаковых отрезков длиной $\Delta x = \frac{x - x_0}{n}$.

Значения функции $y(x)$ последовательно вычисляются в узловых точках $x_k = x_0 + \Delta x \cdot k$, где индекс $k = 0,1,2, \ldots, n$, причем по договоренности $x_n \equiv x$. В узловой точке x_0 значение функции $y(x_0) = y_0$ определяется из начального условия. Для всех остальных узловых точек x_k вычисление значения функции $y_k \equiv y(x_k)$ осуществляется на основе рекуррентного соотношения $y_k = y_{k-1} + \Delta x \cdot f(x_{k-1}, y_{k-1})$.

Таким образом, если мы знаем значение функции y_0 в точке x_0 (а мы его знаем из начального условия), то можем вычислить значение функции y_1 в точке x_1 по формуле $y_1 = y_0 + \Delta x \cdot f(x_0, y_0)$.

Зная y_1, вычисляем $y_2 = y_1 + \Delta x \cdot f(x_1, y_1)$, и так далее до $y_n = y_{n-1} + \Delta x \cdot f(x_{n-1}, y_{n-1})$. А $y_n \equiv y(x_n)$ - это и есть $y(x)$.

Для реализации метода Эйлера по вычислению решения дифференциального уравнения в точке описываем специальную функцию (называется `solve_deqn()`). У этой функции четыре аргумента:

- название функции, определяющей дифференциальное уравнение (аргумент с названием `f`);
- узловая точка, в которой задается начальное условие (аргумент с названием `x0`);
- значение искомой функции в точке начального условия (аргумент с названием `y0`);
- значение точки, для которой необходимо вычислить значение функции - решения дифференциального уравнения (аргумент с названием `x`).

В качестве результата функция будет возвращать значение решения дифференциального уравнения (для заданной точки). Соответствующий программный код приведен в листинге 3.6.

Листинг 3.6. Решение дифференциального уравнения

```
# Импорт математического модуля
import math
# Функция для решения дифференциального уравнения
def solve_deqn(f,x0,y0,x):
    # Количество отрезков, на которые делится интервал
    # поиска решения уравнения
    n=1000
    # Расстояние между соседними узловыми точками
    dx=(x-x0)/n
    # Начальная точка
    x=x0
    # Начальное значение функции
    y=y0
    # Оператор цикла для вычисления решения
    for k in range(1,n+1):
        # Значение функции в узловой точке
        y=y+dx*f(x,y)
        # Следующая узловая точка
        x=x+dx
    # Результат функции
    return y
# Функция, определяющая дифференциальное уравнение
def diff_eqn(x,y):
    # Результат функции
    return 2*x-y
# Функция точного решения уравнения
def y(x):
    # Результат функции
    return 2*(x-1)+5*math.exp(-x)
# Шаг приращения по аргументу
h=0.5
# Вычисление результата для нескольких
# значений аргумента
for k in range(0,6):
    # Значение аргумента
    x=k*h
    print("Числовое решение:")
    # Числовое решение
    print("x =",x,"-> y(x) =",solve_deqn(diff_eqn,0,3,x))
    print("Точное решение:")
    # Точное решение
    print("x =",x,"-> y(x) =",y(x))
```

При выполнении программного кода получаем такой результат:

Результат выполнения программы (из листинга 3.6)

```
Числовое решение:
x = 0.0 -> y(x) = 3.0
Точное решение:
x = 0.0 -> y(x) = 3.0
Числовое решение:
x = 0.5 -> y(x) = 2.0322741142003067
Точное решение:
x = 0.5 -> y(x) = 2.032653298563167
Числовое решение:
x = 1.0 -> y(x) = 1.8384771238548225
Точное решение:
x = 1.0 -> y(x) = 1.8393972058572117
Числовое решение:
x = 1.5 -> y(x) = 2.1143951442170557
Точное решение:
x = 1.5 -> y(x) = 2.115650800742149
Числовое решение:
x = 2.0 -> y(x) = 2.6753226122334164
Точное решение:
x = 2.0 -> y(x) = 2.6766764161830636
Числовое решение:
x = 2.5 -> y(x) = 3.4091422819998414
Точное решение:
x = 2.5 -> y(x) = 3.410424993119494
```

Хотя программного кода довольно много, программа на самом деле простая. Проанализируем основные ее места и наиболее важные моменты.

Относительно новая для нас инструкция `import math` нужна для импорта математического модуля `math`, а он, в свою очередь, понадобится нам для того, чтобы при проверке найденного числового решения (сравнении его с точным, аналитическим решением), можно было использовать функцию для вычисления значения экспоненты `exp()`.

В теле функции `solve_deqn()`, используемой для решения дифференциального уравнения, командой `n=1000` задается количество отрезков, на которые разбивается интервал поиска решения (имеется в виду интервал значений аргумента функции-решения дифференциального уравнения от `x0` до `x`). Тогда расстояние между соседними узловыми точками будет составлять величину `dx=(x-x0)/n` (длина интервала, деленная на количество отрезков, на которые разбивается интервал). В переменную `x` командой `x=x0` записывается значение точки, в которой задается начальное условие, а командой `y=y0` переменной `y` присваивается значение искомой функ-

ции в точке начального условия. Таким образом, переменные x и y получают свои начальные значения. После этого запускается итерационный процесс, основу которого составляет оператор цикла. Оператор цикла рассчитан на выполнение n итераций, причем за каждую итерацию выполняется две команды. Сначала командой `y=y+dx*f(x,y)` вычисляется значение функции-решения уравнения в следующей (по сравнению с текущей) узловой точке. Затем командой `x=x+dx` вычисляется сама узловая точка. После выполнения оператора цикла значение переменной y возвращается в качестве результата функции `solve_deqn()`.

Также в программе мы определяем функцию, которая задает решаемое дифференциальное уравнение. Речь идет о функции `diff_eqn()`, у которой два аргумента (x и y), а в качестве результат функцией возвращается выражение `2*x-y`.

На заметку

Исходя из определения функции `diff_eqn()` несложно сделать вывод, что речь идет о функции уравнения $f(x,y) = 2x - y$. Следовательно, решается дифференциальное уравнение $y'(x) = 2x - y(x)$. У этого уравнения есть аналитическое (точное) решение $y(x) = 2 \cdot (x - 1).+ C_1 \cdot \exp(-x)$, где постоянная C_1 определяется из начального условия. Например, если начальное условие записывается как $y(0) = 3$ (а в программе при проверке результата мы используем именно такое начальное условие), то решение задачи Коши будет иметь вид $y(x) = 2 \cdot (x - 1) + 5 \cdot \exp(-x)$.

Чтобы сравнить результат, полученный по схеме Эйлера, с точным решением $y(x) = 2 \cdot (x - 1) + 5 \cdot \exp(-x)$ уравнения, в программном коде мы объявляем функцию `y()` с одним аргументом x, которая соответствует аналитическому решению задачи Коши для уравнения $y'(x) = 2x - y(x)$ с начальным условием $y(0) = 3$.

На заметку

В теле функции `y()` для вычисления значения экспоненты вызывается функция `exp()` из модуля `math`, который мы в самом начале программы подключили с помощью инструкции `import`. Аргументом функции `exp()` передается показатель экспоненты. При вызове функции `exp()` явно указывается модуль `math`, поэтому математическому выражению `exp(-x)` (или то же самое, что e^{-x}) в программном коде соответствует инструкция `math.exp(-x)`.

Проверка созданных функций выполняется так: для нескольких значений аргумента x функции-решения уравнения мы находим числовое решение и сравниваем его с точным решением. Для этого запускаем оператор цикла, в котором индексная переменная k пробегает значения от 0 до 5. Пере-

менная x принимает значения k*h (h=0.5 - шаг приращения для аргумента x), приближенное (числовое) решение вычисляется инструкцией solve_deqn(diff_eqn,0,3,x), а для получения точного решения используем выражение y(x).

Рекурсия

Никогда не думай, что ты иная, чем могла бы быть иначе, чем будучи иной в тех случаях, когда иначе нельзя не быть.
Л. Кэрролл "Алиса в стране чудес"

При описании функций иногда удобно использовать *рекурсию*. Рекурсия подразумевает, что в программном коде функции используется вызов этой же самой функции (но обычно с другим аргументом). Лучше всего сказанное проиллюстрировать на примере. В листинге 3.7 приведен программный код функции, которой по порядковому номеру вычисляется число из последовательности Фибоначчи. При описании функции использована рекурсия.

На заметку

Числа Фибоначчи - последовательность чисел, в которой первое и второе число равны единице, а каждое следующее определяется как сумма двух предыдущих. Таким образом, речь идет о числах 1, 1, 2, 3, 5, 8, 13, 21 и так далее.

Листинг 3.7. Числа Фибоначчи

```
# Функция для вычислений чисел Фибоначчи.
# При описании функции использована рекурсия
def Fib(n):
    # Первое и второе число
    # в последовательности равны 1
    if n==1 or n==2:
        return 1
    # Числа в последовательности
    # равно сумме двух предыдущих
    else:
        return Fib(n-1)+Fib(n-2)
# Проверяем работу функции
print("Числа Фибоначчи:")
# Вычисляем 15 первых чисел Фибоначчи
for i in range(1,16):
    # Числа печатаются в одной строке через пробел
    print(Fib(i),end=" ")
```

В результате выполнения программы в ряд через пробел отображается 15 чисел из последовательности Фибоначчи:

Результат выполнения программы (из листинга 3.7)

```
Числа Фибоначчи:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Функция для вычисления чисел Фибоначчи в нашем исполнении называется `Fib()` и у нее один аргумент (обозначен как `n`) - это порядковый номер числа в последовательности. По этому номеру необходимо вычислить число. Процесс вычислений мы реализуем с помощью условного оператора, в котором проверяем значение аргумента функции. Точнее, проверяется условие `n==1 or n==2`. Условие истинно, если аргумент `n` равен 1 или 2, то есть если речь идет о первом или втором числе в последовательности. В этом случае число Фибоначчи равно 1. Поэтому если условие истинно, выполняется инструкция `return 1`. То есть если функции `Fib()` значение аргумента равно 1 или 2, функцией возвращается значение 1.

Пока все просто. Ситуация усложняется, если условие `n==1 or n==2` ложно. Ложно условие, если индекс у числа (его порядковый номер в последовательности) больше чем 2 (экзотические варианты с отрицательными и нецелыми индексами мы не рассматриваем). И здесь мы вспоминаем правило вычисления чисел в последовательности Фибоначчи: каждое число (кроме первых двух) - это сумма двух предыдущих.

Далее, если число Фибоначчи с порядковым номером `n` возвращается в результате вызова функции командой `Fib(n)`, то два предыдущих числа - это `Fib(n-1)` и `Fib(n-2)`. Поэтому в теле функции (с аргументом `n`) в условном операторе в `else`-блоке выполняется команда `return Fib(n-1)+Fib(n-2)`. Собственно, на этом все. У нас есть программный код функции, и этот программный код рабочий.

На заметку

Чтобы понять, почему же рекурсия "работает", разберемся, что происходит, если мы вызываем функцию `Fib()`. Например, если мы вызываем функцию с аргументом 1 или 2 (выражение `Fib(1)` или `Fib(2)` соответственно), то в теле функции "в игру" вступает тот блок в условном операторе, который соответствует истинному условию. А вот чтобы вычислить выражение `Fib(3)` в `else`-блоке условного оператора выполняется попытка вычислить выражение `Fib(2)+Fib(1)`. При этом снова вызывается функция `Fib()`, но только с аргументами 2 и 1. Что происходит в этом случае, мы уже знаем. После того, как вычислены значения `Fib(1)` и `Fib(2)`, может быть вычислено значение `Fib(3)`. Для вычисления значения выражения `Fib(4)` вычисляется сумма `Fib(3)+Fib(2)`, в которой нужно предварительно вычислить `Fib(3)` и `Fib(2)`. Как вычисляются эти значения, мы рассматривали выше. После их вычисления, вычисляется значение `Fib(4)`, и так далее.

Для проверки работы функции `Fib()` мы с ее помощью вычисляем и выводим в одну строку 15 первых чисел из последовательности Фибоначчи.

На заметку

По умолчанию после вывода в консольное окно функцией `print()` значений своих аргументов выполняется переход к новой строке. Поэтому если вызывать подряд несколько раз функцию `print()`, каждое новое сообщение (выводимый текст) будет появляться в отдельной строке. Технически причина в том, что автоматически в конце выводимого в консоль (окно вывода) текста добавляется инструкция перехода к новой строке. Чтобы изменить этот режим работы функции `print()` можно в явном виде указать текстовое значение для аргумента `end`. Значение обычно указывается по ключу, через знак равенства после названия аргумента `end`. Данное значение определяет символ (текст), который добавляется (вместо инструкции перехода к новой строке) в конце выводимого текста. Например, в команде `print(Fib(i),end=" ")` использована инструкция `end=" "`, поэтому после вывода строки добавляется пробел. Переход к новой строке не осуществляется.

Хотя рекурсия и делает обычно программные коды компактными и интуитивно понятными, метод рекурсивного определения функции нельзя назвать экономным в плане использования системных ресурсов. Вместе с тем, нередко рекурсия является единственно возможным на практике способом реализации программного кода функции.

Рассмотрим еще один пример рекурсии. На этот раз перепишем пример из листинга 3.5, в котором, напомним, представлена программа для решения алгебраических уравнений методом последовательных приближений. Только на этот раз функцию, с помощью которой реализуется итерационный процесс уточнения решения уравнения, реализуем через рекурсию. Обратимся к программному коду в листинге 3.8.

Листинг 3.8. Рекурсия для метода последовательных приближений

```
# Описание функции для решения уравнения.
# Используем рекурсию
def solve(f,x0,n):
    # Начальное приближение
    if n==0:
        return x0
    # Рекурсивное соотношение
    else:
        return solve(f,f(x0),n-1)
# Функция, определяющая уравнение
def eqn(x):
    # Значение функции
    return (x**2+5)/6
# Решаем уравнение
```

```
x=solve(eqn,0,10)
# Отображаем результат
print("Решение уравнения: x =",x)
```

Мы немного изменили программный код: частью переписали, частью упростили. Функция `solve()` предназначена для решения уравнения, определяемого первым аргументом (ссылка на функцию уравнения `f`) с начальным приближением, определяемым вторым аргументом (обозначен как `x0`) по количеству итераций, определяемому третьим аргументом (обозначен как `n`).

Основу программного кода функции составляет условный оператор, в котором проверяется условие `n==0`. Что означает истинность этого условия? Истинность этого условия означает, что мы вычисляем нулевое приближение для корня уравнения. Но нулевое приближение определяется вторым аргументом `x0` функции `solve()`. Поэтому нет ничего удивительного в том, что при истинном условии `n==0` командой `return x0` значение `x0` возвращается как результат функции `solve()`. Но если условие `n==0` ложно, в качестве результата возвращается выражение `solve(f,f(x0),n-1)`, в котором рекурсивно вызывается функция `solve()`.

На заметку

Чтобы понять происхождение выражения `solve(f,f(x0),n-1)`, нужно учесть следующее. Допустим, нам нужно вычислить n-ю итерацию для корня уравнения $x = f(x)$ с начальным приближением $x = x_0$. Если функция `solve()` позволяет решать подобные задачи, `f()` на программном уровне является реализаций функции уравнения, параметр `x0` определяет начальное приближение для корня, а `n` - номер итерации, то формально результат получаем с помощью команды `solve(f,x0,n)`. С другой стороны, вычисление n-й итерации на основе нулевого приближения - это то же самое, что вычисление $(n-1)$-го приближения на основе первого приближения x_1. Но $x_1 = f(x_0)$. Поэтому результат на программном уровне может быть представлен также выражением `solve(f,f(x0),n-1)`. Здесь мы учли, что первое приближение для корня уравнения можно вычислить как `f(x0)`.

Результат работы созданной нами функции `solve()` проверяем для решения уравнения $x = \frac{x^2+5}{6}$. С этой целью в программе описана функция `eqn()`. В итоге получаем такой результат:

Результат выполнения программы (из листинга 3.8)

```
Решение уравнения: x = 0.9999925289581152
```

Видим, что определенная через рекурсию функция для решения уравнения выполняется корректно. Желающие могут самостоятельно убедиться в том,

что с помощью этой функции можно найти второй корень уравнения. Для этого лишь нужно представить в ином виде само уравнение - как мы это делали ранее.

Лямбда-функции

Знаешь, одна из самых серьезных потерь в битве - это потеря головы.
Л. Кэрролл "Алиса в стране чудес"

Мы уже знаем, что имя функции можно присвоить в качестве значения переменной, после чего через эту переменную допустимо ссылаться на функцию. Но в языке Python есть и другая "крайность": мы можем создать функцию, но имя ей не присваивать вовсе. Такие функции называются *анонимными функциями* или *лямбда-функциями*. Зачем подобные функции нужны - это вопрос отдельный. Можно привести как минимум несколько примеров, когда использование лямбда-функций представляется оправданным:

- передача лямбда-функции аргументом другой функции;
- возвращение функцией в качестве результата другой функции;
- однократное использование функции.

При описании лямбда-функции используют ключевое слово `lambda`, после которого указываются аргументы функции, а через двоеточие - ее результат. То есть шаблон описания лямбда-функции такой (жирным шрифтом выделены ключевые элементы):

`lambda` *`аргументы`*`:` *`результат`*

У этой конструкции есть результат: ссылка на объект лямбда-функции. Поэтому в принципе `lambda`-конструкцию можно присвоить в качестве значения переменной, тем самым определив, в общем-то, обычную функцию. Небольшой пример, иллюстрирующий некоторые аспекты объявления и использования лямбда-функций, приведен в листинге 3.9.

Листинг 3.9. Лямбда-функции

```
# Функция для отображения значения
# другой функции
def find_value(f,x):
    print("x =",x,"-> f(x) =",f(x))
# Переменной присваивается
# ссылка на лямбда-функцию
my_func=lambda x: 1/(1+x**2)
```

```
# Проверяем результат
find_value(my_func,2.0)
# Аргументом функции передана
# лямбда-функция
find_value(lambda x: x*(1-x),0.5)
# Использование лямбда-функции
# в выражении
z=1+(lambda x,y: x*y-x**2)(2,3)**2
# Проверяем значение переменных
print("z =",z)
```

В программном коде объявляется функция `find_value()` с двумя аргументами. Это самая обычная функция. Мы собираемся ее использовать как вспомогательную для иллюстрации к использованию лямбда-функций. Первый аргумент функции `f`, как предполагается, является ссылкой на функцию, а `x` - предполагаемый аргумент для этой функции. В теле функции `find_value()` командой `print("x =",x,"-> f(x) =",f(x))` отображается сообщение, в котором, кроме прочего, использована инструкция `f(x)` вызова функции, на которую ссылается переменная `f`, с аргументом `x`.

Далее приведено несколько примеров описания и использования лямбда-функций. Так, в команде `my_func=lambda x: 1/(1+x**2)` есть описание лямбда-функции, а ссылка на эту функцию присваивается переменной `my_func`. Собственно функция описывается инструкцией `lambda x: 1/(1+x**2)`, в которой после ключевого слова `lambda` указано формальное название `x` для аргумента функции, а выражение `1/(1+x**2)` означает, что при аргументе функции `x` ее результатом будет значение `1/(1+x**2)`.

На заметку

Таким образом, здесь речь идет о функции $f(x) = \frac{1}{1+x^2}$.

Чтобы проверить "работоспособность" созданной нами функции, используем команду `find_value(my_func,2.0)`. В этой команде первым аргументом функции `find_value()` передается имя переменной `my_func`, которая содержит ссылку на лямбда-функцию. Второй аргумент `2.0` функции `find_value()` - это значение аргумента для функции, на которую ссылается переменная `my_func`. В результате в консольное окно выводится сообщение со значением `my_func(2.0)`. Результатом является значение `0.2` (легко проверить, что $f(2) = \frac{1}{1+2^2} = \frac{1}{5} = 0.2$).

Еще один вариант, который мы рассматриваем - передача лямбда-функции в качестве аргумента функции. Соответствующая команда выглядит как `find_value(lambda x: x*(1-x),0.5)`. Здесь первым аргументом

функции find_value() передается не переменная со ссылкой на лямбда-функцию, как в предыдущем случае, а сама лямбда-функция. Первый аргумент выглядит так: lambda x: x*(1-x). То есть аргументу функции x в соответствие ставится выражение x*(1-x) (это означает, что мы имеем дело с функцией $f(x) = x \cdot (1 - x)$). Второй аргумент функции find_value() - числовое значение 0.5. Это значение играет роль аргумента для функции, переданной первым аргументом функции find_value(). И хотя первым аргументом передана не переменная, а анонимная функция, суть дела не меняется: вычисляется значение $f(0.5) = 0.5 \cdot (1 - 0.5) = 0.25$, то есть вычисленное значение равно 0.25.

Третий пример использования лямбда-функции - использование такой функции в выражении. Примером служит команда z=1+(lambda x,y: x*y-x**2)(2,3)**2, в результате выполнения которой переменная z получает значение 5. Значение выражения 1+(lambda x,y: x*y-x**2)(2,3)**2 вычисляется так: к единице прибавляется значение выражения (lambda x,y: x*y-x**2)(2,3)**2. Данное выражение - это квадрат значения выражения (lambda x,y: x*y-x**2)(2,3). А это выражение, в свою очередь, есть ни что иное, как результат действия лямбда-функции (lambda x,y: x*y-x**2) на аргументы (2,3).

Инструкцией lambda x,y: x*y-x**2 определяется лямбда-функция двух аргументов (x и y), а результатом является значение x*y-x**2, вычисляемое на основе значений аргументов функции. То есть здесь имеем дело с функцией $f(x, y) = xy - x^2$. Если вычислять значение этой функции для аргументов $x = 2$, $y = 3$ получим $f(2,3) = 2 \cdot 3 - 2^2 = 6 - 4 = 2$. Следовательно, значением выражения (lambda x,y: x*y-x**2)(2,3) является 2, значением выражения (lambda x,y: x*y-x**2)(2,3)**2 является 4, а значение выражения 1+(lambda x,y: x*y-x**2)(2,3)**2, таким образом, 5.

В результате выполнения программы получаем такие сообщения в окне вывода:

Результат выполнения программы (из листинга 3.9)

```
x = 2.0 -> f(x) = 0.2
x = 0.5 -> f(x) = 0.25
z = 5
```

Еще одна программа, которая иллюстрирует возможные варианты в применении лямбда-функций, представлена в листинге 3.10. В этом случае лямбда-функция используются для того, чтобы создать функцию, которая возвращает в качестве результата другую функцию.

На заметку

Мы привыкли, и для нас кажется вполне естественным, что результат вызова функции - это некоторое значение (например, число, или текст). Здесь речь идет о том, что результатом функции является функция. Чтобы понять, как такое возможно (как функция может возвращать в качестве результата функцию), стоит вспомнить, что функция в Python - это объект типа `function` (об объектах мы будем говорить в следующих главах, здесь мы вдаваться в подробности относительно такого понятия, как объект, особо не будем). Для нас важно то, что описание функции - это некоторые данные, на которые можно выполнить ссылку и записать эту ссылку в переменную, которая играет роль имени функции. Поэтому если мы говорим, что результатом функции является функция, то означает это, как правило, следующее:

- в теле функции описывается (создается) некоторая функция;
- ссылка на созданную функцию возвращается как результат.

Стоит отметить концептуально важный момент. Когда мы вызываем функцию (например, командой вида `функция(аргументы)`), то сама инструкция вызова функции тоже является функцией. То есть выражение `функция(аргументы)` можно интерпретировать как имя функции (если точнее, то ссылку на функцию). Это ссылка именно на ту функцию, которая возвращается как результат вызываемой функции. Поэтому если после инструкции `функция(аргументы)` в круглых скобках указать аргументы, то получим вызов функции-результата. Другими словами, при вызове функции-результата имеем дело с выражением вида `функция(аргументы)(аргументы)`. Можно поступить иначе: присвоить некоторой переменной значение выражения `функция(аргументы)`. После выполнения команды `переменная=функция(аргументы)` с переменной можно обращаться как с именем функции.

Листинг 3.10. Функция как результат функции

```
# Функция в качестве результата
# возвращает функцию
def my_pow(n):
    return lambda x: x**n
# Проверяем результат
for n in range(1,4): # Внешний цикл
    for x in range(1,11): # Внутренний цикл
        # Выводим результат вызова функции
        print(my_pow(n)(x),end=" ")
    print() # Переходим к новой строке
```

В программном коде описывается функция `my_pow()`, у которой один аргумент `n`. Результатом функции возвращается инструкция `lambda x: x**n`. Это лямбда-функция, у которой один аргумент `x`, а результат - значение аргумента `x` в степени `n`.

На заметку

Функция `my_pow()` при вызове с одним аргументом (обозначим как n) возвращает в качестве результата функцию $f(x) = x^n$. Таким образом, инструкцию `my_pow(n)` можно рассматривать как имя функции, которой передается один аргумент. Чтобы вычислить значение x^n используем инструкцию `my_pow(n)(x)`.

Для проверки работы созданной нами функции мы запускаем вложенные операторы цикла, в которых перебираем значения переменной `n` (от `1` до `3`) и переменной `x` (от `1` до `10`). Для каждой из пар значений командой `print(my_pow(n)(x),end=" ")` выводятся (в одну строку - благодаря инструкции `end=" "`) значения выражения `my_pow(n)(x)`.

Результат выполнения программного кода представлен ниже:

Результат выполнения программы (из листинга 3.10)

```
1 2 3 4 5 6 7 8 9 10
1 4 9 16 25 36 49 64 81 100
1 8 27 64 125 216 343 512 729 1000
```

На заметку

Чтобы отображать значения в одной строке, функции `print()` передается именованный аргумент `end=" "`. Для перехода к новой строке используем `print()` без аргументов.

Первая строка значений - это натуральные числа от `1` до `10` в первой степени. Второй ряд - те же числа, но во второй степени. Третий ряд - третья степень.

На заметку

При описании лямбда-функций аргументам можно задавать значения по умолчанию. Значение по умолчанию указывается после имени аргумента через знак равенства. Например, инструкцией `lambda x=1: 1/x` определяется лямбда-функция для зависимости $f(x) = \frac{1}{x}$ со значением аргумента по умолчанию `1`. Если эту лямбда-функцию присвоить переменной (например, выполнить команду `f=lambda x=1: 1/x`), то затем при вызове функции можем аргумент указывать (например, `f(2)`) или не указывать (например, `f()`). В последнем случае используется значение аргумента по умолчанию (в данном случае `1`).

Локальные и глобальные переменные

А ты не путай свою личную шерсть с государственной.
из к/ф "Кавказская пленница"

Мы уже описали достаточно много функций, и практически каждый раз использовали в теле функций те или иные переменные. Особых недоразумений у нас не возникало. Однако здесь есть одна достаточно серьезная проблема, связанная с тем, где используемые нами переменные доступны, а где - нет. Переменные, которые доступны только в теле функции, называются *локальными*. Переменные, которые доступны вне функции, называются *глобальными*.

На заметку

В таких языках программирования, как C++ и Java, где переменные объявляются с указанием типа, проблем с разделением переменных на локальные и глобальные не возникает. Эту классификацию легко провести на основе того, где, в каком месте программного кода, переменная объявлена. В языке Python ситуация более сложная. Дело в том, что поскольку переменные в Python заранее (до первого использования) не объявляются, то определить, какая переменная локальная, а какая - глобальная, бывает не так уж и просто.

Рассмотрим некоторые несложные примеры, иллюстрирующие особенности использования локальных и глобальных переменных. Так, если в теле функции переменной присваивается значение, по умолчанию такая переменная является локальной. Причем правило действует, даже если ранее (до описания функции) была объявлена глобальная переменная с таким же именем. Как раз такой случай реализован в программном коде в листинге 3.11.

Листинг 3.11. Локальная и глобальная переменные

```
# Глобальная переменная
x=100
# Описание функции
def test_vars():
    # Локальная переменная
    x="локальная переменная"
    # Проверяем значение переменной
    # в теле функции
    print("В теле функции x =",x)
# Выполняем функцию
test_vars()
# Проверяем значение переменной
# вне тела функции
```

```
print("Вне функции x =",x)
```

На что здесь стоит обратить внимание: сначала мы создаем глобальную переменную x со значением 100. Затем объявляется функция test_vars(), в теле которой переменной x присваивается текстовое значение "локальная переменная". Командой print("В теле функции x =",x) проверяем значение переменной x. Это весь код функции.

Вне тела функции выполняется команда test_vars() , которой вызывается описанная выше функция, а затем командой print("Вне функции x =",x) проверяем значение переменной x. Результат выполнения программного кода приведен ниже:

Результат выполнения программы (из листинга 3.11)

```
В теле функции x = локальная переменная
Вне функции x = 100
```

Если по пунктам, то ситуация такая: мы переменной присваиваем значение, вызываем функцию, в которой переменной с таким же именем присваивается значение (значение выводится в консольное окно), и после вызова функции снова проверяем значение переменной. Как видно по результату выполнения программы, при проверке значения переменной x в теле функции и значения переменной x вне тела функции получаем различные значения. Что это означает? Означает это то, что в теле функции и вне тела функции мы имеем дело с разными переменными, хотя у них и совпадающие имена.

Если мы теперь немного изменим программный код функции test_vars(), убрав из него команду присваивания значения переменной x (но оставив команду отображения значения этой переменной), ситуация немного изменится. Новый программный код приведен в листинге 3.12.

Листинг 3.12. Глобальные переменные

```
# Описание функции
def test_vars():
    # Проверяем значение переменной
    # в теле функции. Значение переменной
    # x в теле функции не присваивается
    print("В теле функции x =",x)
# Глобальная переменная
x="глобальная переменная"
# Выполняем функцию
test_vars()
# Проверяем значение переменной
# вне тела функцииprint("Вне функции x =",x)
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 3.12)

```
В теле функции x = глобальная переменная
Вне функции x = глобальная переменная
```

Здесь мы в теле функции выводим на экран значение переменной x, но при этом используется то значение, что было присвоено переменной x до вызова функции.

На заметку

Обратите внимание, что значение переменной x присваивается после описания функции `test_vars()` но до ее вызова.

Таким образом, если в теле функции есть переменная, и этой переменной в теле функции присваивается значение, такая переменная будет локальной. Если в теле функции есть переменная, но значение в теле функции ей не присваивается, то будет использована глобальная переменная (переменная, значение которой присвоено вне тела функции - но до ее вызова).

В такую схему интерпретации уровня доступа переменных иногда нужно вносить изменения. Более конкретно, существует возможность в явном виде выделить (указать) глобальные переменные в теле функции. Для этого используют ключевое слово `global`. После этого ключевого слова указываются (через запятую, если их несколько) переменные, которые следует интерпретировать как глобальные. В листинге 3.13 приведен небольшой пример.

Листинг 3.13. Использование инструкции `global`

```
# Глобальная переменная
x=100
# Описание функции
def test_vars():
    # Объявляем глобальные переменные
    global x,y
    # Проверяем значение переменной x
    print("В теле функции x =",x)
    # Значение глобальной переменной y
    y=200
    # Проверяем значение переменной y
    print("В теле функции y =",y)
    # Значение глобальной переменной x
    x=300
```

```
# Выполняем функцию
test_vars()
# Проверяем значение переменной x
# вне тела функции
print("Вне функции x =",x)
# Проверяем значение переменной y
# вне тела функции
print("Вне функции y =",y)
```

В этом программном коде используется инструкция `global x,y`. Этой инструкцией переменные `x` и `y` объявляются как глобальные. Командой `print("В теле функции x =",x)` в теле функции проверяется значение переменной `x`, а до этого, до объявления функции, переменной `x` присвоено значение `100`. Именно это значение будет отображаться при выполнении указанной выше команды.

Свое первое значение переменная `y` получает в теле функции (команда `y=200`). Затем мы проверяем значение этой переменной (команда `print("В теле функции y =",y)`). Наконец, командой `x=300` присваивается новое значение переменной `x` (которая, напомним, является глобальной). На этом описание функции заканчивается.

Для проверки работы функции и роли глобальных переменных вызывается функция `test_vars()`, а после этого проверяются значения переменных `x` и `y`. Результат выполнения программы такой:

Результат выполнения программы (из листинга 3.13)

```
В теле функции x = 100
В теле функции y = 200
Вне функции x = 300
Вне функции y = 200
```

Общий вывод можно сформулировать так: если переменная глобальная, то изменение ее значения также носит "глобальный" характер.

Вложенные функции

Именно так выражается ее потребность в мировой гармонии.
из к/ф "Покровские ворота"

В Python функции можно описывать внутри функций. Такие функции будем называть *вложенными функциями* (или *внутренними функциями*). В этой ситуации было бы мало интересного, если бы не одна особенность: вло-

женная функция имеет доступ к переменным внешней функции (той функции, в которой описана вложенная функция).

Причины, по которым в программном коде используют вложенные функции, могут быть разными. Например, через вложенную функцию удобно реализовать вспомогательные вычисления. Другой вариант - вложенную функцию можно возвращать в качестве результата. Небольшой пример использования вложенных функций приведен в листинге 3.14. Там мы описываем функцию для вычисления суммы квадратов натуральных чисел. При этом в теле функции описаны две вложенные функции.

Листинг 3.14. Использование вложенных функций

```
# Внешняя функция
def sq_sum():
    # Вложенная функция для считывания
    # количества слагаемых
    def get_n():
        # Считываем числовое значение
        n=int(input("Слагаемых в сумме: "))
        # Результат функции get_n() - целое число
        return n
    # Вложенная функция для вычисления
    # суммы квадратов натуральных чисел.
    # Результатом является функция
    def find_sq_sum():
        # Начальное значение суммы
        s=0
        # Оператор цикла для вычисления суммы
        for i in range(1,n+1):
            s+=i**2 # Новое слагаемое в сумме
        # Результат функции find_sq_sum()
        return s
    # Определяем количество слагаемых в сумме
    n=get_n()
    # Результат функции sq_sum() - вложенная функция
    return find_sq_sum
# Вычисляем сумму квадратов чисел
z=sq_sum()()
# Отображаем результат
print("Сумма квадратов равна:",z)
```

В теле внешней функции, которая называется `sq_sum()`, и которая предназначена непосредственно для вычисления суммы квадратов натуральных чисел, описываются вложенные функции `get_n()` и `find_sq_sum()`.

При выполнении функции `get_n()` пользователю предлагается ввести целое число, которое определяет количество слагаемых в сумме. Введенное пользователем значение возвращается в качестве результата функции `get_n()`. В теле функции `get_n()` командой `n=int(input("Слагаемых в сумме: "))` отображается сообщение с просьбой ввести число и считывается это число. Число записывается в переменную `n`, которая командой `return n` возвращается в качестве результата функции.

На заметку

Для преобразования текстового значения, возвращаемого функцией `input()`, к целочисленному формату мы используем функцию `int()`.

Вложенная функция для вычисления суммы квадратов натуральных чисел `find_sq_sum()` содержит следующие команды. Командой `s=0` присваивается нулевое начальное значение переменной `s`, в которую записывается сумма квадратов чисел. Для вычисления суммы запускается оператор цикла. В операторе цикла переменная `i` пробегает значение от 1 до `n`, и за каждый цикл выполняется команда `s+=i**2`, которой к текущему значению суммы `s` прибавляется квадрат очередного слагаемого `i**2`. По завершении оператора цикла значение переменной `s` возвращается в качестве результата функции.

Здесь есть один интересный момент: в операторе цикла (в инструкции `range(1,n+1)`) используется переменная `n`, которой в теле вложенной функции `find_sq_sum()` значение не присваивается. Значение переменной `n` определяется в теле внешней функции `sq_sum()`: там есть команда `n=get_n()`, которой с помощью вызова вложенной функции `get_n()` определяется количество слагаемых и записывается в переменную `n`. После этого командой `return find_sq_sum` результатом функции `sq_sum()` возвращается функция `find_sq_sum()`.

Таким образом, когда мы вызываем функцию `sq_sum()`, в теле этой функции вызывается функция `get_n()` и, как следствие появляется сообщение с просьбой ввести числовое значение. Это значение используется во вложенной функции `find_sq_sum()`, которая возвращается в качестве результата. Таким образом, выражение `sq_sum()` можем интерпретировать как переменную, которая ссылается на функцию (то есть как имя функции). Если мы хотим вызвать эту функцию, необходимо добавить еще одну пару скобок. Более конкретно, выражение `sq_sum()()` фактически представляет собой вызов функции `find_sq_sum()` (с предварительным вызовом функции `get_n()`). Поэтому после выполнения команды `z=sq_sum()()` в переменную `z` записывается сумма квадратов натуральных чисел. Полу-

ченное значение отображаем с помощью команды `print("Сумма квадратов равна:",z)`. Ниже показан результат выполнения программы (жирным шрифтом выделено введенное пользователем значение):

Результат выполнения программы (из листинга 3.14)

```
Слагаемых в сумме: 10
Сумма квадратов равна: 385
```

Как видим, результат получился корректный.

Функция как результат функции

Независимые умы никогда не боялись банальностей.
из к/ф "Покровские ворота"

Выше, при рассмотрении лямбда-функций, мы уже познакомились с ситуацией, когда функция в качестве результата возвращает функцию. На практике такие задачи возникают достаточно часто. Откровенно говоря, возможны и другие, даже более экзотические варианты. Но экзотику мы оставим "на потом", а здесь еще раз вернемся к теме создания функций, которые возвращают своим результатом функции. На этот раз рассмотрим вопрос более основательно.

Важный момент, который мы уже подчеркивали и который следует иметь в виду - это то, что функция как таковая представляет собой некоторый объект. То есть когда в программном коде встречается описание функции, для этой функции создается объект. Ссылка на объект записывается в переменную - название функции. В этом отношении важно понимать, что переменная, ссылающаяся на число, концептуально мало чем отличается от переменной, ссылающейся на функцию (объект функции). Если мы в любой момент можем изменить значение "числовой" переменной, то точно так же мы можем изменить значение для переменной - имени функции.

Достаточно этой переменной присвоить новое значение - ссылку на объект другой функции. Более того, поскольку в Python тип для переменных не указывается и переменные могут ссылаться на любые значения, то теоретически никто не запрещает присвоить переменной, которая ранее ссылалась на функцию, ссылку на данные другого типа (например, на числовое значение или текст). Возможна и обратная ситуация: переменная, которая ранее на функцию не ссылалась, после присваивания значения может "стать функцией". Это не вопрос возможности, а скорее вопрос целесообразности.

Например, если мы хотим, чтобы функция в качестве результата возвращала функцию, то теоретически возможны такие варианты (наиболее очевидные):

- Воспользоваться лямбда-функцией и возвращать ссылку на эту функцию. С таким подходом мы встречались ранее.
- Описать в теле функции вложенную функцию и вернуть ссылку на эту функцию как результат внешней функции.
- В качестве результата функции можно возвращать имя другой функции (не вложенной). Такой подход тоже возможен.

В листинге 3.15 приведен пример функции, которая в качестве результата, в зависимости от переданного ей логического аргумента, возвращает ссылку на функцию для вычисления факториала или двойного факториала числа.

На заметку

Напомним, что факториалом числа называется произведение всех натуральных чисел, не превышающих это число: $n! = n \cdot (n-1) \cdot ... \cdot 2 \cdot 1$. Двойной факториал числа - также произведение натуральных чисел, но только "через одно число": $n! = n \cdot (n-2) \cdot (n-4)$ ... (последний множитель равен 2 для четного числа n и 1 для нечетного числа n).

Листинг 3.15. Вычисление факториала

```
# Функция для вычисления факториала
# и двойного факториала
def factor(mode=True):
    # Вложенная функция для вычисления
    # факториала числа
    def sf(n):
        s=1 # Начальное значение произведения
        i=n # Начальное значение индекса
        while i>1: # Условие
            s*=i # Умножение на индекс
            i-=1 # Уменьшение индекса на 1
        return s # Результат вложенной функции
    # Вложенная функция для вычисления
    # двойного факториала числа
    def df(n):
        s=1 # Начальное значение произведения
        i=n # Начальное значение индекса
        while i>1: # Условие
            s*=i # Умножение на индекс
            i-=2 # Уменьшение индекса на 2
```

```
        return s # Результат вложенной функции
    # Если аргумент mode равен True
    if mode:
        return sf # Ссылка на функцию для
                  # вычисления факториала
    # Если аргумент mode равен False
    else:
        return df # Ссылка на функцию для
                  # вычисления двойного факториала
#.Вызываем функцию factor() для вычисления факториала
print("5! =",factor()(5))
print("5! =",factor(True)(5))
# Вызываем функцию factor() для вычисления
# двойного факториала
print("5!! =",factor(False)(5))
```

Результат выполнения программного кода такой:

Результат выполне4ния программы (из листинга 3.15)

```
5! = 120
5! = 120
5!! = 15
```

Функция `factor()` описана с одним аргументом `mode`, у которого есть значение по умолчанию `True`. Таким образом, функция может вызываться без аргументов или с одним логическим аргументом. Если аргумент равен `True`, функцией в качестве результата возвращается функция, предназначенная для вычисления факториала числа. Если функция `factor()` вызывается с логическим аргументом `False`, результатом будет функция, предназначенная для вычисления двойного факториала числа.

В теле функции `factor()` описаны две вложенные функции: функцией `sf()` вычисляется факториал, а функцией `df()` вычисляется двойной факториал. В зависимости от значения аргумента `mode` в условном операторе в качестве результата функции `factor()` возвращается ссылка `sf` или `df`.

На заметку

В Python существуют более простые способы вычисления произведения натуральных чисел, чем создание для этой цели оператора цикла. Тем не менее, мы к ним принципиально не прибегаем.

При проверке функциональности созданной функции `factor()` нам на помощь приходят команды `factor()(5)`, `factor(True)(5)` (в обо-

их случаях речь идет о вычислении значения $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$) и `factor(False)(5)` (вычисление значения $5!! = 5 \cdot 3 \cdot 1 = 15$).

На заметку

Обратите внимание, что результатами выражений `factor()`, `factor(True)` и `factor(False)` являются ссылки на функции. Поэтому данные выражения следует рассматривать как "названия" функций. Чтобы вызвать эти функции, после "названий" в круглых скобках указывается аргумент функции (в данном случае число 5). Отсюда получаем соответственно `factor()(5)`, `factor(True)(5)` и `factor(False)(5)`.

Еще одно замечание относительно рассмотренной программы. Легко заметить, что программные коды вложенных функций `sf()` и `df()` практически идентичны и отличаются лишь значением декремента индекса в операторе цикла (величина, на которую уменьшается переменная `i` - при вычислении обычного факториала это 1, а при вычислении двойного факториала индекс уменьшается на 2). Учитывая это обстоятельство, рассмотренный ранее программный код можно было бы организовать несколько иначе. Пример приведен в листинге 3.16.

Листинг 3.16. Факториал и двойной факториал

```
# Функция для вычисления факториала
# и двойного факториала
def factor(mode=True):
    # Вложенная функция для вычисления
    # обычного/двойного факториала числа
    def f(n,d):
        s=1 # Начальное значение произведения
        i=n # Начальное значение индекса
        while i>1: # Условие
            s*=i # Умножение на индекс
            i-=d # Уменьшение индекса
        return s # Результат вложенной функции
    # Значение декремента для индекса
    d=1 if mode else 2
    # Результат функции
    return lambda n: f(n,d) # Лямбда-функция
# Вызываем функцию factor() для вычисления факториала
print("5! =",factor()(5))
print("5! =",factor(True)(5))
# Вызываем функцию factor() для вычисления
# двойного факториала
print("5!! =",factor(False)(5))
```

Результат выполнения этого программного кода точно такой же, как и при выполнении программы из листинга 3.15. Что касается самого программного кода, то теперь в функции `factor()` описана всего одна вложенная функция, которая называется `f()`, и у нее два аргумента: число `n`, для которого вычисляется обычный или двойной факториал, а также шаг дискретности `d` для уменьшения индексной переменной в операторе цикла.

Фактически, значение аргумента `d` определяет, какой факториал (обычный или двойной) будет вычисляться: для обычного факториала значение аргумента `d` равно `1`, а для вычисления двойного факториала значение этого аргумента должно быть равно `2`. В теле функции `factor()` командой `d=1 if mode else 2` (аналог *тернарного оператора*) переменной `d` присваивается значение `1` или `2` в зависимости от значения логического аргумента `mode` функции `factor()`. После этого в качестве результат функцией `factor()` возвращается ссылка на лямбда-функцию, которая задается выражением `lambda n: f(n,d)`.

Главная идея в том, что вложенной функции при вызове `f()` передаются два аргумента, в то время как в качестве результата функция `factor()` должна возвращать функцию, у которой один аргумент. Поэтому возвращать ссылку на функцию `f()` в качестве результата функции `factor()` - мысль плохая. Мы поступаем иначе: создаем анонимную функцию с одним аргументом, которая подразумевает вызов функции `f()` с этим же первым аргументом, а второй аргумент функции `f()` - ранее определенное числовое значение `d`.

Наконец, еще один способ решения все той же задачи проиллюстрирован в листинге 3.17.

Листинг 3.17. Еще один способ вычислить факториал

```
# Функция для вычисления факториала числа
def factorial(n):
    if n==1:
        return 1
    else: # Рекурсия
        return n*factorial(n-1)
# Функция для вычисления двойного факториала
def dfactorial(n):
    if n==1 or n==2:
        return n
    else: # Рекурсия
        return n*dfactorial(n-2)
# Функция для вычисления факториала
# и двойного факториала
```

```
def factor(mode=True):
    # Результат - ссылка на внешнюю функцию
    return factorial if mode else dfactorial
# Вызываем функцию factor() для вычисления факториала
print("5! =",factor()(5))
print("5! =",factor(True)(5))
# Вызываем функцию factor() для вычисления
# двойного факториала
print("5!! =",factor(False)(5))
```

В программном коде описаны две (внешние) функции: `factorial()` для вычисления факториала числа и `dfactorial()` для вычисления двойного факториала числа. При описании этих функций использовалась рекурсия. При этом программный код функции `factor()` исключительно простой и состоит всего из родной команды, которой в качестве результат функции возвращается выражение `factorial if mode else dfactorial`. Это выражение основано на *тернарном операторе* и его значение равно `factorial`, если аргумент `mode` принимает значение `True`. В противном случае результат выражения равен `dfactorial`. И в том, и в другом случае речь идет о возвращении ссылки на внешнюю (по отношению к функции `factor()`) функцию. Результат выполнения программного кода такой же, как и в предыдущих случаях.

Очередной пример, который мы рассмотрим здесь - это функция, которой аргументом передается функция, и которая в качестве результат возвращает функцию. Если более конкретно, то речь пойдет о вычислении производной.

На заметку

Производной для функции $f(x)$ называется некоторая функция (обозначается как $f'(x)$ или $\frac{df}{dx}$), которая равна пределу отношения разности значений функции $f(x)$ к приращению аргумента Δx при стремлении последнего к нулю: $f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$. Для нас же важно следующее: существует правило (правило вычисления производной), на основе которого одной функции можно в соответствие поставить другую функцию (которая называется производной). Например, если $f(x) = x^2$, то $f'(x) = 2x$, а если $f(x) = \frac{1}{1+x}$, то $f'(x) = -\frac{1}{(1+x)^2}$.

Вообще процедура вычисления производной - операция аналитическая. Но на практике часто прибегают к вычислению производной в приближенном виде, для чего выбирается малое, но конечное, приращение аргумента Δx, и в точке x производная $f'(x)$ вычисляется как отношение $f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$. Именно таким подходом мы воспользуемся для вычисления производной в программном коде.

В листинге 3.18 приведен пример программы, в которой описана функция, позволяющая вычислять (в приближенном виде) производную.

Исходная (дифференцируемая) функция передается аргументом, а производная функция возвращается как результат.

Листинг 3.18. Вычисление производной

```
# Функция для вычисления производной
def D(f):
    # Вложенная функция. Вычисляет
    # приближенное значение производной
    def df(x,dx=0.001):
        # Результат вложенной функции
        return (f(x+dx)-f(x))/dx
    # Результат функции - производная
    return df
# Первая функция для дифференцирования
def f1(x):
    return x**2
# Вторая функция для дифференцирования
def f2(x):
    return 1/(1+x)
# Функция для отображения производной в
# нескольких точках. Аргументы такие:
# F - производная (приближенная)
# Nmax - количество точек (минус один)
# Xmax - правая граница по аргументу
# dx - приращение аргумента
# f - производная (аналитически)
def show(F,Nmax,Xmax,dx,f):
    # Точки, в которых вычисляется производная
    for i in range(Nmax+1):
        x=i*Xmax/Nmax # Значение аргумента
        # Приближенное и точное значение производной
        print(F(x),F(x,dx),f(x),sep=" -> ")
# Производная для первой функции
F1=D(f1)
# Производная для второй функции
F2=D(f2)
# Значения в разных точках
# производной для первой функции
print("Производная (x**2)'=2x:")
show(F1,5,1,0.01,lambda x: 2*x)
```

```
# Значения в разных точках
# производной для второй функции
print("Производная (1/(1+x))'=-1/(1+x)**2:")
show(F2,5,1,0.01,lambda x: -1/(1+x)**2)
```

Результат получаем такой:

Результат выполнения программы (из листинга 3.18)

```
Производная (x**2)'=2x:
0.001 -> 0.01 -> 0.0
0.4009999999999986 -> 0.4099999999999999 -> 0.4
0.8009999999999962 -> 0.8099999999999996 -> 0.8
1.2010000000000076 -> 1.21 -> 1.2
1.6009999999999636 -> 1.6100000000000003 -> 1.6
2.0009999999996975 -> 2.0100000000000007 -> 2.0
Производная (1/(1+x))'=-1/(1+x)**2:
-0.9990009990008542 -> -0.990099009900991 -> -1.0
-0.6938662225923764 -> -0.688705234159781 ->
-0.6944444444444444
-0.5098399102682061 -> -0.5065856129686019 ->
-0.5102040816326532
-0.3903810118676132 -> -0.38819875776396895 ->
-0.39062499999999994
-0.3084706027516315 -> -0.30693677102516803 ->
-0.30864197530864196
-0.2498750624687629 -> -0.24875621890546595 -> -0.25
```

У функции `D()`, предназначенной для вычисления производной, один аргумент, который обозначен как `f` и который отождествляет название дифференцируемой функции. В теле функции `D()` описана вложенная функция `df()` с двумя аргументами. Первый аргумент `x` обозначает тот аргумент, ту точку, в которой вычисляется производная. Второй аргумент `dx` обозначает приращение по аргументу, на основании которого вычисляется выражение для производной. У аргумента `dx` имеется значение по умолчанию. В теле вложенной функции `df()` возвращается (как результат функции) значение выражения `(f(x+dx)-f(x))/dx` (отношение разности значений дифференцируемой функции к приращению аргумента). Функция `D()`, в свою очередь, возвращает в качестве результата ссылку `df` на вложенную функцию `df()`.

Также в программе описываются две функции для вычисления на их основе производных. Речь идет о функции `f1()` (соответствует зависимости $f(x) = x^2$) и функции `f2()` (соответствует зависимости $f(x) = \frac{1}{1+x}$). Производные для этих функций вычисляются командами `F1=D(f1)` и `F2=D(f2)`.

соответственно. После выполнения этих команд переменная F1 ссылается на функцию, соответствующую производной для функции f1(). Аналогично, переменная F2 является ссылкой на производную для функции f2(). Причем функциям F1() и F2() при вызове можно передавать один или два аргумента. Первый аргумент определяет точку, в которой вычисляется производная, а второй аргумент, если он задан, задает приращение по аргументу для вычисления производной.

Для вычисления и отображения значений производных функций в нескольких точках используется функция show(). Функция описана со следующими аргументами:

- через F обозначена функция для вычисления производной в числовом виде (на этой позиции при вызове функции указываются названия F1 и F2 функций F1() и F2());
- переменная Nmax обозначает количество интервалов, на которые разбивается диапазон изменения аргумента при вычислении производной (это число на единицу меньше количества точек, в которых вычисляется производная);
- переменная Xmax обозначает правую границу диапазона изменения аргумента (левая граница равна нулю);
- через dx обозначено приращение аргумента;
- через f обозначено имя функции, которая определяет аналитическое значение для производной.

При вызове функции show() последним аргументом передаются лямбда-функции: lambda x: 2*x (соответствует зависимости $f'(x) = 2x$) и lambda x: -1/(1+x)**2 ($f'(x) = -\frac{1}{(1+x)^2}$). При выполнении кода функции show() отображается три значения:

- значение производной, вычисленное на основе используемого по умолчанию приращения аргумента (функции F1() и F2() вызываются с одним аргументом);
- значение производной, вычисленное на основе явно указанного значения для приращения аргумента (функции F1() и F2() вызываются с двумя аргументами);
- значение производной, вычисленное на основе аналитического выражения.

Как видим, в основном результаты вычисления производных числовыми методами неплохо коррелируют с точными (вычисленными на основе аналитических выражений) значениями.

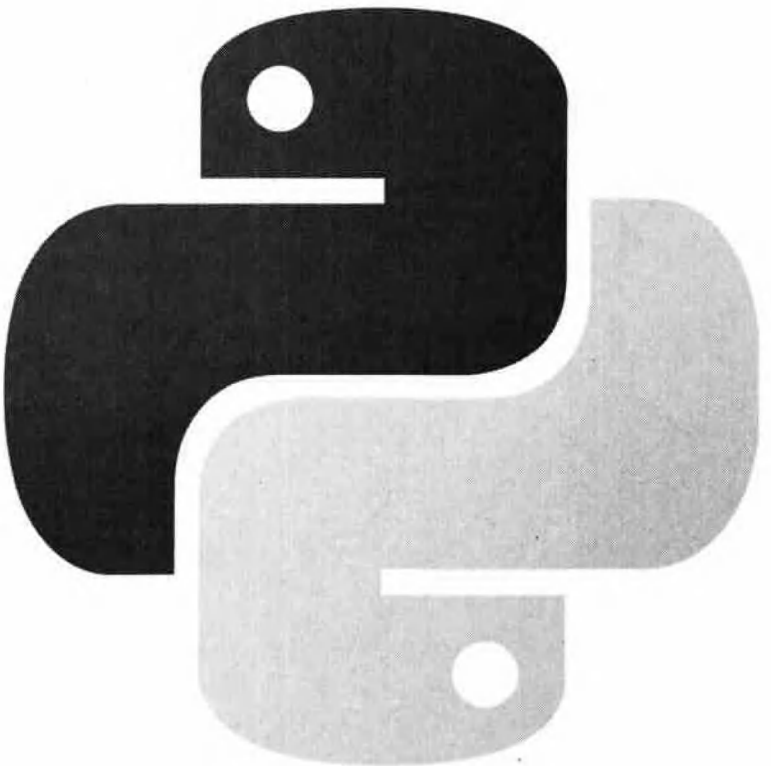
Резюме

Мастера не мудрствуют.
из к/ф "Покровские ворота"

1. При описании функции используем идентификатор `def`, после которого указывается имя функции, список аргументов (в круглых скобках) и, после двоеточия, программный код функции.
2. Инструкция `return` в теле функции приводит к завершению выполнения программного кода функции, а значение, указанное после инструкции `return`, возвращается в качестве результата функции.
3. При описании функции создается объект типа `function`. Имя функции является ссылкой на объект функции. Ссылка на объект функции может присваиваться переменной. В этом случае переменная будет ссылкой на функцию, и эта переменная может использоваться как имя функции.
4. Имя функции может передаваться аргументом другой функции.
5. Функция может возвращать в качестве результат функцию. В этом случае возвращается ссылка на функцию-результат.
6. У аргументов могут быть значения по умолчанию. Значение аргументов по умолчанию указываются через знак равенства после имени аргументов. Аргументы со значениями по умолчанию указываются в списке аргументов функции последними.
7. При описании функции в теле функции можно использовать вызов описываемой функции (обычно с другими аргументами). Такая ситуация называется рекурсией.
8. Лямбда-функция или анонимная функция - это функция без имени. Такие функции могут использоваться, например, для передачи аргументом в другие функции или возвращаться в качестве результата функции. Описывается лямбда-функция с помощью ключевого слова `lambda`, после которого указывается список аргументов и, через двоеточие, выражение, которое является результатом лямбда-функции.
9. Если переменной присваивается значение в теле функции, то такая переменная является локальной. Она доступна только в теле функции. Если переменная в теле функции входит в выражения, но значение ей

не присваивается, то такая переменная интерпретируется как глобальная. Чтобы явно объявить переменную в теле функции как глобальную, используют ключевое слово `global`.

10. В теле функции могут описываться другие функции. Такие функции называются вложенными. Вложенные функции имеют доступ к переменным в теле вешней функции.



Глава 4

Работа со списками и кортежами

Вам трудно угодить. Но я все-таки попробую.
из к/ф "Служебный роман"

В этой главе мы поближе познакомимся со *списками* и *кортежами*. Хотя если честно, то глава в основном посвящена спискам. С кортежами у нас состоится достаточно поверхностное, "шапочное" знакомство. Причина в том, что кортежи, на самом деле, очень "близки" к спискам, причем списки во многих отношениях являются более "гибким" типом данных, по сравнению с кортежами. Можно даже утверждать (с некоторой натяжкой), что кортежи - это такие особые списки. Поэтому мы сначала познакомимся со списками, определимся с основными методами их использования, а уже затем кратко остановимся на кортежах.

Знакомство со списками

Огласите весь список, пожалуйста!
из к/ф "Операция Ы и другие приключения Шурика"

Хотя списки мы подробно не обсуждали, но нам уже приходилось с ними сталкиваться. Но там, в предыдущих главах, мы в детали особо не вдавались. Теперь пришло время посвятить спискам больше внимания. И, надо сказать, они того стоят.

Список - это упорядоченный набор элементов. В некотором смысле списки в Python играют роль, аналогичную роли массивов в иных языках программирования. Но это только "в некотором смысле". На самом деле список в Python - это исключительно гибкая, эффективная и где-то даже уникальная штука. Достаточно сказать, что элементами списка могут быть объекты (данные) разного типа. Более того, элементами списка, в свою очередь, могут быть списки. То есть с помощью списков мы можем создавать вложенную структуру практически любой (в разумных пределах, разумеется) сложности.

На заметку

Если мы говорим о списке как таком, то речь идет о типе `list`. Но у списка есть элементы. И каждый из таких элементов может относиться к какому-то определенному типу.

Как это было с числовыми и текстовыми значениями, ссылка на список записывается в переменную. Такую переменную мы обычно и будем называть списком - если это не будет приводить к недоразумениям. Вообще же переменная *ссылается* на список. Сам список создается, в общем-то, просто. Достаточно перечислить элементы списка через запятую, а всю эту последовательность элементов заключить в квадратные скобки. Например, командой `nums=[1,2,3]` создается список из трех элементов, которые являются целыми числами (1, 2 и 3). Также можем воспользоваться для создания списка функцией `list()`. В этом случае аргументами функции передаются элементы списка. Особенно удобно воспользоваться функцией `list()` для создания списков на основе текстовых значений: если аргументом функции `list()` передать текст, то в результате получим список, элементы которого - буквы, формирующие текст. Так, при выполнении команды `symbs=list("Python")` создается список из букв `P`, `y`, `t`, `h`, `o` и `n`. Список может быть и более "изысканным": скажем, командой `data=["text",100,[5,10]]` создается список `data` из трех элементов, причем первый элемент списка - это текст `"text"`, второй элемент списка - целое число `100`, а третий элемент - список из двух элементов `[5,10]`.

На заметку

Создавать списки можно с помощью специальных генераторов списков. В квадратных скобках указывается выражение, зависящее от индексной переменной, которая пробегает определенные значения. Диапазон изменения индексной переменной также указывается в квадратных скобках. Например, командой `pows_of_two=[2**i for i in range(11)]` создается список из десяти элементов, представляющих собой степени двойки. Если после создания списка выполнить команду `print(pows_of_two)`, получим такой результат: `[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]`. В данном случае при формировании списка элементы списка определяются выражением `2**i`, и при этом переменная `i` пробегает значения в диапазоне от `0` до `10` включительно (инструкция `for i in range(11)`). Каждому индексу `i` соответствует элемент в списке, а значение элемента равно `2**i`.

Кроме диапазона изменения индексной переменной можем указать условие, которое должно выполняться для включения элемента в список. Как иллюстрацию рассмотрим команду `nums_list=[7*i+1 for i in range(20) if i%4==3]`, которой создается список `[22, 50, 78, 106, 134]` (чтобы увидеть этот список, разумно воспользоваться командой `print(nums_list)`). Полученный список состоит из чисел, которые при делении на `7` дают в остатке `1`, а при делении на `4` целой части от деления на `7` дают в остатке `3` (то есть если вычислить целую часть от деления числа на `7` и поделить ее на `4`, то в остатке будет `3`). Как получается список? Индекс `i` пробегает значения из диапазона от `0` до `19` включительно, и если для текущего значения индекса `i` выполняется условие `i%4==3` (остаток от деления индекса `i` на `4` равен `3`), то по формуле `7*i+1` вычисляется очередной элемент списка.

Обращение к элементу списка выполняется по индексу элемента. Элементы индексируются, начиная с нуля (то есть самый первый элемент имеет нулевой индекс). Индекс указывается в квадратных скобках после имени переменной, которая ссылается на список. Если воспользоваться приведенными выше примерами, то, скажем, инструкция `nums[0]` является ссылкой на первый элемент списка `nums` (то есть значение `1`). Для обращения к третьему (по порядку) элементу списка `symbs` (буква `t`) используем инструкцию `symbs[2]`.

При обращении к элементу списка можно указывать отрицательный индекс - в этом случае элементы "отсчитываются" начиная с конца списка. Так, самый последний элемент будет иметь индекс `-1`, предпоследний элемент будет иметь индекс `-2`, и так далее.

На заметку

Определить количество элементов в списке позволяет функция `len()`. Список передается аргументом функции: например, результатом выражения `len(symbs)` является значение `6` - количество элементов в списке `symbs`. Аналогично, результатом выражения `len(data)` будет значение `3`, поскольку в списке `data` всего `3` элемента: (обратите внимание: хотя последний элемент в списке `data` сам является списком, "учитывается" он именно как один элемент).

Поскольку индексация элементов списка начинается с нуля, то последний элемент списка имеет индекс, на единицу меньший, чем длина списка.

Список можно изменять поэлементно. Другими словами, обращаясь к элементу списка, мы можем не только "прочитать" значение элемента, но и изменить его. Например, после выполнения команды `nums[1]=-10` значение второго (по порядку) элемента списка `nums` станет равным `-10`, а сам список, (который до этого был `[1,2,3]`) станет `[1,-10,3]`.

При работе со списками мы можем обращаться сразу к нескольким элементам. В этом случае говорят о получении *среза*. Фактически, речь идет о том, что мы берем список, и "извлекаем" из него некоторую часть: все элементы, индексы которых попадают в указанный диапазон. Для большей конкретики представим, что у нас есть некоторый `список`. Если мы хотим извлечь из этого списка группу элементов (подсписок) с индексами от `i`-го до `j`-го включительно, то соответствующая инструкция будет выглядеть как `список[i:j+1]`. То есть в том месте, где мы раньше указывали индекс, теперь указывается, разделенные двоеточием, два индекса, которые и определяют элементы среза. Например, значением выражения `symbs[1:4]` является список из элементов `symbs[1]`, `symbs[2]` и `symbs[3]`, то есть список из букв `y`, `t` и `h`.

На заметку

Обратите внимание: если извлекаются элементы с индексами от `i` до `j` включительно, то в квадратных скобках перед двоеточием указывается индекс `i` первого извлекаемого элемента, а после двоеточия указывается индекс `j+1` - то есть этот индекс на единицу больше, чем индекс последнего из извлекаемых элементов. Еще одно уточнение касается смысла, которым мы наделяем слово "извлекается": важно понимать, что исходный список не меняется. Каким он был, таким и останется после процедуры получения среза.

Процедура получения среза проста, удобна и часто позволяет создавать эффективные программные коды. Существует несколько правил, которые окажутся вполне полезными для эффективного выполнения процедуры получения среза. Для удобства восприятия мы выделим основные форматы, в которых может выполняться срез. Сначала рассмотрим команду получения среза вида `список[i:j]`, то есть когда после имени списка в квадратных скобках (через двоеточие) указывается два индекса, причем оба эти индекса *неотрицательные*. В этом случае имеет смысл помнить, что:

- Результатом инструкции `список[i:j]` является список, состоящий из элементов с индексами, которые больше или равны `i` и меньше `j`. Например, если `m=[1,5,10,15,20,25]`, то результатом выражения `m[2:5]` является список `[10,15,20]` (элементы со 2-го по 4-й включительно).
- Если индексы `i` или `j` превышают длину списка (количество элементов в `списке` можно определить с помощью инструкции вида `len(список)`), то соответствующий индекс (или индексы) неявно заменяется на значение `len(список)` (то есть слишком большие индексу автоматически "урезаются"). Так, для списка `m=[1,5,10,15,20,25]` результатом выражения `m[2:100]` будет значение `[10,15,20,25]`, поскольку при вычислениях второй индекс `100` автоматически заменяется на значение `6` (значение выражения `len(m)` равно `6`).
- .Если не указать первый индекс `i`, он по умолчанию считается нулевым: например, команда `список[:j]` эквивалентна команде `список[0:j]`. Для списка `m` командой `m[:4]` возвращается список из элементов с индексами от `0` до `3` включительно (получаем список `[1,5,10,15]`, как для команды `m[0:4]`).
- Если не указать второй индекс `j`, то он по умолчанию равен длине списка: например, команда `список[i:]` эквивалентна команде `список[i:len(список)]`. Для списка `m` результатом выражения

m[3:] будет список [15,20,25] (такой же результат получаем как значение инструкции m[3:6]).

- Если первый индекс больше второго индекса или равен ему, срез будет пустым (получаем список без элементов). Например, в качестве результата выражением m[4:3] возвращается пустой список (то есть []).

На заметку

Таким образом, копию списка можем получить с помощью инструкции список[:].

В команде формата список[i:j] один из индексов или даже оба могут быть отрицательными. Если первый и/или второй индексы отрицательные, то чтобы понять, как обрабатывается такая инструкция следует заменить отрицательный индекс на len(список)+индекс (но при этом индекс -0 означает индекс 0). Скажем, если мы имеем дело со списком m=[1,5,10,15,20,25], то команда m[1:-2] эквивалентна команде m[1:4], поскольку значение -2 для индекса j эквивалентно значению 4 (длина списка 6 плюс формальное значение индекса -2, что в результате дает 4). Далее, по той же схеме легко можем показать, что команда m[-5:-2] эквивалентна команде m[1:4].

На заметку

Отрицательный индекс, таким образом, можем интерпретировать как порядковый номер элемента в списке, если считать элементы с конца. Правда, такая интерпретация отрицательных индексов не всегда удачна для понимания техники вычислений.

Кроме того, если указан "слишком отрицательный" индекс (то есть отрицательный индекс, который по модулю превышает длину списка), для такого индекса используется нулевое значение.

При получении среза можно указывать не два, а три индекса. В этом случае третий индекс (который, кстати, может быть отрицательным) определяет приращение для индекса элемента, включаемого в срез. Речь идет об инструкции вида список[i:j:k]. В результате получаем срез - список из элементов с индексами i, i+k, i+2*k, i+3*k и так далее (но последний индекс не больше, чем j). Например, для списка m=[1,5,10,15,20,25] результатом выражения m[1:5:2] является список-срез из элементов с индексами 1 и 3 (то есть список [5,15]).

Если команда получения среза использована в формате с тремя индексами (то есть если мы пытаемся получить срез командой вида список[i:j:k])

и третий индекс (индекс k) положительный, то справедливыми остаются приведенные выше правила интерпретации отсутствующих индексов, отрицательных индексов и индексов, превосходящих значение индекса последнего элемента. Но вот если третий индекс k отрицательный, то понять, по каким правилам получается срез, может быть не так уж просто. Главное правило, которое следует запомнить: если в инструкции получения среза третий индекс отрицательный, то элементы из списка выбираются в срез в обратном порядке, то есть справа налево. При этом первый индекс i должен быть таким, чтобы элемент, который соответствует этому индексу, находился справа от элемента, который соответствует индексу j. Если это условие не выполнено, то в результате получаем пустой список.

В случае если при отрицательном третьем индексе (имеется в виду индекс k) не указан первый индекс i, то в этой "позиции" используется индекс последнего элемента в списке. Если же не указан второй индекс j, то перебор элементов выполняется до начального элемента включительно.

На заметку

Что касается третьего индекса k (в команде вида `список[i:j:k]`), то его значение не может быть нулевым, а если этот индекс не указан (команда вида `список[i:j:]` или `список[i:j]`), то по умолчанию используется единичное значение.

Ситуация с выполнением среза командой вида `список[i:j:k]` с отрицательным третьим индексом k может показаться немного запутанной. Поэтому для большей наглядности приведем несколько простых примеров выполнения среза в формате команды с тремя индексами. Срез будем получать на основе списка `m=[1,5,10,15,20,25]`. Результаты ряда команд с пояснениями приведены в таблице 4.1.

Таблица 4.1. Выполнение среза

Команда	Результат	Пояснение
`m[5:1:-2]`	`[25,15]`	Выбираются элементы с 5-го по 1-й индекс с шагом два, причем элемент с индексом 1 в список-результат не включается. В итоге получается список из элементов с индексами 5 и 3

Команда	Результат	Пояснение
`m[5:-1:-1]`	`[]`	Хотя первый индекс формально больше второго, второй индекс отрицательный (значение `-1`) и соответствует последнему элементу в списке `m`. Первый индекс (значение `5`) также соответствует последнему элементу в списке `m`. Поэтому результат среза - пустой список
`m[-5:-1:-1]`	`[]`	Индекс `-5` соответствует второму слева элементу в списке `m`. Индекс `-1` соответствует последнему (шестому слева) элементу в списке `m`. Поскольку третий индекс (шаг приращения) отрицательный, в результате выполнения среза получаем пустой список
`m[-1:-5:-1]`	`[25,20,15,10]`	Срез получается выбором из списка `m` элементов с последнего (индекс `-1`) до третьего слева (индекс `-4`) с единичным шагом уменьшения индекса. Обратите внимание, что элемент с индексом `-5` в список-срез не включается
`m[::-1]`	`[25,20,15,10,5,1]`	Не указан первый и второй индекс, а третий индекс равен `-1`. В этом случае перебор элементов выполняется от последнего элемента до первого (включительно!) с единичным шагом (то есть элементы списка включаются в срез в обратном порядке)

Команда	Результат	Пояснение
`m[:-10:-1]`	`[25,20,15,10,5,1]`	Первый индекс не указан, а второй индекс `-10` выходит за диапазон значений индексов элементов списка `m`. В результате выборка элементов для среза выполняется (с единичным шагом уменьшения индекса), начиная с последнего элемента и заканчивая первым элементом списка `m`
`m[10:-10:-1]`	`[25,20,15,10,5,1]`	Первый и второй индексы выходят за диапазон реальных значений индексов элементов списка `m`. В этом случае при вычислении среза выбираются все элементы списка `m` (в обратном порядке, от последнего до первого включительно)
`m[10:0:-1]`	`[25,20,15,10,5]`	Значение первого индекса слишком большое (в списке `m` всего `6` элементов). В этом случае эффект такой же, как если бы был указан индекс последнего элемента списка `m`. Второй индекс нулевой. Поскольку третий индекс отрицательный (значение `-1`), то при получении среза выбираются элементы начиная с последнего с единичным шагом уменьшения индекса. Индексы выбираемых элементов должны быть больше нуля. Как следствие, первый элемент списка `m` (элемент с нулевым индексом) в срез не попадает

На заметку

Выше мы видели, насколько разнообразными могут быть индексы при получении среза. Но важно понимать, что во всех этих случаях речь шла об *интерпретации* различных числовых значений при использовании их в качестве индексов. Что касается списков, то напомним: индексация элементов списка начинается с нуля, а индекс последнего элемента в списке на единицу меньше длины списка (количества элементов в списке).

Далее мы рассмотри некоторые простые операции, которые выполняются со списками.

Основные операции со списками

- Меня вчера муха укусила.
- Да, я заметила.
- Или я с цепи сорвался.
- Вот это уже ближе к истине.
из к/ф "Служебный роман"

Понятно, что списки в программе создаются не просто так, а для чего-то. Так для чего же создаются списки и как они используются? Чтобы ответить на этот вопрос, имеет смысл рассмотреть те операции, которые могут выполняться со списками. Пока что мы знаем не очень много: как создать список, как обратиться к элементу списка и как получить срез. Последнюю операцию, очевидно, можно рассматривать как расширенный вариант процедуры обращения к элементу списка. Но вместе с тем, совершенно естественным образом возникает ряд вопросов. Например, как добавить или удалить элемент из списка? Как копировать списки? Конечно, вопросов может быть намного больше, но для начала разберемся хотя бы с этими.

Для добавления элемента в конец списка используют метод `append()`. Метод вызывается из списка (формат команды такой: `список.append(аргумент)`). Аргументом методу передается значение (переменная), добавляемое в список. Допустим, что у нас есть список `s=[10,20,30]`. Чтобы добавить в конец этого списка новый элемент (пускай для определенности это будет число `100`), используем команду `s.append(100)`. Если после этого проверить значение списка `s` (воспользовавшись командой `print(s)`), получим список `[10,20,30,100]`.

На заметку

Метод во многом похож на функцию, только функция существует сама по себе, а метод "привязан" к некоторому объекту и вызывается из этого объекта. В данном случае в роли объекта выступает список. Если мы говорим о методе `append()`, то у каждого списка есть свой метод с таким названием. При вызове метода нужно ука-

зать не только имя вызываемого метода, но и список, для которого вызывается метод. При этом используется "точечный" синтаксис: после имени списка через точку указывается имя вызываемого метода. Вызываемый метод имеет доступ к списку, из которого вызывается. Поэтому, например, метод `append()` может изменить тот список, из которого был вызван (добавить в список элемент).

Похожим образом действует и метод `extend()`, но этот метод предназначен для добавления в конец списка группы элементов. Добавляемая группа элементов, в свою очередь, обычно также реализуется в виде списка. Другими словами, с помощью метода `extend()` в конец списка может быть добавлен другой список, причем добавляется он не как отдельный элемент, а поэлементно: каждый элемент добавляемого списка становится элементом исходного списка (того, из которого вызывается метод `extend()`). В этом случае будем говорить о *расширении* списка.

На заметку

Если мы в список `A` добавляем список `B` с помощью метода `append()` (команда `A.append(B)`), то в список `A` будет добавлен новый элемент - список `B` (весь список `B` входит в список `A` как один элемент). Если мы в список `A` добавляем список `B` с помощью метода `extend()` (команда `A.extend(B)`), то в список `A` будут добавлены элементы списка `B`.

Как иллюстрацию рассмотрим небольшой программный код, представленный в листинге 4.1.

Листинг 4.1. Добавление элементов в список

```
# Базовый список
s=[10,20,30]
# Добавляем в список новый
# элемент - список
s.append([1,2])
# Проверяем результат
print(s)
# Расширяем список за счет
# другого списка
s.extend([3,4])
# Проверяем результат
print(s)
```

Результат выполнения этого программного кода приведен ниже:

Результат выполнения программы (из листинга 4.1)

```
[10, 20, 30, [1, 2]]
[10, 20, 30, [1, 2], 3, 4]
```

В данном случае создаем список s=[10,20,30] из трех элементов. Когда выполнятся команда s.append([1,2]) в список s добавляется новый элемент, и этот элемент - список [1,2]. Результатом является список [10,20,30,[1,2]]. То есть если мы добавляем список в другой список, то добавляемый список становится новым элементом в том списке, в который он добавляется. Когда же мы расширяем список s командой s.extend([3,4]), то список [3,4] добавляется в список s не как отдельный элемент, а в виде отдельных элементов. В результате (с учетом внесенных ранее в список s изменений) получаем список [10,20,30,[1,2],3,4].

При использовании функций append() или extend() элемент или элементы добавляются в конец списка. Метод insert() позволяет вставить элемент в список в той позиции, которую мы указываем сами. Первым аргументом методу передается индекс добавляемого в список элемента. Фактически, этот тот индекс, который будет иметь добавленный в список элемент. Элементы, которые уже были в списке, сдвигаются на одну позицию вправо. То есть тот элемент, который в списке находился на позиции, в которую добавляется новый элемент, не теряется, а сдвигается вправо. Вторым аргументом методу insert() передается собственно добавляемый в список элемент. Еще раз отметим, что с помощью метода insert() в список можно добавить только один элемент. Если нам нужно добавить несколько элементов в список (по принципу, как это делается методом extend(), когда добавляемые элементы сами организованы в виде списка), можем воспользоваться инструкцией присваивания значения срезу. В частности, если мы имеем дело со списком и нам нужно в позицию, которая соответствует i-му индексу, вставить некоторое значение (имеется в виду вставка элемента, а не присваивание элементу списка нового значения), то соответствующая команда будет выглядеть как список[i:i]=значение. Другими словами, слева от оператора присваивания размещается инструкция среза, в которой через двоеточие указывается один и тот же индекс - этот индекс и определяет позицию в списке, в которую выполняется вставка значения. Если же мы воспользуемся командой вида список[i:i+1]=значение, то будет выполнена не вставка нового элемента в список, а изменение значение элемента списка с индексом i.

На заметку

Если бы мы захотели получить срез некоторого списка с помощью команды вида список[i:i], то получили бы пустой список, поскольку в данном случае первый и второй индексы совпадают. Команда получения среза вида список[i:i+1] - это на самом деле список из одного элемента с индексом i. Таким образом, если мы присваиваем значение выражению список[i:i], то как бы получается, что присваиваем значение элементу, которого нет. В результате такой элемент в списке появляется. Если мы присваиваем значение выражению список[i:i+1], то зна-

чение присваивается элементу списка с индексом i. Более того, как мы увидим далее, если присвоить некоторое значение срезу список[i:i+k], то это приведет к тому, что в списке будут удалены элементы с индексами от i до i+k-1 включительно и вместо них добавлены элементы из списка, который присваивается в качестве значения.

Некоторые примеры вставки новых элементов в список представлены в листинге 4.2.

Листинг 4.2. Вставка элементов в список

```
# Исходный список
s=[10,20,30]
# В список добавляется элемент
# (число -5 вторым слева элементом в списке)
s.insert(1,-5)
# Проверяем результат
print(s)
# В список добавляется элемент - список
# (второй слева элемент - список [1,2])
s.insert(1,[1,2])
# Проверяем результат
print(s)
# В список добавляется два элемента
# (реализованы в виде списка из двух элементов)
s[2:2]=[3,4]
# Проверяем результат
print(s)
# Элементу списка присваивается значение
# (список [100,200] - значение третьего слева слева)
s[2:3]=[100,200]
# Проверяем результат
print(s)
```

Результат выполнения этой программы такой:

Результат выполнения программы (из листинга 4.2)

```
[10, -5, 20, 30]
[10, [1, 2], -5, 20, 30]
[10, [1, 2], 3, 4, -5, 20, 30]
[10, [1, 2], 100, 200, 4, -5, 20, 30]
```

В качестве исходного мы используем список s=[10,20,30]. Чтобы в этот список на позицию с индексом 1 (вторая позиция слева) добавить новый элемент (число -5) используем команду s.insert(1,-5), в которой метод insert() вызывается из списка s, а аргументами методу передаются индекс 1 (место в списке s для вставки элемента) и добавляемое в список

значение -5 (элемент, который добавляется в список s). После этого командой print(s) отображаем содержимое списка s. В данном случае получаем результат [10,-5,20,30]. Что мы видим? В список s, который изначально состоял из трех чисел [10,20,30] на позицию с индексом 1 (второй элемент слева) добавлен числовой элемент -5, а элементы списка 20 и 30 сдвинулись на одну позицию вправо. Далее выполняется команда s.insert(1,[1,2]). В результате на позицию с индексом 1 помещается новый элемент - список [1,2]. Этот список, как элемент списка s, размещается на второй позиции слева (индекс 1), а остальные три элемента списка s смещаются вправо. Вследствие этой операции список s становится таким: [10,[1,2],-5,20,30].

На следующем шаге мы добавляем в этот список два элемента. Для этого используем команду s[2:2]=[3,4]. В данном случае вставка выполняется на позицию с индексом 2 (третий элемент слева в списке s). Но теперь в список s добавляется не список-элемент, а два элемента - те элементы, которые формируют список [3,4]. Поэтому получаем список [10,[1,2],3,4,-5,20,30]. Наконец, после выполнения команды s[2:3]=[100,200] третий элемент списка s (элемент с индексом 2) меняет свое значение с 3 на [100,200]. В итоге получается список [10,[1,2],100,200,4,-5,20,30].

На заметку

Методы append(), extend() и insert() изменяют список, из которого вызываются и результат не возвращают.

Для объединения списков можно использовать и оператор сложения. Например, результатом выражения [1,2,3]+[4,5] является список [1,2,3,4,5].

Для удаления элемента из списка используют метод pop(). Аргументом методу передается индекс элемента, который нужно удалить. Другими словами, метод pop() позволяет удалить элемент списка с указанным индексом. Метод pop() не только изменяет список, из которого вызывается (в списке удаляется элемент), но еще и возвращает результат - значение удаляемого элемента. Если нужно удалить элемент списка с определенным значением, то используют метод remove(). Аргументом методу передается значение элемента, который нужно удалить из списка. Если в списке несколько элементов имеют такое значение, удаляется первый слева элемент. Метод remove() изменяет список, из которого вызывается и не возвращает результат.

Еще один примем, используемый для удаления элемента или элементов из списка состоит в том, что срезу списка в качестве значения присваивается

пустой список. Например, если мы хотим в некотором списке удалить все элементы с индексами от i до j включительно, то соответствующая команда могла бы выглядеть как список[i:j+1]=[].

На заметку

Напомним, что если присвоить срезу непустой список, элементы из среза в исходном списке будут удалены. Вместо них вставляются элементы из того списка, который присваивается в качестве значения срезу.

Удалить элемент из списка можно с помощью инструкции del - той же инструкции, которой удаляются из памяти переменные. Только если для удаления переменной ее имя нужно указать после инструкции del, то для удаления элемента из списка после инструкции del указывается ссылка на этот элемент (в формате список[индекс]). Например, если мы хотим удалить в списке s элемент с индексом i, команда для удаления этого элемента выглядит как del s[i]. Небольшой пример с удалением и заменой элементов в списке приведен в листинге 4.3.

Листинг 4.3. Удаление элементов из списка

```
# Создаем исходный список
s=[i*(10-i) for i in range(11)]
# Проверяем результат
print(s)
# Удаляем элемент с индексом 5
# и отображаем значение элемента
print("Удаляем элемент",s.pop(5))
# Проверяем содержимое списка
print(s)
# Удаляем элемент со значением 21
s.remove(21)
# Проверяем содержимое списка
print(s)
# Удаляем из списка элемент
# с индексом 3
del s[3]
# Проверяем содержимое списка
print(s)
# Удаляем несколько элементов в списке
s[2:5]=[]
# Проверяем содержимое списка
print(s)
# В списке удаляется два элемента
# и добавляется пять элементов
s[1:3]=[-1,-2,-3,-4,-5]
```

```
# Проверяем содержимое списка
print(s)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 4.3)

```
[0, 9, 16, 21, 24, 25, 24, 21, 16, 9, 0]
Удаляем элемент 25
[0, 9, 16, 21, 24, 24, 21, 16, 9, 0]
[0, 9, 16, 24, 24, 21, 16, 9, 0]
[0, 9, 16, 24, 21, 16, 9, 0]
[0, 9, 16, 9, 0]
[0, -1, -2, -3, -4, -5, 9, 0]
```

Сначалакомандой `s=[i*(10-i) for i in range(11)]` мысоздаемсписок из числовых значений (индексная переменная `i` пробегает значения от `0` до `10`, а соответствующий элемент списка определяется выражением `i*(10-i)`). В результате получаем список `[0,9,16,21,24,25,24,21,16,9,0]`.

Командой `print("Удаляем элемент",s.pop(5))`, во-первых, удаляется элемент с индексом 5, и, во-вторых, значение этого элемента отображается в консольном окне. Здесь мы воспользовались тем, что при выполнении инструкции `s.pop(5)` из списка `s` удаляется элемент с индексом 5, а значение этой инструкции - это значение удаляемого элемента (в данном случае речь идет об элементе со значением `25`). Результатом выполнения приведенной выше команды является сообщение `Удаляем элемент 25`, а сам список `s` при этом будет таким: `[0,9,16,21,24,24,21,16,9,0]` (то есть в нем действительно удален элемент со значением `25`).

С помощью команды `s.remove(21)` удаляем элемент со значением `21`. После этого список `s` станет таким: `[0,9,16,24,24,21,16,9,0]`. Важно здесь то, что до удаления элемента их в списке `s` было два: у двух элементов было значение `21`. При удалении элемента, удаляется тот, который в списке первый. А второй элемент со значением `21` в списке `s` остается.

Командой `del s[3]` из списка `s` удаляем элемент с индексом `3`. После этого список `s` равен `[0,9,16,24,21,16,9,0]`. Видим, что список потерял один элемент со значением `24` (тот, который находился на позиции с индексом `3`).

Чтобы удалить элементы с индексами от 2-го до 4-го включительно, используем команду `s[2:5]=[]`. Здесь срезу присваивается в качестве значения пустой список. Список `s` превращается в `[0,9,16,9,0]`.

Команда `s[1:3]=[-1,-2,-3,-4,-5]` иллюстрирует операцию вставки (с заменой) в список нескольких элементов. Удаляются элементы списка `s` с индексами 1 и 2, а вместо них вставляется группа элементов из списка `[0,-1,-2,-3,-4,-5,9,0]`.

Для проверки того, входит элемент в список или нет, используют оператор `in`. Результатом выражения формата `значение in список` является логическое значение `True` или `False` в зависимости от того, входит `значение` в `список`, или нет. Например, результатом выражения `2 in [1,2,3]` является значение `True`, поскольку в списке `[1,2,3]` представлено значение 2, а результатом выражения `0 in [1,2,3]` является значение `False`, поскольку в списке `[1,2,3]` значения 0 нет.

С помощью оператора `in` можно получить ответ на вопрос, входит или нет значение в список. Если нужно узнать, где именно в списке (на какой позиции) находится значение в списке, используют метод `index()`. Метод вызывается из списка, аргументом методу передается значение, которое проверяется на предмет вхождения в список. Результатом является индекс элемента с соответствующим значением. Если проверяемое значение сразу у нескольких элементов, возвращается индекс первого элемента с начала списка. Например, значением выражения `[1,2,3,2,1].index(2)` является 1, поскольку в списке `[1,2,3,2,1]` первый элемент со значением 2 находится на позиции с индексом 1.

На заметку

Обращаем внимание читателя и на то, как выше вызывался метод `index()`. А именно, мы вызывали этот метод непосредственно из списка, без предварительного присваивания списка в качестве значения переменной. Такой подход допустим, однако не всегда оправдан. В частности, метод `index()` не изменяет список, из которого вызывается. Поэтому в данной конкретной ситуации наш подход приемлем.

Методу `index()` вторым аргументом можно передать значение индекса, с которого в списке начинается поиск. Например, значением выражения `[1,2,3,4,3,2,1].index(2,3)` является 5. Поиск элемента со значением 2 начинается в списке `[1,2,3,4,3,2,1]`, начиная с элемента с индексом 3. Поэтому первая двойка в списке в область поиска не попадает и в результате возвращается индекс 5 того элемента, значение которого равно 2, но который находится справа от элемента с индексом 3 (сам элемент с индексом 3 также попадает в диапазон поиска). Методу `index()` также допустимо передать третий аргумент, который определяет конечную границу диапазона поиска элемента. Следует учесть, что если элемент с указанным значением не найден, возникает ошибка.

Метод `count()` позволяет вычислить количество элементов в писке с определенным значением. Метод вызывается из списка, а аргументом методу передается значение элементов для поиска. Например, значение выражения `[1,2,3,2,1,3,2].count(2)` равно 3, поскольку в списке `[1,2,3,2,1,3,2]` есть 3 элемента со значением 2.

Методе `reverse()` и функция `reversed()` позволяют изменить порядок следования элементов списка на противоположный. Различие между методом и функцией в том, что:

- метод `reverse()` (без аргументов) вызывается из списка, не возвращает результат и изменяет список (из которого вызывается);
- при вызове функции `reversed()` список передается ей аргументом, причем сам список не меняется, а результатом функции возвращается новый список с обратным порядком следования элементов (по сравнению со списком, переданным аргументом функции).

Аналогичная ситуация имеет место для метода `sort()` и функции `sorted()`. Метод и функция предназначены для сортировки элементов списка:

- метод `sort()` вызывается из списка, который собственно и сортируется - в порядке возрастания, если метод вызван без аргументов, и в порядке убывания, если метод вызван со значением аргумента `reverse=True` (аргумент обычно передается по ключу);
- функции `sorted()` передается исходный список, а результатом является отсортированный список (сортировка в порядке возрастания, а для сортировки в порядке убывания функции передают еще один аргумент `reverse=True`).

Существуют и другие очень полезные функции и методы, которые широко используются при работе со списками (и не только). Знакомиться с ними будем по мере необходимости.

На заметку

Есть ряд математических функций, предназначенных для работы со списками. Например, функции `min()` и `max()` предназначены соответственно для определения минимального и максимального значения из списка значений (список передается аргументом функции). Например, значение выражения `min([3,4,-1,7,0])` равно `-1` (наименьшее из значений, представленных в списке `[3,4,-1,7,0]`), а значение выражения `max([3,4,-1,7,0])` равно `7` (наибольшее из значений, представленных в списке `[3,4,-1,7,0]`). Функция `sum()` используется для вычисления суммы значений списка, переданного аргументом функции: значение выражения `sum([3,4,-1,7,0])` равняется `13` (сумма чисел в списке `[3,4,-1,7,0]`).

Копирование и присваивание списков

Ну, а это довесок к кошмару.
из к/ф "Старики-разбойники"

Есть одна, на первый взгляд простая, задача, которая имеет серьезные "последствия". Эта задача - создание копии списка. Формально здесь все просто: на основе уже существующего списка создается точно такой же. Почему же эта задача имеет последствия? Потому что "механизмы", которые задействованы или проявляются при решении этой задачи, дают ключ к пониманию многих важных моментов, связанных с программированием в Python. Чтобы не быть голословными, сразу обратимся к примеру.

Предположим, что у нас есть список - например, такой: `a=[10,20,30]`. После этого выполняем команду `b=a`. В соответствии с ней переменной `b` в качестве значения присваивается значение переменной `a`, которая, в свою очередь, ссылается на список. Логично ожидать, что переменная `b` тоже будет ссылаться на список - такой же, как и список, на который ссылается переменная `a`. В том, что списки одинаковые, убедиться легко. Для этого можем воспользоваться, например, командами `print(a)` и `print(b)`. Результаты будут одинаковыми (отображается один и тот же список `[10,20,30]`). Но ситуация не такая простая, как может показаться на первый взгляд. Допустим, командой `b[1]=0` мы изменили значение второго (с индексом `1`) элемента списка `b`. Если мы после этого командой `print(a)` проверим значение списка `a`, то получим результат `[10,0,30]`. Другими словами, изменения мы вносили в список `b`, а изменили при этом список `a` (список `b` тоже изменился - желающие могут в этом легко убедиться). Объяснение такое: при выполнении команды `b=a` переменная `b` получает ссылку не просто на такой же список, как переменная `a`, а на *тот же самый список*, что и переменная `a`. Таким образом, после выполнения команды `b=a` и переменная `a`, и переменная `b` ссылаются на один и тот же список. Поэтому когда мы командой `b[1]=0` изменяем элемент в списке, на который ссылается переменная `b`, мы тем самым изменяем тот же самый список, на который ссылается и переменная `a`.

Естественным образом возникает вопрос: что делать, если нам нужно создать копию списка (то есть если нам нужно, чтобы две переменные ссылались на разные списки с одинаковыми значениями)? Есть несколько вариантов решения проблемы. Во-первых, можем воспользоваться операцией получения среза. Например, значением выражения `a[:]` является список, такой же, как тот, на который ссылается переменная `a`. Поэтому если переменной `b` значение присваивается командой `b=a[:]`, то переменные `b` и `a` будут ссылаться на физически разные списки, но значения в этих списках будут одинаковые. Как следствие после выполнения команды `b[1]=0` спи-

сок b изменится (будет [10,0,30]), а список a останется прежним (значение [10,20,30]).

Также для создания копии списка используют метод copy(), который вызывается из списка, для которого создается копия. Например, в результате выполнения команды b=a.copy() создается копия того списка, на который ссылается переменная a, и ссылка на этот новый список записывается в переменную b. Здесь ситуация такая же, как и при использовании среза для создания копии. В листинге 4.4 приведен пример программы, в которой разными способами создаются копии списков.

Листинг 4.4. Создание копии списка

```
# Базовый список
a=[10,20,30]
# Проверяем содержимое списка
print("Список a:",a)
# Присваивание переменных
b=a
# Создание копии списка с помощью среза
c=a[:]
# Создание копии списка с помощью метода copy()
d=a.copy()
# Проверяем содержимое списков
print("Список b:",b)
print("Список c:",c)
print("Список d:",d)
print("Меняем значение элемента a[1]=0.")
# Изменяем элемент в базовом списке
a[1]=0
# Проверяем содержимое списков
print("Список a:",a)
print("Список b:",b)
print("Список c:",c)
print("Список d:",d)
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 4.4)

```
Список a: [10, 20, 30]
Список b: [10, 20, 30]
Список c: [10, 20, 30]
Список d: [10, 20, 30]
Меняем значение элемента a[1]=0.
Список a: [10, 0, 30]
Список b: [10, 0, 30]
```

```
Список c: [10, 20, 30]
Список d: [10, 20, 30]
```

На первый взгляд может показаться, что вопрос с созданием копии списка прояснился. Но это только на первый взгляд. Чтобы показать, что не все так просто, рассмотрим еще один небольшой пример. Пускай список `x=[10,20,[100,200],30,[300,400]]`. Важно здесь то, что среди элементов списка x есть элементы-списки (это список `[100,200]`, который является элементом `x[2]`, и список `[300,400]`, который является элементом `x[4]`). Мы командами `y=x[:]` и `z=x.copy()` создадим две копии списка x, изменим командами `x[2][1]=0` и `x[4]=0` элементы этого списка и затем проверим, как эти изменения повлияли на копии y и z списка x.

На заметку

Элемент `x[2]` - это третий с начала элемент списка x. Легко заметить, что данный элемент сам является списком: это список `[100,200]`, состоящий из двух элементов. Для нас важно то, что `x[2]` - это список. Поэтому выражение `x[2][1]` является ссылкой на элемент с индексом `1` в списке `x[2]`. Легко понять, что речь идет о втором элементе в списке `[100,200]`, то есть элементе со значением `200`. Поэтому командой `x[2][1]=0` в списке `[100,200]` (который сам является элементом списка x), значение `200` меняется на значение `0`.

Выражение `x[4]` является ссылкой на элемент `[300,400]` списка x. Данный элемент также является списком. Командой `x[4]=0` ему в качестве значения вместо ссылки на список присваивается число.

Соответствующий программный код представлен в листинге 4.5.

Листинг 4.5. Копирование вложенных списков

```
# Исходный список
x=[10,20,[100,200],30,[300,400]]
# Копия списка
y=x[:]
# Копия списка
z=x.copy()
# Проверяем содержимое списков
print("Список x:",x)
print("Список y:",y)
print("Список z:",z)
print("Меняем элементы: x[2][1]=0 и x[4]=0.")
# Изменяем значение элемента во внутреннем списке
x[2][1]=0
# Изменяем элемент во внешнем списке
x[4]=0
# Проверяем содержимое списков
print("Список x:",x)
```

```
print("Список y:",y)
print("Список z:",z)
```

При выполнении этого программного кода получаем такой результат:

Результат выполнения программы (из листинга 4.5)

```
Список x: [10, 20, [100, 200], 30, [300, 400]]
Список y: [10, 20, [100, 200], 30, [300, 400]]
Список z: [10, 20, [100, 200], 30, [300, 400]]
Меняем элементы: x[2][1]=0 и x[4]=0.
Список x: [10, 20, [100, 0], 30, 0]
Список y: [10, 20, [100, 0], 30, [300, 400]]
Список z: [10, 20, [100, 0], 30, [300, 400]]
```

Напомним, что нас интересует, насколько изменения в исходном списке x влияют на копии y и z. По логике, изменения в списке x не должны влиять на списки y и z. Что же мы видим? Во-первых, с самим списком x происходит то, что и ожидалось: после присваивания значений элементам списка, список меняется. Во-вторых, изменение элемента x[4] действительно не сказывается на списках y и z. Здесь нет ничего неожиданного. Но есть сюрприз, связанный с командой x[2][1]=0 (и это, в-третьих). Дело в том, что хотя изменения вносятся в список x, они имеют место и в списках y и z. Понятно, что ситуация требует разъяснений.

Разумно более детально остановиться на том, как создается копия списка, в котором есть элемент-список. Чтобы легче было понять происходящее, выделим ключевые моменты и термины:

- имеется список, который копируется - это *исходный*, или *копируемый список*;
- в копируемом списке есть элемент, который сам является списком - это *элемент-список*;
- список, созданный в результате копирования исходного, копируемого списка - это *список-копия*.

В принципе, процесс простой: для каждого из элементов создается копия. Если элемент - число, то в списке-копии соответствующий элемент имеет такое же числовое значение. Если в исходном списке элемент - это список, то ситуация несколько иная. Такой элемент на самом деле *ссылается* на список. Элемент-список в списке-копии будет иметь такое же значение, как и элемент-список в исходном, копируемом списке. Но, грубо говоря, это значение - адрес списка, на который ссылается элемент. Или другими словами, в списке-копии элемент-список технически получает в качестве значения

точно такую же ссылку, как и аналогичный элемент в исходном списке. Поэтому и получается, что оба элемента (в исходном списке и списке-копии) ссылаются физически на один и тот же список.

Копии, которые создаются с помощью операции получения среза или метода `copy()`, обычно называют *поверхностными копиями*. Если нам нужно, чтобы при создании копии списка создавались копии даже для внутренних списков (то есть списков, являющихся элементами списков), имеет смысл воспользоваться функцией создания "глубокой", или *полной копии* `deepcopy()`. Аргументом функции передается список, для которого создается копия. Функция становится доступной после подключения модуля `copy`. Если инструкция подключения модуля имеет вид `import copy`, то функция вызывается в формате `copy.deepcopy()`. Пример использования функции `deepcopy()` приведен в листинге 4.6.

Листинг 4.6. Создание полной копии списка

```
# Импортируем модуль copy
import copy
# Исходный список с элементами-списками
A=[[10,20],[[30,40],[50,60]]]
# Полная копия списка
B=copy.deepcopy(A)
# Проверяем содержимое списков
print("Список A:",A)
print("Список B:",B)
print("Выполняются команды A[0][1]=0 и A[1][1][1]=0.")
# Изменяем элементы списка
A[0][1]=0
A[1][1][1]=0
# Проверяем содержимое списков
print("Список A:",A)
print("Список B:",B)
```

В результате при выполнении кода получаем следующее:

Результат выполнения программы (из листинга 4.6)

```
Список A: [[10, 20], [[30, 40], [50, 60]]]
Список B: [[10, 20], [[30, 40], [50, 60]]]
Выполняются команды A[0][1]=0 и A[1][1][1]=0.
Список A: [[10, 0], [[30, 40], [50, 0]]]
Список B: [[10, 20], [[30, 40], [50, 60]]]
```

Исходный список создается командой `A=[[10,20],[[30,40],[50,60]]]`. В этом списке два элемента. Пер-

вый элемент [10,20] (ссылка A[0]) является списком из двух числовых элементов. Второй элемент [[30,40],[50,60]] (ссылка A[1]) - это тоже список из двух элементов, но эти элементы, в свою очередь, также являются списками (ссылка A[1][0] на список [30,40] и ссылка A[1][0] на список [50,60]). Таким образом, имеем дело с иерархией вложенных друг в друга списков. Командой B=copy.deepcopy(A) создаем полную копию списка A. Таким образом, переменные A и B ссылаются на физически разные списки, но с одинаковыми элементами. Поэтому при отображении с помощью функции print() содержимого списков A и B получаем одинаковые результаты.

На следующем этапе командами A[0][1]=0 и A[1][1][1]=0 изменяются значения элементов во внутренних списках. После этого снова проверяем содержимое списков A и B. Легко заметить, что список A изменился, а список B - нет.

Есть одна интересная особенность оператора присваивания, которая имеет отношение к спискам и о которой желательно знать. Касается она множественного присваивания, когда в левой части от оператора присваивания указывается (через запятую) несколько переменных, а справа (тоже через запятую) - присваиваемые переменным значения. Например, в результате выполнения команды x,y,z=1,2,3 переменной x будет присвоено значение 1, переменной y будет присвоено значение 2, а переменной z будет присвоено значение 3. Вообще количество переменных слева от оператора присваивания должно совпадать с количеством значений справа от оператора присваивания (чтобы было однозначное соответствие). Но может быть и более хитрая ситуация, когда справа значений больше, чем переменных слева. Тогда если перед одной из переменных указать звездочку *, то все "лишние" значения будут записаны в эту переменную в виде списка. Например, при выполнении команды x,*y,z=1,2,3,4,5 переменная x получит значение 1, переменная z получит значение 5, а переменная y будет ссылкой на список [2,3,4].

Еще одно замечание имеет отношение к многократному присваиванию (в выражении несколько операторов присваивания). Например, в результате выполнения команды x=y=1 переменные x и y получат значение 1. Причем если впоследствии изменить значение одной из переменных (например, выполнить команду x=2), то значение другой переменной (в данном случае y) не изменится. Но это если речь не идет о списках. Скажем, выполняется команда x=y=[1,2]. После ее выполнения переменные x и y ссылаются не просто на одинаковые списки, а на один и тот же список [1,2]. Если мы присвоим переменной x другой список (или значение иного типа), переменная y не изменится. Так, если после команды x=y=[1,2] выполняется ко-

манда `x=[3,4]`, то переменная `x` будет ссылаться на список `[3,4]`, а переменная `y` будет ссылаться на исходный список `[1,2]`. Но если мы изменим только элемент в исходном списке, то изменится значение обеих переменных. Допустим, после команды `x=y=[1,2]` выполняется команда `x[0]=0`. После этого переменная `x` ссылается на список `[0,2]` (как и должно быть), но и переменная `y` ссылается на этот список! Чтобы понять, почему так происходит, необходимо опять же учесть, что в Python переменные ссылаются на значения. Поэтому важно проводить четкую границу между изменением ссылки и изменением значения, на которое ссылка выполнена.

Списки и функции

Это простейшая цепь рассуждений.
из к/ф "Приключения Шерлока Холмса и доктора Ватсона"

Учитывая особую роль списков в языке Python, далее рассмотрим некоторые моменты (в основном прикладного характера), связанные с использованием списков. В этом разделе остановимся на том, как концепция списков соотносится с концепцией функций. Другими словами, в сферу нашего интереса на данном этапе попадают вопросы совместного использования функций и списков.

Сразу же в глаза бросаются две очевидных ситуации: это передача списка аргументом функции и возвращение функцией списка в качестве результата. С формальной точки зрения здесь нет ничего сложного. Так, если список передается аргументом функции, то при обработке такого аргумента просто необходимо учесть, что имеем дело именно со списком. Если функция возвращает в качестве результата список, то в теле функции список должен быть сформирован, а затем с помощью инструкции `return` "выдан" в качестве значения функции (возвращается на самом деле ссылка на список).

Как иллюстрацию рассмотрим программный код, в котором определены и затем используется несколько функций. В частности, описывается функция `rand_vector()` для создания списка указанной длины (определяется аргументом функции) со случайными целочисленными элементами, функция `show()` для отображения содержимого списка, а также функция `scal_prod()` для вычисления скалярного произведения векторов.

На заметку

Если есть два вектора $\vec{a} = (a_1, a_2, \ldots, a_n)$ и $\vec{b} = (b_1, b_2, \ldots, b_n)$, то скалярным произведением этих векторов называется число $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$.

Программный код, в котором реализованы перечисленные выше функции, а также проиллюстрированы их возможности, приведен в листинге 4.7.

Листинг 4.7. Функции и списки

```
# Подключаем модуль для
# генерирования случайных чисел
import random
# Функция для отображения содержимого списка.
# Аргумент функции - отображаемый список
def show(a):
    # Длина списка
    n=len(a)
    # Начальная фраза
    print(n,"D-вектор: <",sep="",end="")
    # Оператор цикла для отображения элементов
    for i in range(n-1):
        # Отображается элемент списка
        print(a[i],end="|")
    # Отображается последний элемент списка
    print(a[n-1],">.",sep="")
# Функция для создания списка со случайными
# целочисленными элементами. Аргумент функции -
# количество элементов в списке
def rand_vector(n):
    # Создаем пустой список
    r=[]
    # Оператор цикла для добавления новых
    # элементов в конец списка
    for i in range(n):
        # Добавляем в конец списка новый
        # элемент - случайное целое число
        # в диапазоне от 0 до 6
        r.append(random.randint(0,6))
    # Значение функции - список
    return r
# Функция для вычисления скалярного произведения
# двух векторов. Аргументы функции - списки, на основе
# которых реализуются векторы
def scal_prod(a,b):
    # Длина первого списка
    Na=len(a)
    # Длина второго списка
    Nb=len(b)
    # Наименьшее из двух значений
    N=min(Na,Nb)
```

```
    # Начальное значение суммы (через
    # определяется скалярное произведение)
    s=0
    # Оператор цикла для вычисления суммы
    for i in range(N):
        # Добавляем к сумме новое слагаемое
        s+=a[i]*b[i]
    # Результат функции
    return s
# Инициализация генератора случайных чисел
random.seed(2014)
# Первый случайный вектор
a=rand_vector(3)
# Второй случайный вектор
b=rand_vector(5)
# Отображаем первый вектор
show(a)
# Отображаем второй вектор
show(b)
# Вычисляем скалярное произведение
p=scal_prod(a,b)
# Отображаем результат для скалярного произведения
print("Скалярное произведение:",p)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 4.7)

```
3D-вектор: <2|4|3>.
5D-вектор: <1|2|1|3|3>.
Скалярное произведение: 13
```

Поскольку мы в программном коде собираемся генерировать случайные числа, командой `import random` подключаем модуль `random`, предназначенный именно для таких целей.

На заметку

В данном случае нам понадобится всего две функции из модуля `random`, хотя на самом деле там очень много других полезных функций.

При отображении содержимого списка явно указывается количество элементов в списке, а сами элементы размещаются в угловых скобках с вертикальной чертой, играющей роль разделителя. Все это реализуется в функции `show()`, предназначенной для отображения содержимого списка (переданного аргументом функции - аргумент обозначен как `a`). В теле функции ко-

мандой `n=len(a)` вычисляется длина списка (количество элементов в списке). После этого командой `print(n,"D-вектор: <",sep="",end="")` отображается текст. Первые два аргумента функции `print()` - это количество элементов в списке `n` и текст `"D-вектор: <"`. Функции `print()` по ключу передаются аргументы `sep=""` и `end=""` (оба значения - пустые текстовые строки).

Такие аргументы означают буквально следующее: разделителя (аргумент `sep`) между отображаемыми элементами нет (точнее, разделитель - пустой текст), а в конце (аргумент `end`) переход к новой строке не выполняется (вместо используемой по умолчанию инструкции перехода к новой строке - пустой текст). Для отображения элементов списка `a` в операторе цикла индексная переменная `i` пробегает значения от `0` до `n-2` (напомним, что инструкцией вида `range(m)` возвращается виртуальная последовательность целых чисел от `0` до `m-1` включительно) - то есть перебираются все элементы списка, кроме последнего.

Командой `print(a[i],end="|")` отображается очередной элемент списка, а сразу после этого элемента - вертикальная черта. Переход к новой строке не осуществляется. По завершении выполнения оператора цикла командой `print(a[n-1],">.",sep="")` отображается последний элемент и закрывающая угловая скобка. Разделитель между этими текстовыми фрагментами служит пустой текст (благодаря переданному по ключу аргументу `sep=""`).

Функция для создания списка со случайными целочисленными элементами называется `rand_vector()`. У нее один аргумент (обозначен как `n`) - предполагается, что это целое число, которое определяет размер списка. В теле функции командой `r=[]` создается пустой список.

После этого запускается оператор цикла, и за каждый цикл командой `r.append(random.randint(0,6))` в список (в конец списка) добавляется новый элемент - случайное целое число в диапазоне от `0` до `6`. Для генерирования случайных чисел мы использовали функцию `randint()` из модуля `random`. Аргументами функции передаются границы диапазона, в котором генерируются целые числа.

Результат функции `randint()`, как несложно догадаться - целое число.

На заметку

Как отмечалось выше, в модуле `random` много других функций, используемых, кроме прочего, для генерирования случайных чисел. Например:

- функцией `random()` генерируется случайное действительное число в диапазоне от `0` (включительно) до `1` (не включая);

- функция `uniform()` предназначена для генерирования равномерно распределенных действительных чисел в определенном диапазоне - границы диапазона передаются аргументами функции;
- функция `betavariate()` возвращает случайное число, подчиняющееся закону бета-распределения;
- функция `expovariate()` возвращает случайное число, подчиняющееся закону экспоненциального распределения;
- функция `gammavariate()` возвращает случайное число, подчиняющееся закону гамма-распределения;
- функция `gauss()` возвращает случайное число, подчиняющееся закону распределения Гаусса;
- функция `lognormvariate()` возвращает случайное число, подчиняющееся закону логнормального распределения;
- функция `normalvariate()` возвращает случайное число, подчиняющееся закону нормального распределения (фактически то же, что и распределение Гаусса);
- функция `paretovariate()` возвращает случайное число, подчиняющееся закону распределения Парето;
- функция `weibullvariate()` возвращает случайное число, подчиняющееся закону распределения Вейбулла.

Аргументами этим функциям передаются параметры соответствующих распределений (если таковые имеются).

Для добавления нового элемента в список использован метод `append()`, который вызывается из списка. После того, как список `r` сформирован, командой `return r` он возвращается как результат функции `rand_vector()`.

С помощью функции `scal_prod()` вычисляется скалярное произведение двух векторов, которые мы отождествляем со списками. Аргументы функции - списки. Результат функции - это сумма попарных произведений элементов списков. Здесь мы сталкиваемся с одной потенциальной проблемой: в списках может быть разное количество элементов.

В таком случае неявно будем полагать, что недостающие элементы нулевые. А поскольку произведение нуля на любое число равно нулю и соответствующее слагаемое в общую сумму вклада не вносит, то вместо того, чтобы в более "коротком" списке дописывать нулевые элементы, мы просто можем игнорировать лишние элементы в более "блинном" списке. Именно такой подход использован при определении результата функции `scal_prod()`. В теле функции командами `Na=len(a)` и `Nb=len(b)` определяется длина (количество элементов) первого и второго списков, переданных аргумен-

тами функции. Затем командой `N=min(Na,Nb)` в переменную `N` записывается минимальное из двух значений `Na` и `Nb`. Также командой `s=0` задается нулевое начальное значение для переменной, в которую будет записываться сумма - результат скалярного произведения. В операторе цикла индексная переменная `i` пробегает значения от `0` до `N-1` и служит индексом для перебора элементов в первом и втором списках. За каждую итерацию командой `s+=a[i]*b[i]` значение переменной `s` увеличивается на величину произведения соответствующих элементов (элементов с одинаковыми индексами) списков `a` и `b`. По завершении оператора цикла командой `return s` значение переменной `s` возвращается как результат функции.

Помимо описанных функций, в программном коде есть следующие команды:

- Инструкцией `random.seed(2014)` выполняется инициализация генератора случайных чисел. Вообще, эта команда необязательная. Можно было ее не использовать вовсе. Если все же мы вызываем функцию `seed()` с некоторым числовым аргументом, то генератором при каждом запуске программы будет генерироваться одна и та же последовательность случайных чисел.

> **На заметку**
>
> Пикантность ситуации в том, что любой генератор случайных чисел на самом деле "выдает" совсем не случайные числа. Поэтому правильнее говорить о псевдослучайных числах. Внешне очень похоже, что числа случайные - но это не так. Для получения последовательности таких чисел используется определенная формула или алгоритм. И чтобы все это "запустить", нужно какое-то "затравочное" или начальное значение, которое берется за основу при вычислении последовательности псевдослучайных чисел. Такое крайне важное для генератора случайных чисел значение обычно задается при инициализации генератора случайных чисел - фактически, в этом инициализация генератора чисел и состоит. Если мы в явном виде инициализируем генератор, вызвав функцию `seed()` с определенным аргументом, то этот аргумент используется для "старта" вычислительного процесса. Поэтому каждый раз последовательность псевдослучайных чисел будет одна и та же. Если явно инициализация генератора случайных чисел не выполняется, то "затравочное" значение определяется на основе системного времени, поэтому каждый раз при запуске программы получаем новую последовательность псевдослучайных чисел.

- Командами `a=rand_vector(3)` и `b=rand_vector(5)` создаются два случайных списка с количеством элементов 3 и 5 соответственно.
- Командами `show(a)` и `show(b)` содержимое списков отображается в консоли.
- Командой `p=scal_prod(a,b)` вычисляется скалярное произведение, а командой `print("Скалярное произведение:",p)` отображаем результат вычисления скалярного произведения.

Вложенные списки

Меня не проведешь. Приемы сыщиков я вижу на пять футов вглубь.
из к/ф "Приключения Шерлока Холмса и доктора Ватсона"

Хотя различных вариантов списков очень много, на одном типе списков остановимся подробнее. Это *вложенные списки*: списки, элементами которых, в свою очередь, тоже являются списки. Нас будет интересовать прикладной аспект в использовании такого рода "конструкций". Поэтому мы более подробно остановимся на том, как создавать вложенные списки, и как их затем использовать на практике. Понятно, что эти задачи можно решать различными способами. Рассмотрим небольшой пример, в котором описаны несколько функций, предназначенных для работы с матрицами: есть функция для создания матрицы с элементами - случайными числами, есть функция для создания единичной матрицы, описана функция для вычисления произведения квадратных матриц, а также имеется функция для построчного отображения матрицы в окне вывода (консоли).

🕮 На заметку

Нелишним будет освежить некоторые моменты, связанные с матрицами. Матрицей называется набор чисел, упорядоченных по строкам и столбцам - то есть в виде таблицы. Квадратной называется матрица, в которой одинаковое количество строк и столбцов. Матрица, таким образом, описывается набором элементов: если про матрицу A говорят, что она состоит из элементов a_{ij} (индексы $i = 1,2,\ldots,n$ и $j = 1,2,\ldots,m$), то это означает, что на пересечении i-й строки и j-го столбца находится число a_{ij}. Единичная матрица - это такая квадратная матрица, у которой на главной диагонали единичные элементы, а все прочие элементы равны нулю: то есть $a_{ij} = 1$ если $i = j$, и $a_{ij} = 0$ если $i \neq j$.

Если имеется две квадратные матрицы $\hat{A}$ с элементами a_{ij} и $\hat{B}$ с элементами b_{ij} (индексы $i,j = 1,2,\ldots,n$), то произведением этих матриц называется матрица $\hat{C} = \hat{A}\hat{B}$ с элементами $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$. Следует отметить, что умножать можно не только квадратные матрицы. Но в любом случае количество столбцов первой матрицы должно совпадать с количеством строк второй матрицы.

Соответствующий программный код представлен в листинге 4.8.

Листинг 4.8. Использование вложенных списков

```
# Подключаем модуль для генерирования случайных чисел
from random import *
# Функция для создания матрицы со случайными элементами
```

```
def rand_matrix(n,m):
    # Создаем матрицу с элементами - случайными числами
    A=[[randint(0,9) for j in range(m)] for i in range(n)]
    # Результат функции
    return A
# Функция для создания единичной матрицы
def unit_matrix(n):
    # Единичная матрица
    A=[[int(i==j) for i in range(n)] for j in range(n)]
    # Результат функции
    return A
# Функция для вычисления произведения квадратных матриц
def mult_matrix(A,B):
    # Размер матрицы
    n=len(A)
    # Матрица-результат с начальными нулевыми значениями
    C=[[0 for i in range(n)] for j in range(n)]
    # Первый индекс
    for i in range(n):
        # Второй индекс
        for j in range(n):
            # Внутренний индекс, по которому берется сумма
            for k in range(n):
                # Добавляем слагаемое в сумму
                C[i][j]+=A[i][k]*B[k][j]
    # Результат функции
    return C
# Функция для отображения матрицы
def show_matrix(A):
    # Перебор строк матрицы
    for a in A:
        # Перебор элементов в строке матрицы
        for b in a:
            # Отображаем элемент матрицы
           print(b,end=" ")
        # Переход новой строке (при выводе в консоль)
        print()
# Инициализация генератора случайных чисел
seed(2014)
# Матрица со случайными числами
A=rand_matrix(3,5);
# Матрица в виде списка
print("Список:",A)
# Матрица как она есть
print("Эта же матрица:")
show_matrix(A)
```

```
# Единичная матрица
E=unit_matrix(4)
# Отображаем матрицу
print("Единичная матрица:")
show_matrix(E)
# Первая матрица
A1=rand_matrix(3,3)
# Вторая матрица
A2=rand_matrix(3,3)
# Произведение матриц
A3=mult_matrix(A1,A2)
# Отображаем первую матрицу
print("Первая матрица:")
show_matrix(A1)
# Отображаем вторую матрицу
print("Вторая матрица:")
show_matrix(A2)
# Отображаем произведение матриц
print("Произведение матриц:")
show_matrix(A3)
```

В программном коде командой `from random import *` подключаем модуль `random` для генерирования случайных чисел (причем при таком подключении модуля для вызова функций из этого модуля имя модуля явно указывать не нужно). Нам понадобится из модуля `random` функция `rendint()`, с помощью которой будем генерировать элементы матрицы в теле функции `rand_matrix()`. Функция описана с двумя аргументами `n` и `m`, которые определяют соответственно количество строк и столбцов в матрице, возвращаемой в качестве результата.

Матрица создается командой `A=[[randint(0,9) for j in range(m)] for i in range(n)]`. В данном случае мы использовали вложенные *генераторы списков*. Инструкцией `[randint(0,9) for j in range(m)]` формируется список из `m` элементов, причем каждый элемент - результат вызова функции `randint(0,9)`, то есть целое случайное число в диапазоне от 0 до 9 включительно.

Сама же инструкция `[randint(0,9) for j in range(m)]` указана как такая, что формирует элемент списка для внешнего генератора списков. В этом внешнем генераторе списков индексная переменная пробегает целочисленные значения от 0 до `n-1`. Поэтому в результате получаем список из `n` элементов, причем каждый элемент этого списка сам является списком из `m` элементов (а эти элементы - случайные числа).

Функция для создания единичной матрицы называется `unit_matrix()` и у нее один аргумент (обозначен как `n`) - размер (ранг) матрицы (количество строк и оно же количество столбцов, поскольку матрица по определению квадратная). Единичная матрица `A` в теле функции вычисляется командой `A=[[int(i==j) for i in range(n)] for j in range(n)]`.

Команда очень похожа на ту, что была в теле функции `rand_matrix()`, но есть небольшие отличия. Во-первых, обе индексных переменных (во внешнем и внутреннем генераторе списков) пробегают одинаковые диапазоны значений (в квадратной матрице количество строк равняется количеству столбцов). Во-вторых, при формировании элементов во внутреннем генераторе списков вместо функции `randint()` использована инструкция `int(i==j)`. Здесь логика такая: значение выражения `i==j` равняется `True`, если индексы `i` и `j` одинаковые, и `False`, если эти индексы разные. Логическое значение при преобразовании с помощью функции `int()` к целочисленному виду дает `1` там, где было `True`, и `0` там, где было `False`. В результате получаем список, состоящий из `n` списков, каждый из которых, в свою очередь, состоит из `n` нулей или единиц.

В каждом внутреннем списке только одна единица (остальные элементы нулевые), причем единица во внутреннем списке находится на той позиции, которая совпадает с позицией внутреннего списка во внешнем списке. На языке матриц это означает, что единицы находятся только на главной диагонали - то есть мы создали единичную матрицу. Эта матрица и является результатом функции `unit_matrix()`.

Функция `mult_matrix()` предназначена для вычисления произведения квадратных матриц. Ее аргументы `A` и `B`, как мы предполагаем, являются именно теми матрицами, которые нужно перемножить. Результат функции - матрица, которая получается в результате умножения матриц, переданных аргументами функции.

В теле функции командой `n=len(A)` определяется размер матрицы - первого аргумента функции `mult_matrix()`. Строго говоря, в переменную `n` записывается количество элементов в списке `A`. Элементами списка `A`, в свою очередь, также являются списки (внутренние списки). Количество элементов в списке `A` - это количество строк в соответствующей матрице. Количество элементов во внутренних списках - количество столбцов матрицы. Мы предполагаем, что матрица `A` квадратная, и количество столбцов у нее такое же, как количество строк. Более того, мы также исходим из того, что второй аргумент функции `mult_matrix()` - матрица `B`, - также квадратная и того же размера, что и матрица `A`. Но проверка всех этих важных моментов в программном коде не реализуется. Желающие могут подумать, как ее можно было бы реализовать.

На заметку

Удобный механизм обработки всевозможных "неприятностей" (ошибок, возникающих при выполнении программы) связан с использованием конструкции `try-except`. Этот подход мы уже обсуждали в одной из предыдущих глав.

В теле функции `mult_matrix()` командой `C=[[0 for i in range(n)] for j in range(n)]` создаем вложенный список `C` с нулевыми элементами. Этот список задает структуру матрицы-результата функции. Все элементы этой матрицы перед началом вычислений нулевые.

Далее в теле функции запускается три вложенных оператора цикла: в первых двух операторах перебираются индексы элементов матрицы `C`, а по индексу третьего оператора цикла вычисляется сумма. При этом выполняется всего одна команда `C[i][j]+=A[i][k]*B[k][j]`, которой сумма для значения элемента `C[i][j]` увеличивается на величину произведения элементов `A[i][k]` и `B[k][j]` матриц `A` и `B`. После завершения вычислений матрица `C` возвращается как результат функции.

Для отображения матриц в более приемлемом с математической точки зрения виде определяем функцию `show_matrix()`, аргументом которой передается матрица для отображения.

На заметку

Если попытаться отобразить вложенный список с помощью функции `print()`, то элементы списка мы, конечно же, увидим. Но отображаться они будут не в виде таблицы, как принято отображать матрицы, а как вложенные списки.

В теле функции `show_matrix()` запускаются вложенные операторы цикла. Во внешнем цикле переменная `a` пробегает набор значений из списка `A`. Другими словами, переменная `a` последовательно принимает значения элементов из списка `A`. Поэтому `a` можем интерпретировать как элемент списка `A`. Поскольку список `A` сам состоит из списков, то `a` - это список. Там находятся элементы, соответствующие определенному ряду в матрице `A`. Во втором, внутреннем операторе цикла переменная `b` пробегает значения элементов из списка `a`. Таким образом, если `a` - это ряд элементов в матрице `A`, то `b` - это один из элементов этого ряда. Командой `print(b,end=" ")` элемент ряда отображается в окне вывода, переход к новой строке не осуществляется, а вместо этого добавляется пробел. Таким образом, элементы ряда отображаются в одну строку через пробел. После того, как элементы определенного ряда матрицы перебраны, и работа внутреннего оператора цикла завершена, выполняется команда `print()`. В результате выполняется переход к новой строке для вывода информации. Во внешнем операторе цик-

ла переменная a принимает новое значение, и все начинается снова: элементы ряда отображаются в одну строку через пробел, и так далее.

Помимо описанных функций есть несколько команд, которые позволяют проиллюстрировать, как эти самые функции можно использовать.

Инициализацию генератора случайных чисел выполняем командой `seed(2014)`. Еще раз напомним, что такую инициализацию можно было бы и не выполнять.

Матрица со случайными числами создается командой `A=rand_matrix(3,5)`. В данном случае речь идет о матрице из 3 строк и 5 столбцов. Для сравнения командой `print("Список:",A)` отображаем содержимое списка A стандартным способом, а затем командой `show_matrix(A)`.

Единичная матрица ранга 4 создается командой `E=unit_matrix(4)`. Матрицу отображаем командой `show_matrix(E)`.

Кроме этого, командами `A1=rand_matrix(3,3)` и `A2=rand_matrix(3,3)` создаем две случайные квадратные матрицы ранга 3. Произведение этих матриц вычисляем с помощью инструкции `A3=mult_matrix(A1,A2)`. После этого командами `show_matrix(A1)`, `show_matrix(A2)` и `show_matrix(A3)` отображаем две исходные матрицы, а также матрицу-произведение.

Ниже показано, что может быть выведено при выполнении программного кода:

Результат выполнения программы (из листинга 4.8)

```
Список: [[5, 9, 6, 3, 5], [3, 6, 6, 7, 0], [9, 6, 5, 2, 9]]
Эта же матрица:
5 9 6 3 5
3 6 6 7 0
9 6 5 2 9
Единичная матрица:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
Первая матрица:
7 2 7
3 4 0
1 2 3
Вторая матрица:
3 4 6
6 6 5
```

```
1 2 4
Произведение матриц:
40 54 80
33 36 38
18 22 28
```

Желающие могут самостоятельно проверить правильность вычисления произведения матриц.

Одно из конкурентных преимуществ языка Python состоит как раз в том, что одна и та же задача может решаться разными способами, причем нередко эти способы отличаются разительно. Поэтому очевидно, что предложенный выше подход не является единственно возможным. С некоторыми другими концепциями составления программных кодов мы еще познакомимся по мере изучения материала книги.

🕮 На заметку

При создании списков (в том числе и вложенных), нередко используется оператор повторения * (звездочка). Если этот бинарный оператор (который формально совпадает с оператором умножения) применяется к списку (один операнд - список, другой операнд - целое число), то результатом будет новый список, который получается повторением элементов из исходного списка. Количество повторений определяется целочисленным операндом. Например, в результате выполнения команды `A=[0]*5` (или команды `A=5*[0]`) переменная `A` будет ссылаться на список из пяти нулей `[0,0,0,0,0]`. Если `B=[1,2]*3`, то на самом деле результатом будет список, получающийся троекратным повторением пары элементов 1 и 2, то есть список `[1,2,1,2,1,2]`. Или, скажем, после выполнения команды `C=[[1]*2]*3` переменная `C` будет ссылаться на список `[[1,1],[1,1],[1,1]]`. Почему так? Потому что результатом выражения `[1]*2` является список `[1,1]`, а результатом выражения `[1,1]*3` является список `[[1,1],[1,1],[1,1]]` (троекратное повторение элемента `[1,1]`).

Стоит заметить, что оператор повторения * может применяться не только к спискам, но, например, и к тексту.

Знакомство с кортежами

- Что за вздор. Как Вам это в голову взбрело?
- Да не взбрело бы, но факты, как говорится, упрямая вещь.
из к/ф "Чародеи"

Кортеж представляет собой упорядоченный набор некоторых элементов. В этом смысле он мало чем отличается от списка. То есть и список, и кортеж - это упорядоченный набор элементов. В чем же различие? Различие - в способе или механизме реализации, а также в том, какие операции допустимы со списками и кортежами. Можно также сказать, что в основе списков

и кортежей одна и та же структура - упорядоченный набор элементов. Эта структура может реализоваться через списки, а может реализоваться через кортежи.

Если от абстрактных понятий перейти к более практическим вопросам, то следует отметить, что принципиальное отличие кортежей от списков состоит в том, что кортежи нельзя изменять. То есть после того, как кортеж создан, внести в него изменения уже не получится.

На заметку

На первый взгляд эта особенность кортежей очень сильно ограничивает область их применимости. Но это только на первый взгляд. На самом деле данная особенность скорее "технического" характера и ее во многих случаях удается обойти. Например, если переменная ссылается на кортеж и нам нужно внести в кортеж изменения, то можно создать новый кортеж и ссылку на этот кортеж присвоить переменной.

С точки зрения "типологии" языка Python кортеж относится к типу `tuple`. Поэтому нет ничего удивительного, что создать кортеж можно с помощью функции `tuple()`. Если аргумента у функции нет, то будет создан пустой кортеж. Если аргументом функции `tuple()` передать, например, список или текстовую строку, получим кортеж, состоящий соответственно из элементов списка или букв текстовой строки. Также можем создать кортеж, если в круглых скобках перечислить через запятую элементы, формирующие кортеж.

На заметку

Пустые круглые скобки означают пустой кортеж. Если в кортеже один элемент, и мы создаем такой кортеж с помощью круглых скобок, то после этого элемента следует поставить запятую. Если запятую не поставить, то получим не кортеж, а просто тот элемент, который в круглых скобках. Например, командой `a=()` создается пустой кортеж, и ссылка на него записывается в переменную `a`. Командой `a=(10,20)` создается кортеж из двух элементов (`10` и `20`). А командой `a=(100)` кортеж не создается: переменная `a` ссылается на целочисленное значение `100`. Чтобы создать кортеж, состоящий всего из одного элемента `100`, используем команду `a=(100,)`.

Еще один важный момент: если не использовать круглые скобки, а просто перечислить через запятую несколько элементов (и присвоить эту конструкцию переменной), получим кортеж.

Обращение к элементам кортежа выполняется так же, как и к элементам списка - с помощью индекса. Индекс указывается в квадратных скобках после имени кортежа. Индексация элементов кортежа начинается с нуля. Так же как и для списка, для кортежа можно получить срез.

На заметку

Еще раз напомним, что получение доступа к элементу кортежа или получение среза для кортежа означает, что мы можем "прочитать" соответствующие значения, но изменить их не можем.

Некоторые несложные примеры создания и использования кортежей приведены в листинге 4.9.

Листинг 4.9. Знакомство с кортежами

```
# Создаем пустой кортеж
a=tuple()
# Проверяем содержимое кортежа
print(a)
# Создаем кортеж на основе списка
b=tuple([10,20,30])
# Проверяем содержимое кортежа
print(b)
# Создаем кортеж на основе текста
c=tuple("Python")
# Проверяем содержимое кортежа
print(c)
# Объединение кортежей
a=b+(40,[1,2,3],60)
# Проверяем результат объединения кортежей
print(a)
# Получение среза
print(a[2:5])
```

При выполнении этого программного кода получаем следующий результат:

Результат выполнения программы (из листинга 4.9)

```
()
(10, 20, 30)
('P', 'y', 't', 'h', 'o', 'n')
(10, 20, 30, 40, [1, 2, 3], 60)
(30, 40, [1, 2, 3])
```

Например, командой `a=tuple()` создается пустой кортеж, и ссылка на него записывается в переменную a. Командой `b=tuple([10,20,30])` создаем кортеж из трех числовых элементов. Примером создания кортежа на основе текстового значения может быть команда `c=tuple("Python")`.

Команда `a=b+(40,[1,2,3],60)` иллюстрирует сразу несколько "моментов", связанных с использованием кортежей. Во-первых, здесь объединяются два кортежа. Один определяется переменной b, а второй

(40,[1,2,3],60) задан явно. Для объединения кортежей использован оператор сложения. Во-вторых, в кортеже (40,[1,2,3],60) один из элементов - список [1,2,3]. В-третьих, ссылка на результат операции b+(40,[1,2,3],60) записывается в переменную a, которая до этого ссылалась на пустой кортеж. Здесь нужно учесть, что в результате вычисления выражения b+(40,[1,2,3],60) создается новый кортеж, который получается последовательным объединением кортежей b и (40,[1,2,3],60). Именно на этот новый кортеж будет ссылаться переменная a. В последнем несложно убедиться, например, по результату получения среза для кортежа, на который ссылается переменная a. А именно, чтобы получить срез мы используем инструкцию a[2:5] - то есть действуем точно так же, как и в случае получения среза для списка.

На заметку

Обратите внимание: срезом кортежа является кортеж.

Методов у кортежей всего два. С помощью метода index() по значению элемента в кортеже можно узнать индекс этого элемента. Если элементов с таким значением в кортеже несколько, возвращается индекс первого элемента. С помощью метода count() выполняется подсчет количества элементов в кортеже с определенным значением (значение передается аргументом методу).

Узнать общее количество элементов в кортеже можно с помощью функции len(). Аргументом функции передается кортеж (или переменная, которая ссылается на кортеж). Оператор in позволяет определить, входит элемент в кортеж, или нет.

Резюме

Очень убедительно. Мы подумаем, к кому это применить.
из к/ф "31 июня"

1. Список - это упорядоченный набор элементов. Для создания списка элементы заключаются в квадратные скобки. Также можно использовать функцию list().
2. Элементы списка могут быть разного типа.
3. Определить количество элементов в списке можно с помощью функции len().

4. Для получения доступа к элементу списка после имени списка в квадратных скобках указывают индекс элемента. Индексация элементов начинается с нуля. Для индексации элементов с конца списка используют отрицательные индексы.

5. Для получения среза (доступа к нескольким элементам) в квадратных скобках после имени массива указывается, через двоеточие, два индекса, которые определяют последовательность элементов среза.

6. Значения элементов в списке можно менять. Элементы из списка можно удалять (например, методами `pop()` и `remove()`), добавлять в список (например, с помощью методов `append()` и `insert()`), а также заменять одну группу элементов на другую (скажем, через операцию присваивания значения срезу). Списки можно объединять (метод `extend()`).

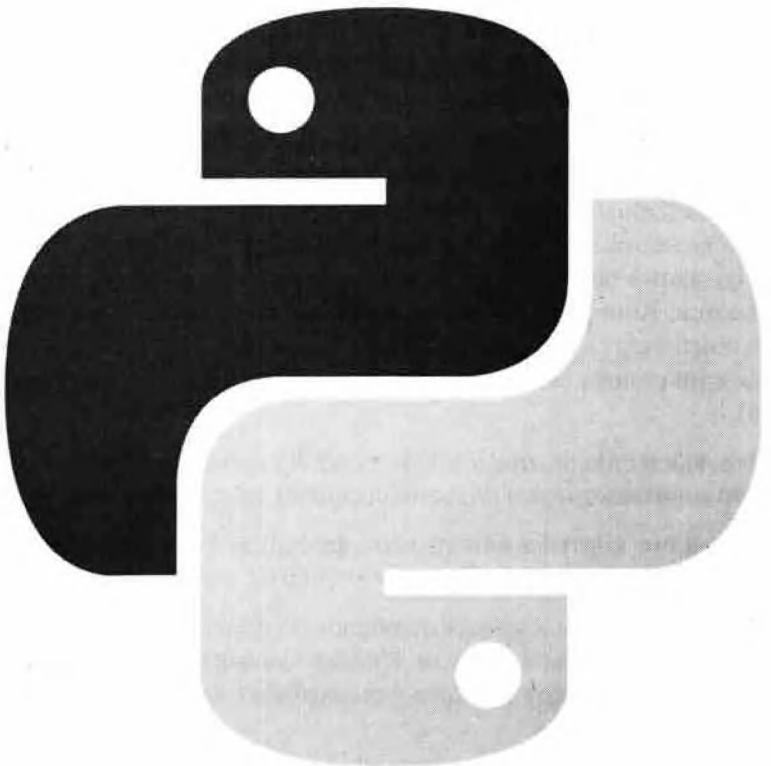
7. При копировании списков присваивание обычно не используется (поскольку при этом копия списка не создается, просто две переменные будут ссылаться на один список). Поверхностное копирование списков осуществляется через получение среза или с помощью метода `copy()`. При создании поверхностной копии для внутренних списков копии не создаются. Копирование списков можно выполнять с помощью функции `deepcopy()` (из модуля `copy`). В последнем случае создается полная копия списка (то есть создаются копии даже для внутренних списков).

8. Кортеж, как и список, представляет собой упорядоченный набор элементов. Во многом кортежи похожи на списки, но кортежи нельзя изменять.

9. Для создания кортежа используют функцию `tuple()` или в круглых скобках указывают через запятую элементы кортежа.

10. Доступ к элементам кортежа получают по индексу. Индексация элементов кортежа начинается с нуля. Индекс указывается в квадратных скобках после имени кортежа. Срез для кортежа получают так же, как и срез для списка.

11. Определить количество элементов в кортеже можно с помощью функции `len()`. Индекс элемента в кортеже определяется методом `index()`, а количество элементов с заданным значением можно определить с помощью метода `count()`.



Глава 5

Множества, словари и текст

Может где-нибудь высоко в горах, но не в нашем районе, вы что-нибудь обнаружите для вашей науки.
из к/ф "Кавказская пленница"

В этой главе мы продолжаем обсуждать те типы данных, которые можно было бы назвать *множественными*, поскольку их концепция близка к теории упорядоченных и неупорядоченных множеств. В предыдущей главе состоялось наше знакомство со списками и кортежами. Поэтому некоторое представление о том, каким же образом большие массивы данных могут реализоваться в программном коде в Python, читатель уже имеет. И хотя концепция списков позволяет просто и эффективно решать широкий класс задач, в некоторых случаях полезными могут оказаться и другие программные конструкции - такие, например, как множества и словари. О них будет идти речь в этой главе. Также мы более подробно обсудим текст.

Множества

Как, и здесь начинается то же самое?
из к/ф "Ирония судьбы или с легким паром"

Множество - это неупорядоченный набор уникальных элементов. В данном случае *неупорядоченный* означает, что порядок следования (включения) элементов во множестве значения не имеет. Что касается *уникальности*, то во множестве не может быть два одинаковых элемента (одинаковые элементы рассматриваются как один элемент). Фактически, если у нас есть некоторый элемент, то по отношению к тому или иному множеству мы можем лишь сказать, входит элемент во множество или нет. Говорить о позиции элемента внутри множества нет смысла.

🕮 На заметку

Условие уникальности элементов во множестве имеет последствия: элементами множества могут быть значения так называемых *неизменяемых типов* - например, числа или строки. Понятно, что при этом элементы одного множества могут относиться к разным типам данных (в пределах допустимых типов).

Для создания множества используют функцию `set()`, аргументом которой передается список со значениями, которые включаются во множество. Также можно сформировать множество с помощью пары фигурных скобок, внутри которых через запятую перечисляются элементы множества. Однако вариант с функцией `set()` все же является более предпочтительным, поскольку с помощью фигурных скобок задаются (правда, в несколько ином формате) не только множества, но и *словари* (словари описываются немного позже).

> **На заметку**
>
> В продолжение к вышесказанному можно добавить, что пустые фигурные скобки интерпретируются не как пустой список, а не как пустое множество. Чтобы создать пустое множество, используем инструкцию `set()`.

Небольшой пример создания множеств приведен в листинге 5.1.

Листинг 5.1. Создание множеств

```
# Исходный список
A=[1,30,"text",True,30,100,False]
# Отображаем содержимое списка
print("Список A:",A)
# На основе списка создается множество
B=set(A)
# Отображаем содержимое множества
print("Множество B:",B)
# Создаем еще одно множество
C={1,30,"text",True,30,100,False}
# Отображаем содержимое множества
print("Множество C:",C)
# Проверяем равенство множеств
print("Равенство множеств:",B==C)
# Вхождение элемента 1 в множество C
print("Элемент 1 в множестве C:",1 in C)
```

В результате выполнения программного кода получаем следующее:

Результат выполнения программы (из листинга 5.1)

```
Список A: [1, 30, 'text', True, 30, 100, False]
Множество B: {False, 1, 'text', 100, 30}
Множество C: {False, True, 'text', 100, 30}
Равенство множеств: True
Элемент 1 в множестве C: True
```

В данном случае сначала командой `A=[1,30,"text",True,30,100,False]` мы создаем список A с несколькими элементами разного типа. Для удобства список отображается в окне вывода (команда `print("Список A:",A)`). Затем на основе этого списка командой `B=set(A)` создается множество (используем функцию `set()`, аргументом которой передаем список A.). Содержимое множества B - а это элементы множества, - отображается в окне вывода командой `print("Множество B:",B)`. Что мы видим? Во-первых, в списке A были совпадающие числовые элементы со значением 30. Во множестве B элемент с таким значением остался всего один. Во-вторых, во множестве B, по сравнению со списком A, куда-то пропал элемент со значением `True`. Объяснение практически такое же, как и в предыдущем случае.

Дело в том, что в Python логические значения являются подмножеством множества целых чисел, поэтому `True` интерпретируется как 1. Элемент 1 в списке A тоже есть. Поэтому из двух элементов `True` и 1 "выживает" только один из них - в данном случае это элемент 1. В-третьих, в множестве B, по сравнению со списком A, порядок элементов поменялся. Но данное обстоятельство вообще не должно смущать, поскольку для множества порядок следования элементов значения не имеет. Точнее, такого понятия, как место элемента в множестве, вообще нет. Имеет значение только то, входит элемент в множество, или нет.

Командой `C={1,30,"text",True,30,100,False}` создаем еще одно множество. По идее, набор элементов, которые указаны через запятую в фигурных скобках такой же, как и набор элементов в списке A, на основе которого ранее создавалось множество B. Логично ожидать, что множества B и C будут совпадать (поскольку создавались на основе одинаковых наборов элементов). Но если с помощью команды `print("Множество C:",C)` отобразить содержимое множества C и сравнить с содержимым множества B, могут возникнуть сомнения в справедливости утверждения о равенстве множеств.

На заметку

Два множества полагают равными друг другу, если они состоят из одинаковых элементов. В частности, если известно, что два множества равны, то это означает, что каждый элемент из первого множества есть во втором множестве, а каждый элемент из второго множества также представлен и в первом множестве.

Более конкретно: во множестве B есть элемент 1, но нет элемента `True`, а во множестве C есть элемент `True`, но нет элемента 1. Однако если вспомнить, что логическое значение `True` на самом деле эквивалентно целочисленному значению 1, то проблема в принципе снимается. Все-же мы про-

веряем равенство множеств B и C, для чего используем команду `print("Равенство множеств:",B==C)`. Для сравнения множеств на предмет равенства мы использовали оператор ==. Результатом инструкции `B==C` является значение `True`, что бывает только в тех случаях, когда сравниваемые с помощью оператора == множества содержат одинаковые наборы элементов. Еще мы проверяем отдельно наличие элемента `1` во множестве `C` (напомним, что формально там числового элемента `1` нет, но есть логическое значение `True`).

Для проверки вхождения элемента в множество используют оператор `in`. Результатом команды вида `элемент in множество` будет `True`, если `элемент` входит в `множество`, и `False`, если такого `элемента` в `множестве` нет. В команде `print("Элемент 1 в множестве C:",1 in C)` нами использована инструкция `1 in C`, результатом которой является `True`. Несложно догадаться, что при проверке наличия `1` в множестве `C` как единичный числовой элемент интерпретируется логическое значение `True`.

Среди методов и функций, используемых при работе с множествами, многие нам уже знакомы. Так, количество элементов в множестве определяют с помощью функции `len()` (множество передается аргументом функции). Для создания копии множества используют метод `copy()` (метод без аргументов вызывается из копируемого множества).

На заметку

Множество может включать только элементы *неизменяемых* типов. Поэтому, например, список элементом множества быть не может. Не может быть элементом множества другое множество. Как следствие проблема создания поверхностных или полных копий, с которой мы столкнулись при обсуждении списков, для множеств неактуальна.

Вместе с тем, если попытаться создать копию множества с помощью оператора присваивания - то есть, просто присвоив одной переменной значение другой переменной (которая ссылается на множество), то получим две переменные, которые ссылаются на одно и то же множество.

Проверить, ссылаются ли переменные на один и тот же объект (в том числе таким объектом может быть множество) можно с помощью оператора `is`.

Для добавления элемента в множество из множества вызывают метод `add()`, аргументом которому передают добавляемый элемент. Обратная процедура, - то есть удаление элемента из множества, - выполняется с помощью методов `remove()` или `discard()`. Методы вызываются из множества, а аргументом передается элемент, который следует удалить из множества. Главное различие между методами в том, как они "реагируют" на си-

туацию, когда удаляемого элемента во множестве нет: при использовании метода `remove()` возникает ошибка, а метод `discard()` ошибки не вызывает - в этом случае просто ничего не происходит.

На заметку

Для полной очистки множества (удаления всех элементов из множества) можно использовать метод `clean()`.

Есть ряд операций (в основном математического характера), которые могут выполняться с множествами и для которых предусмотрены специальные методы и/или операторы.

На заметку

Здесь нелишним будет освежить в памяти некоторые понятия из теории множеств. Итак, допустим, что имеется два множества, которые обозначим как A и B. Объединением двух множеств A и B называется такое множество $A \cup B$, которое содержит в себе все элементы множества A и все элементы множества B. Например, если $A = \{1,2,3,4\}$ и $B = \{3,4,5,6\}$, то $A \cup B = \{1,2,3,4,5,6\}$. Результат получается объединением в одно множество элементов из множеств A и B. При этом "общие" элементы (то есть те, что есть и в множестве A, и в множестве B - а в данном случае это числа 3 и 4) включаются только один раз (то есть не дублируются). Причина в том, что по определению множество состоит из уникальных элементов. Два одинаковых элемента в множество входить не могут.

Пересечением множеств A и B называется такое множество $A \cap B$, которое состоит из элементов, входящих одновременно как в множество A, так и в множество B. Например, если $A = \{1,2,3,4\}$ и $B = \{3,4,5,6\}$, то $A \cap B = \{3,4\}$. Поскольку по определению, пересечение множеств состоит из общих элементов этих множеств, а в данном случае общими элементами для множеств A и B являются числа 3 и 4, то именно эти элементы формируют множество-результат.

Разницей множеств A и B называется множество $A \setminus B$, которое состоит из элементов множества A, которые при этом не являются элементами множества B. Например, если $A = \{1,2,3,4\}$ и $B = \{3,4,5,6\}$, то $A \setminus B = \{1,2\}$. Здесь результат формируется так: берем элементы из множества A, за исключением тех, что входят также и в множество B (это числа 3 и 4). Поэтому в итоге получаем множество, которое состоит из чисел 1 и 2.

Симметрической разницей множеств A и B называется такое множество $A \Delta B$, которое состоит из элементов множества A, не входящих в множество B, и из элементов множества B, не входящих в множество A. Другими словами, по определению симметрическая разность множеств A и B - это объединение множеств $A \cup B$, за исключением их общих элементов (пересечение множеств $A \cap B$): то есть имеет место соотношение $A \Delta B = (A \cup B) \setminus (A \cap B)$. Например, если $A = \{1,2,3,4\}$ и $B = \{3,4,5,6\}$, то $A \Delta B = \{1,2,5,6\}$. Поступаем так: берем все элементы из множеств A и B, за исключением тех из них, что одновременно входят как в множество A, так и в множество B (числа от 1 до 6 включительно). Общие элементы для двух множеств - числа 3 и 4. Поэтому результат - множество чисел 1, 2, 5 и 6.

Для вычисления объединения, пересечения и разности (в том числе и симметрической) множеств в Python предусмотрена целая группа специальных методов (и операторов). Рассмотрим их. Далее через A и B обозначены некоторые множества.

Для вычисления *объединения* множеств могут использоваться оператор | (оператор объединения множеств - если операнды являются множествами) или метод `union()`, который вызывается из первого множества, а аргументом ему передается второе множество. Соответствующие команды имеют вид `A|B` или `A.union(B)` (можно также воспользоваться командой `B.union(A)` - результат будет таким же). Сами множества A и B при этом не изменяются, а результатом является новое множество, состоящее из элементов множеств A и B. Если нам не нужно в результате объединения множеств создавать новое множество, а мы хотим вместо этого "расширить", например, множество A за счет элементов множества B, полезным будет метод `update()`. В результате выполнения команды `A.update(B)` создается объединение множеств A и B, а результат записывается в переменную A. Такого же эффекта можем добиться, скажем, с помощью команд `A=A|B` или `A|=B` (сокращенная форма оператора объединения множеств |). Примеры выполнения операции объединения множеств приведены в листинге 5.2.

Листинг 5.2. Объединение множеств

```
# Первое множество (множество A)
A={1,2,3,4}
# Отображаем содержимое множества
print("Множество A:",A)
# Второе множество (множество B)
B={3,4,5,6}
# Отображаем содержимое множества
print("Множество B:",B)
# Объединение множеств (множество C)
C=A|B
# Отображаем результат
print("Множество C=A|B:",C)
# Объединение множеств
print("Множество A.union(B):",A.union(B))
print("Множество B.union(A):",B.union(A))
# Изменяем множество A
A.update(B)
# Проверяем результат
print("Множество A:",A)
# Изменяем множество B
B=B|{-1,-2,-3}
# Проверяем результат
```

```
print("Множество B:",B)
# Изменяем множество C
C|={7,8,9}
# Проверяем результат
print("Множество C:",C)
```

Результат выполнения данного программного кода следующий:

Результат выполнения программы (из листинга 5.2)

```
Множество A: {1, 2, 3, 4}
Множество B: {3, 4, 5, 6}
Множество C=A|B: {1, 2, 3, 4, 5, 6}
Множество A.union(B): {1, 2, 3, 4, 5, 6}
Множество B.union(A): {1, 2, 3, 4, 5, 6}
Множество A: {1, 2, 3, 4, 5, 6}
Множество B: {3, 4, 5, 6, -2, -3, -1}
Множество C: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Для вычисления *пересечения* двух множеств можно использовать метод `intersection()` или оператор `&` (оператор пересечения множеств - если операнды являются множествами): формат команд выглядит как `A.intersection(B)` или `A&B`. Как и в случае объединения множеств, сами множества `A` и `B` не изменяются, а порядок указания множеств значения не имеет (команды `A.intersection(B)`, `B.intersection(A)`, `A&B` и `B&A` с точки зрения конечного результата эквивалентны).

Если мы хотим, чтобы при вычислении пересечения множеств вместо создания нового множества результат записывался в одно из исходных множеств, используют метод `intersection_update()`. Например, в результате выполнения команды `A.intersection_update(B)` вычисляется пересечение множеств `A` и `B` и ссылка на полученное множество записывается в переменную `A`. Такого же результата добиваемся посредством команд `A=A&B` или `A&=B` (сокращенная форма оператора пересечения множеств `&`). В листинге 5.3 представлены примеры вычисления пересечения множеств.

Листинг 5.3. Пересечение множеств

```
# Первое множество (множество A)
A={1,2,3,4}
# Отображаем содержимое множества
print("Множество A:",A)
# Второе множество (множество B)
B={3,4,5,6}
# Отображаем содержимое множества
```

```
print("Множество B:",B)
# Пересечение множеств (множество C)
C=A&B
# Отображаем результат
print("Множество C=A&B:",C)
# Пересечение множеств
print("Множество A.intersection(B):",A.intersection(B))
print("Множество B.intersection(A):",B.intersection(A))
# Изменяем множество A
A.intersection_update(B)
# Проверяем результат
print("Множество A:",A)
# Изменяем множество B
B=B&{4,6,8,10}
# Проверяем результат
print("Множество B:",B)
# Изменяем множество C
C&={1,2,3}
# Проверяем результат
print("Множество C:",C)
```

В результате выполнения представленного программного кода получаем следующее:

Результат выполнения программы (из листинга 5.3)

```
Множество A: {1, 2, 3, 4}
Множество B: {3, 4, 5, 6}
Множество C=A&B: {3, 4}
Множество A.intersection(B): {3, 4}
Множество B.intersection(A): {3, 4}
Множество A: {3, 4}
Множество B: {4, 6}
Множество C: {3}
```

Разность множеств вычисляется методом `difference()` или с помощью оператора - (оператор "минус" для операторов-множеств интерпретируется как оператор вычисления разности множеств). В результате вычисления выражений вида `A.difference(B)` и `A-B` возвращается новое множество, которое состоит из тех элементов множества `A`, которые не являются элементами множества `B` (очевидно, что в данном случае имеет значение порядок указания множеств). Для того чтобы результат вычисления разности множеств записывался в переменную `A`, используем команды `A.difference_update(B)` (здесь из множества `A` вызывается метод `difference_update()` с аргументом - множеством `B`), `A=A-B` или `A-=B`

(сокращенная форма оператора вычисления разности множеств). Примеры вычисления разности множеств представлены в листинге 5.4.

Листинг 5.4. Разность множеств

```
# Первое множество (множество A)
A={1,2,3,4}
# Отображаем содержимое множества
print("Множество A:",A)
# Второе множество (множество B)
B={3,4,5,6}
# Отображаем содержимое множества
print("Множество B:",B)
# Разность множеств (множество C)
C=A-B
# Отображаем результат
print("Множество C=A-B:",C)
# Разность множеств
print("Множество A.difference(B):",A.difference(B))
print("Множество B.difference(A):",B.difference(A))
# Изменяем множество A
A.difference_update(B)
# Проверяем результат
print("Множество A:",A)
# Изменяем множество B
B=B-{4,6,8,10}
# Проверяем результат
print("Множество B:",B)
# Изменяем множество C
C-={1,3,5}
# Проверяем результат
print("Множество C:",C)
```

Результат выполнения программного кода будет таким:

Результат выполнения программы (из листинга 5.4)

```
Множество A: {1, 2, 3, 4}
Множество B: {3, 4, 5, 6}
Множество C=A-B: {1, 2}
Множество A.difference(B): {1, 2}
Множество B.difference(A): {5, 6}
Множество A: {1, 2}
Множество B: {3, 5}
Множество C: {2}
```

Чтобы вычислить *симметрическую разность* множеств, используем метод `symmetric_difference()`. Вызывается метод из первого множества, второе множество передается аргументом, а результатом является новое множество. Множество, получающееся как значение выражения `A.symmetric_difference(B)` состоит из элементов множеств `A` и `B`, но только тех, которые принадлежат лишь одному из множеств. Такой же результат получаем с помощью инструкции `A^B`. Здесь мы используем оператор `^` (если операндами являются множества, то оператор вычисляет симметрическую разность).

Если воспользоваться командой `A=A^B` или `A^=B` (сокращенная форма вычисления симметрической разности), то будет вычислена симметрическая разность множеств `A` и `B`, а результат записан в переменную `A`. Этой же цели служит метод `symmetric_difference_update()`.

Примеры вычисления симметрической разности приведены в листинге 5.5.

Листинг 5.5. Симметрическая разность множеств

```
# Первое множество (множество A)
A={1,2,3,4}
# Отображаем содержимое множества
print("Множество A:",A)
# Второе множество (множество B)
B={3,4,5,6}
# Отображаем содержимое множества
print("Множество B:",B)
# Симметрическая разность множеств (множество C)
C=A^B
# Отображаем результат
print("Множество C=A^B:",C)
# Разность множеств
print("Множество A.symmetric_difference(B):",A.symmetric_
difference(B))
print("Множество B.symmetric_difference(A):",B.symmetric_
difference(A))
# Изменяем множество A
A.symmetric_difference_update(B)
# Проверяем результат
print("Множество A:",A)
# Изменяем множество B
B=B^{4,6,8,10}
# Проверяем результат
print("Множество B:",B)
# Изменяем множество C
```

```
C^={1,3,5}
# Проверяем результат
print("Множество C:",C)
```

После выполнения этого программного кода получаем такой результат:

Результат выполнения программы (из листинга 5.5)

```
Множество A: {1, 2, 3, 4}
Множество B: {3, 4, 5, 6}
Множество C=A^B: {1, 2, 5, 6}
Множество A.symmetric_difference(B): {1, 2, 5, 6}
Множество B.symmetric_difference(A): {1, 2, 5, 6}
Множество A: {1, 2, 5, 6}
Множество B: {3, 5, 8, 10}
Множество C: {2, 3, 6}
```

На заметку

Операторы `|`, `&` и `^`, которые выше упоминались соответственно при вычислении объединения, пересечения и симметрической разницы множеств, нам уже знакомы (во всяком случае, визуально) - до этого они были предъявлены миру в качестве побитовых операторов. Фактически многое зависит от операндов. Например, операторы `|`, `&` и `^` допустимо использовать для выполнения логических операций: оператором `|` вычисляется логическое или (аналог оператора `or`), оператором `&` вычисляется логическое и (аналог оператора `and`), а оператором `^` вычисляется логическое исключающее или (аналог оператора `xor`). Все это происходит, если операндами для операторов `|`, `&` и `^` указать логические значения `True` или `False`. Оправдание такому демократизму очевидное. Логические значения являются подмножеством целых чисел. Более конкретно, логическое значение `True` интерпретируется как `1`, а логическое значение `False` интерпретируется как `0`. Для этих целых чисел выполняются побитовые операции, результатом которых может быть опять же число `1` или `0`, которые интерпретируются как логические значения `True` и `False` соответственно.

Если "смотреть в корень", то операции с множествами, логические операции и побитовые операции на самом деле однотипные. Аналогию проиллюстрируем на примере логических операций. В этом случае под `True` следует понимать принадлежность элемента множеству, а под `False` - отсутствие элемента в множестве. Например, выражение вида `A|B` с логическими операндами `A` и `B` соответствует логическому или: результатом будет `True`, если хотя бы один из операндов равен `True`. Для множеств `A` и `B` выражение `A|B` - это объединение множеств. Результатом операции `A|B` является множество, состоящее из некоторых элементов. Возьмем теперь произвольный элемент. Он может принадлежать множеству `A|B` (аналог значение `True` для результата логического выражения), а может не принадлежать множеству `A|B` (аналог значение `False` для результата логического выражения). Элемент принадлежат множеству `A|B`, если он принадлежит хотя бы одному из множеств `A` (аналог значения `True` для операнда `A` в логическом выражении) или `B` (аналог значения `True` для операнда `B` в логическом выражении).

Для логического и значение выражения A&B с логическими операндами A и B равно True, только если оба операнда A и B равны True. Для множеств A и B результатом выражения A&B (пересечение множеств) является множество, состоящее из элементов, принадлежащих одновременно A и B (аналог значений True для операндов в логическом выражении).

Наконец, логические исключающее или для логических операндов A и B (выражение A^B) дает значение True если один и только один из операндов равен True. Симметрическая разница A^B для множеств A и B дает в результате множество, состоящее из элементов, принадлежащих одному и только одному из множеств A и B.

Помимо перечисленных операций с множествами, часто приходится иметь дело с операциями отношения (выполняемыми в контексте работы с множествами). Как сравнивать множества на предмет равенства и как проверять, входит ли элемент в состав множества, мы уже знаем. В первом случае полезным будет оператор ==, а во втором - оператор in.

На заметку

Два множества A и B считаются равными (обозначается как $A = B$), если каждый элемент множества A входит в множество B, а каждый элемент множества B входит в множество A.

Множество A является подмножеством множества B (обозначается как $A \subset B$ или $A \subseteq B$ если множества A и B могут совпадать), если каждый элемент множества A является элементом множества B.

В качестве операторов отношения используются стандартные операторы сравнения, а для некоторых есть еще дополнительные методы. Основные операторы и соответствующие им операции описаны в таблице 5.1. Через A и B обозначены некоторые множества, над которыми выполняются операции.

Таблица 5.1. Основные операции отношения для множеств

Оператор и/или метод	Описание
<	Результатом выражения A<B является True, если все элементы множества A входят в множество B (множество A является подмножеством множества B), причем оба множества не равны. В противном случае значение выражения равно False

Оператор и/или метод	Описание
<= или issubset()	Результатом выражения A<=B (или выражения A.issubset(B)) является True, если все элементы множества A входят в множество B (множество A является подмножеством множества B), причем допускается равенство множеств. В противном случае значение выражения равно False
>	Результатом выражения A>B является True, если все элементы множества B входят в множество A (множество B является подмножеством множества A), причем оба множества не равны. В противном случае значение выражения равно False
>= или issuperset()	Результатом выражения A>=B (или выражения A.issuperset(B)) является True, если все элементы множества B входят в множество A (множество B является подмножеством множества A), причем допускается равенство множеств. В противном случае значение выражения равно False
==	Результатом выражения A==B является True, если все элементы множества A входят в множество B, а все элементы множества B входят во множество A (равенство множеств). В противном случае значение выражения равно False
isdisjoint()	Результатом выражения A.isdisjoint(B) является значение True, если пересечение множеств A и B является пустым множеством (то есть если у множеств A и B нет общих элементов). В противном случае значение выражения равно False
in	Результатом выражения a in A является значение True, если элемент a входит в множество A. В противном случае значение выражения равно False

На заметку

Пустым называется множество, которое не содержит ни одного элемента (обычно обозначается как ∅).

Следует заметить, что множества могут использоваться в операторах цикла для перебора значений переменной цикла, как это было и со списками. Единственное, о чем следует помнить - что в отличие от списка, в множестве порядок элементов не фиксирован. Какое именно значение переменная цикла будет принимать на той или иной итерации, сказать крайне проблематично.

На заметку

Для создания множеств можно использовать *генераторы множеств*. Принцип тот же, что и в генераторах списков, только для множеств генератор заключается не в прямоугольные, а в фигурные скобки.

Как иллюстрацию к использованию множеств рассмотрим решение программными методами следующей задачи: необходимо определить, какие числа в диапазоне значений от 1 до 100 удовлетворяют следующим условиям:

- числа при делении на 5 дают в остатке 2 или 4;
- числа при делении на 7 дают в остатке 3;
- при делении чисел на 3 в остатке получается число, отличное от 1.

Это несложная задача и решаться она может различными способами. Самая простая и очевидная идея, которая приходит в голову - последовательно перебрать все натуральные числа от 1 до 100 и для каждого числа проверить выполнение перечисленных выше условий. Но нас, понятно, интересует не столько конечный результат, сколько методическая сторона вопроса. Поэтому для решения этой задачи в Python мы будем "отталкиваться" от теории множеств и возможностей языка Python.

На заметку

Чтобы понять алгоритм решения задачи, необходимо принять в расчет следующие обстоятельства.

Множество натуральных чисел от 1 до 100 обозначим как E. Множество чисел, которые при делении на 5 дают в остатке 2 обозначим как A_1. Множество чисел, которые при делении на 5 дают в остатке 4, обозначим как A_2. Числа, которые при делении на 7 дают в остатке 3, обозначим как множество B. Множество чисел, которые при делении на 3 дают в остатке 1, обозначим как C. Во всех перечисленных

случаях речь идет о числах, не превышающих 100. Поэтому все эти множества являются подмножествами множества E.

Множество чисел, которые при делении на 5 дают в остатке 2 или 4, обозначим как A. Очевидно, что $A = A_1 \cup A_2$. Множество чисел, которые при делении на 5 дают в остатке 2 или 4, а при делении на 7 дают в остатке 3, определяется выражением $A \cap B$, поскольку такие числа должны одновременно принадлежать как множеству A (остаток 2 или 4 при делении на 5), так и множеству B (остаток 3 при делении на 7). Если мы накладываем условие, чтобы такие числа еще и при делении на 3 не давали в остатке 1, то это означает, что числа не должны принадлежать множеству C (остаток 1 при делении на 3). Поэтому из множества $A \cap B$ следует "вычесть" множество C. Такие образом, множество $D = (A \cap B) \setminus C$ содержит нужные числа - то есть числа, удовлетворяющие всем перечисленным условиям.

Соответствующий программный код приведен в листинге 5.6.

Листинг 5.6. Использование множеств

```
# Верхняя граница
n=100
# Множество натуральных чисел
E={s for s in range(1,n+1)}
# Множество чисел, которые при делении
# на 5 дают в остатке 2
A1={s for s in E if s%5==2}
# Множество чисел, которые при делении
# на 5 дают в остатке 4
A2={s for s in E if s%5==4}
# Множество чисел, которые при делении
# на 5 дают в остатке 2 или 4
A=A1|A2
# Множество чисел, которые при делении
# на 7 дают в остатке 3
B={s for s in E if s%7==3}
# Множество чисел, которые при делении
# на 3 дают в остатке 1
C={s for s in E if s%3==1}
# Множество чисел, которые при делении
# на 5 дают в остатке 2 или 4, при делении на 7
# дают в остатке 3, а при делении на 3 не дают
# в остатке 1
D=(A&B)-C
# Отображаем результат
print("Приведенные ниже числа от 1 до",n)
print("1) при делении на 5 дают в остатке 2 или 4;")
print("2) при делении на 7 дают в остатке 3;")
print("3) при делении на 3 не дают в остатке 1:")
print(D)
```

В данном случае получаем следующий результат:

Результат выполнения программы (из листинга 5.6)

```
Приведенные ниже числа от 1 до 100
1) при делении на 5 дают в остатке 2 или 4;
2) при делении на 7 дают в остатке 3;
3) при делении на 3 не дают в остатке 1:
{24, 17, 59, 87}
```

В переменную n записывается значение для верхней границы числового диапазона. Множество натуральных чисел со значениями в диапазоне от 1 до n формируется командой `E={s for s in range(1,n+1)}`. Здесь мы использовали генератор множества: в фигурных скобках инструкция `s for s in range(1,n+1)` означает, что элемент множества определяется переменной s, которая пробегает значения натурального ряда от 1 до n.

Множество A1 создаем командой `A1={s for s in E if s%5==2}`. Теперь в генераторе множества переменная s перебирает значения элементов из множества E. Элемент заносится в множество A1 при условии `s%5==2`, то есть если остаток от деления s на 5 равняется 2. Аналогично командами `A2={s for s in E if s%5==4}`, `B={s for s in E if s%7==3}` и `C={s for s in E if s%3==1}` создаются прочие множества. Множество чисел, которые при делении на 5 дают в остатке 2 или 4, вычисляем путем объединения множеств A1 и A2 - с помощью команды `A=A1|A2`. Наконец, командой `D=(A&B)-C` находим нужный результат.

Кому-то, возможно, более удачным покажется иной подход, в котором множество-результат формируется сразу с помощью генератора множеств. Данный способ решения задачи представлен в листинге 5.7.

Листинг 5.7. Генерирование множества

```
# Верхняя граница
n=100
# Множество чисел, которые при делении
# на 5 дают в остатке 2 или 4, при делении на 7
# дают в остатке 3, а при делении на 3 не дают
# в остатке 1
D={s for s in range(1,n+1) if (s%5==2 or s%5==4) and s%7==3
and s%3!=1}
# Отображаем результат
print("Приведенные ниже числа от 1 до",n)
print("1) при делении на 5 дают в остатке 2 или 4;")
print("2) при делении на 7 дают в остатке 3;")
print("3) при делении на 3 не дают в остатке 1:")
```

```
print(D)
```

Результат выполнения этого программного кода такой же, как и в предыдущем случае:

Результат выполнения программы (из листинга 5.7)

```
Приведенные ниже числа от 1 до 100
1) при делении на 5 дают в остатке 2 или 4;
2) при делении на 7 дают в остатке 3;
3) при делении на 3 не дают в остатке 1:
{24, 17, 59, 87}
```

Здесь мы результат сформировали фактически одной командой `D={s for s in range(1,n+1) if (s%5==2 or s%5==4) and s%7==3 and s%3!=1}`, в которой в генераторе множеств переменная `s` пробегает значения натурального ряда от `1` до `n`. Но соответствующее число включается в создаваемое множество только если выполнено сложное условие `(s%5==2 or s%5==4) and s%7==3 and s%3!=1`. В этом условии с помощью логических операторов `or` (*логическое или*) и `and` (*логические и*) объединены несколько логических выражений: одновременно должны выполняться условия `(s%5==2 or s%5==4)`, `s%7==3` и `s%3!=1`, причем условие `s%5==2 or s%5==4` состоит в том, что выполняется хотя бы одно из условий `s%5==2` или `s%5==4`.

С другой стороны, можно воспользоваться и более "классическим" подходом. Пример приведен в листинге 5.8.

Листинг 5.8. Создание множества через цикл

```
# Верхняя граница
n=100
# Пустое множество
D=set()
# Формируется множество чисел, которые при делении
# на 5 дают в остатке 2 или 4, при делении на 7
# дают в остатке 3, а при делении на 3 не дают
# в остатке 1
for s in range(1,n+1):
    if s%5==2 or s%5==4:
        if s%7==3:
            if s%3!=1:
                D.add(s)
# Отображаем результат
print("Приведенные ниже числа от 1 до",n)
print("1) при делении на 5 дают в остатке 2 или 4;")
```

```
print("2) при делении на 7 дают в остатке 3;")
print("3) при делении на 3 не дают в остатке 1:")
print(D)
```

Результат получим такой:

Результат выполнения программы (из листинга 5.8)

```
Приведенные ниже числа от 1 до 100
1) при делении на 5 дают в остатке 2 или 4;
2) при делении на 7 дают в остатке 3;
3) при делении на 3 не дают в остатке 1:
{24, 17, 59, 87}
```

При таком подходе мы сначала создаем командой `D=set()` пустое множество, а затем с помощью оператора цикла и группы вложенных условных операторов добавляем в список новые элементы - разумеется, если эти элементы удовлетворяют всем критериям (условия в условных операторах). Для добавления элемента в множество задействован метод `add()`.

На заметку

Помимо типа `set` (множество), в языке Python есть тип `frozenset` (неизменяемое множество). Главное отличие неизменяемого множество от обычного множества состоит в том, что неизменяемое множество, как несложно догадаться, нельзя изменить. Создать неизменяемое множество можно с помощью функции `frozenset()`, аргументом которой передают, например, список или текстовую строку. Многие методы, применяемые к обычным множествам, могут применяться к неизменяемым множествам - речь, безусловно, идет о методах, не изменяющих структуру исходного множества.

Далее мы рассмотрим еще один полезный тип данных - а именно, речь пойдет о словарях.

Словари

Боже, какой типаж! Браво, браво! Слушайте, я не узнаю вас в гриме! Кто Вы такой?
из к/ф "Иван Васильевич меняет профессию"

В некотором смысле *словари* можно рассматривать как нечто среднее между списком и множеством. Словарь представляет собой неупорядоченный набор элементов. Однако в отличие от множества, каждый элемент словаря "идентифицирован" специальным образом. А именно, каждому элементу словаря сопоставляется *ключ*, который однозначно идентифицирует элемент в словаре. Ситуация немного похожа на индексацию элементов в спи-

ске. Но в списке индексы - целочисленные ранжированные значения. Ключи же в словаре не обязательно должны быть числами: помимо чисел, это могут быть, например, строки, списки или кортежи. То есть с некоторой натяжкой можно думать о словарях как о списках, но с нечисловыми (в общем случае) индексами.

Для создания словаря используют функцию `dict()`. Аргументом функции могут передаваться, разделенные запятыми, выражения вида `ключ=значение`. Также можно передать список, элементами которого являются списки (или кортежи) по два элементов в каждом: ключ и соответствующее ему значение.

На заметку

Если функции `dict()` при вызове не передать аргументы, будет создан пустой словарь. Если при вызове аргументом функции `dict()` передается уже существующий словарь, то в результате будет создана копия такого словаря (правда, поверхностная - то есть для таких вложенных элементов словаря, как списки и словари, выполняется копирование ссылок, но не копирование элементов).

Можно создать словарь, указав в фигурных скобках через запятые выражения вида `ключ:значение` (ключ и соответствующее ему значение указываются через двоеточие).

Обращение к элементу словаря выполняется формально так же, как и обращение к элементу списка: с помощью квадратных скобок после имени словаря, но только в случае словаря в квадратных скобках указывается не индекс, а ключ. Другими словами, если мы имеем дело со словарем, то узнать, каково значение элемента с указанным ключом можем инструкцией вида `словарь[ключ]`. Причем такую инструкцию используют не только для того, чтобы прочитать значение элемента, но и чтобы изменить это значение. Небольшие примеры создания словарей приведены в листинге 5.9.

Листинг 5.9. Создание словарей

```
# Список для формирования словаря
A=[["Пушкин А.С.","Капитанская дочка"],["Чехов А.П.","Вишневый
сад"],["Толстой Л.Н.","Война и мир"]]
# Создаем словарь на основе списка
writers=dict(A)
# Отображаем содержимое словаря
print("Словарь:")
print(writers)
# Обращение к элементу словаря по ключу
print("Чехов написал пьесу:",writers["Чехов А.П."])
# Изменяем значение элемента словаря
```

```
writers["Чехов А.П."]="Каштанка"
# Проверяем содержимое словаря
print("Словарь после изменения элемента:")
print(writers)
# Добавляем в словарь новый элемент
writers["Достоевский Ф.М."]="Преступление и наказание"
# Проверяем содержимое словаря
print("Словарь после добавления элемента:")
print(writers)
print()
# Перебор элементов словаря по ключу
print("Авторы и их произведения.")
for s in writers.keys():
    print("Автор:",s)
    print("Произведение:",writers[s])
    print()
# Создаем новый словарь
lights=dict(красный="движение запрещено",желтый="всем
внимание",зеленый="движение разрешено")
# Проверяем содержимое словаря
print("Новый словарь:")
print(lights)
# Значение ключа
color="зеленый"
# Обращение к элементу словаря по ключу
print("Если горит",color,"свет, то",lights[color]+"!")
print()
# Создаем еще один словарь
girls={(90,60,90):"Света",(85,65,89):"Юля",(92,58,91):"Нина"}
# Проверяем содержимое словаря
print("Еще один словарь:")
print(girls)
# Значение ключа
params=(90,60,90)
# Обращение к элементу словаря по ключу
print(girls[params]+":",params)
```

При выполнении программного кода получаем такой результат:

Результат выполнения программы (из листинга 5.9)

```
Словарь:
{'Чехов А.П.': 'Вишневый сад', 'Пушкин А.С.': 'Капитанская
дочка', 'Толстой Л.Н.': 'Война и мир'}
Чехов написал пьесу: Вишневый сад
Словарь после изменения элемента:
```

```
{'Чехов А.П.': 'Каштанка', 'Пушкин А.С.': 'Капитанская дочка',
'Толстой Л.Н.': 'Война и мир'}
Словарь после добавления элемента:
{'Чехов А.П.': 'Каштанка', 'Достоевский Ф.М.': 'Преступление
и наказание', 'Пушкин А.С.': 'Капитанская дочка', 'Толстой
Л.Н.': 'Война и мир'}

Авторы и их произведения.
Автор: Чехов А.П.
Произведение: Каштанка

Автор: Достоевский Ф.М.
Произведение: Преступление и наказание

Автор: Пушкин А.С.
Произведение: Капитанская дочка

Автор: Толстой Л.Н.
Произведение: Война и мир

Новый словарь:
{'зеленый': 'движение разрешено', 'желтый': 'всем внимание',
'красный': 'движение запрещено'}
Если горит зеленый свет, то движение разрешено!

Еще один словарь:
{(92, 58, 91): 'Нина', (85, 65, 89): 'Юля', (90, 60, 90):
'Света'}
Света: (90, 60, 90)
```

В этом программном коде иллюстрируется несколько способов создания словарей, а также демонстрируются некоторые несложные приемы работы со словарями. Сначала командой `A=[["Пушкин А.С.","Капитанская дочка"],["Чехов А.П.","Вишневый сад"],["Толстой Л.Н.","Война и мир"]]` создается список `A`. Список `A` состоит из трех элементов. Каждый элемент сам является списком. В каждом внутреннем списке по два элемента. Первый элемент играет роль ключа при создании словаря, а второй элемент - непосредственно элемент словаря. В данном случае все значения текстовые, поэтому и ключи словаря, и непосредственно элементы словаря будут текстом.

Для создания словаря список `A` передаем аргументом функции `dict()` (команда `writers=dict(A)`). Чтобы оценить содержимое словаря `writers` передаем имя словаря аргументом функции `print()` (команда `print(writers)`).

На заметку

Как несложно заметить из содержимого в окне вывода, словарь отображается в таком формате: в фигурных скобках через запятую отображаются пары значений ключа и соответствующего ему элемента словаря. Ключ и значение элемента разделяются двоеточием. Порядок отображения таких пар в фигурных скобках произвольный. Если вдуматься, то становятся очевидными причины такого "либерализма": значение элемента словаря однозначно определяется по ключу и не зависит от "места" (или позиции) элемента в словаре. Более того, такое понятие, как позиция элемента в словаре не имеет смысла.

Как отмечалось ранее, обращение к элементу словаря выполняется по ключу. Например, инструкция `writers["Чехов А.П."]` представляет собой обращение к элементу словаря `writers` с ключом `"Чехов А.П."` (элемент со значением `"Вишневый сад"`).

Командой `writers["Чехов А.П."]="Каштанка"` мы изменяем значение элемента словаря с ключом `"Чехов А.П."`. Если после этого проверить содержимое словаря `writers` (что, собственно, и делается), то можно заметить изменения в содержимом словаря.

Если в команде присваивания значения элементу словаря указать ключ, который в словаре не представлен, то в словарь будет добавлен элемент с соответствующим ключом. Например, в результате выполнения команды `writers["Достоевский Ф.М."]="Преступление и наказание"` в словарь `writers` добавляется элемент `"Преступление и наказание"` с ключом `"Достоевский Ф.М."`.

Для перебора элементов словаря по ключу может использоваться оператор цикла. Доступ к ключам словаря получают с помощью метода `keys()`. Как результат метод возвращает итерируемый (то есть допускающий процесс "перебора" содержимого) объект с ключами словаря (того словаря, из которого вызывается метод). На результат вызова метода `keys()` можно смотреть как на "контейнер", в котором "содержатся" ключи словаря. В операторе цикла этот контейнер указывается так же, как если бы это было множество, состоящее из ключей словаря. Скажем, если мы имеем дело с оператором цикла, в котором начальная инструкция имеет вид `for s in writers.keys()`, то это означает, что переменная `s` будет последовательно принимать значения ключей из словаря `writers` (правда неизвестно, в какой именно очередности будут перебираться ключи). В теле оператора цикла командой `print("Автор:",s)` отображается значение ключа, а командой `print("Произведение:",writers[s])` отображается значение элемента с соответствующим ключом. Для создания пустой строки в окне вывода используем команду `print()`.

Новый словарь создается командой `lights=dict(красный="движени е запрещено",желтый="всем внимание",зеленый="движение ра зрешено")`. Аргументами функции `dict()` передаются выражения вида `ключ=значение`. И ключи, и значения, как и в предыдущем случае, текстовые.

На заметку

Обратите внимание, что текстовые значения ключей в данном конкретном случае указываются без кавычек.

Содержимое словаря проверяется командой `print(lights)`. Командой `color="зеленый"` в переменную `color` записывается значение ключа, после чего эта переменная используется в команде `print("Если горит ",color,"свет, то",lights[color]+"!")` для отображения значения ключа и значения соответствующего ключу элемента словаря `lights`.

Наконец, командой `girls={(90,60,90):"Света",(85,65,89):"Юля ",(92,58,91):"Нина"}` создается еще один словарь. Здесь две особенности: во-первых, элементы словаря вместе с ключами (разделяются двоеточием) указываются в фигурных скобках. Во-вторых, ключами словаря в данном случае служат кортежи. В этом нет ничего необычного: кортеж вполне легитимен в качестве ключа словаря. Как иллюстрация к сказанному - команда `print(girls[params]+":",params)` где ключом для элемента словаря `girls` указана переменная `params`, которой предварительно командой `params=(90,60,90)` присвоен кортеж.

На заметку

Если при обращении к элементу словаря указать неправильный ключ (ключ, которого в словаре нет), возникнет ошибка. Есть метод `get()`, который позволяет по ключу получить значение элемента словаря. При этом если указан несуществующий ключ (аргумент метода), ошибки не будет (методом возвращается "пустое" значение `None`). Методу `get()` можно передать второй аргумент - данное значение будет возвращаться методом при неверном ключе.

Помимо получения доступа к элементу словаря по ключу, существуют и другие интересные операции, которые можно выполнять со словарями. Так, мы уже знаем, что с помощью метода `keys()` получают доступ к ключам словаря. Доступ к значениям словаря получают с помощью метода `values()`.

На заметку

Оба метода возвращают в качестве результата объекты. Для метода `keys()`

это объект класса dict_keys. Метод values() возвращает объект класса dict_values. В этих объектах "спрятаны" соответственно ключи и значения элементов словаря. Мы сами объекты обсуждать не планируем. Для нас лишь важно, что эти объекты допускают итерации. То есть содержимое объектов (ключи и значения элементов) можно последовательно "перебирать", хотя делается и не так "прямолинейно", как можно было бы подумать. Поэтому об объектах, возвращаемых методами dict_keys и values() с некоторой натяжкой можно думать как о некоторой виртуальной последовательности значений. Эту виртуальную последовательность можно преобразовать к более простому и понятному формату: списку, кортежу или множеству - в зависимости от конкретных потребностей и специфики решаемой задачи. Для этого результат метода dict_keys или values() передается аргументом методу list() (создание списка), tuple() (создание кортежа) или set() (создание множества).

Есть также метод items(), который возвращает объект класса dict_items, содержащий кортежи с парами значений для ключей и элементов. Приведенные выше замечания относятся и к методу items().

Удалить элемент из словаря можно с помощью метода pop(). Аргументом методу передается ключ удаляемого элемента. Метод возвращает значение - это значение того элемента, который удаляется из словаря. Если нужно удалить элемент словаря "по тихому", без возвращения результата, - используют инструкцию del, после которой указывают ссылку на удаляемый из словаря элемент (имя словаря и в квадратных скобках после имени - ключ элемента). Полностью очистить словарь можно с помощью метода clear().

Как добавляется элемент в словарь, мы уже видели в рассмотренном выше примере: элементу с новым ключом просто присваивается значение. Если нужно добавить сразу несколько элементов, то удобнее воспользоваться методом update(). Метод не возвращает результат и вызывается из того словаря, в который нужно добавить новые элементы. Аргументом методу может передаваться словарь, элементы которого добавляются в исходный словарь (из которого вызван метод). Также аргументом метода update() может быть список, через который реализуется "добавляемый" словарь или набор разделенных запятыми выражений вида ключ=значение. Некоторые примеры использования перечисленных выше методов для работы со словарями представлены в листинге 5.10.

Листинг 5.10. Работа со словарями

```
# Создаем словарь
symbs=dict([["a","первый"],["b","второй"]])
# Создаем еще один словарь
more_symbs=dict([["c","третий"],["d","четвертый"]])
# Добавляем второй словарь в первый
symbs.update(more_symbs)
```

```
# Содержимое словаря
print("Словарь:",symbs)
# Длина словаря
print("Количество ключей в словаре:",len(symbs))
# Доступ к элементу по ключу (ключ в словаре есть)
print("Элемент с ключом 'c':",symbs.get("c","нет такого
ключа!"))
# Проверяем наличие ключа в словаре
print("Наличие элемента с ключом 'c':","c" in symbs)
# Удаляем элемент из словаря
del symbs["c"]
# Содержимое словаря
print("Словарь:",symbs)
# Доступ к элементу по ключу (ключа в словаре нет)
print("Элемент с ключом 'c':",symbs.get("c","нет такого
ключа!"))
# Проверяем наличие ключа в словаре
print("Наличие элемента с ключом 'c':","c" in symbs)
# Список ключей словаря
print("Ключи словаря:",list(symbs.keys()))
# Список значений элементов словаря
print("Значения элементов словаря:",list(symbs.values()))
# Содержимое словаря
print("Содержимое словаря:",list(symbs.items()))
# Удаление элемента из словаря
print("Удаляется элемент со значением:",symbs.pop("b"))
# Содержимое словаря
print("Словарь:",symbs)
# Очистка словаря
symbs.clear()
# Содержимое словаря
print("Словарь:",symbs)
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 5.10)

```
Словарь: {'d': 'четвертый', 'c': 'третий', 'b': 'второй', 'a':
'первый'}
Количество ключей в словаре: 4
Элемент с ключом 'c': третий
Наличие элемента с ключом 'c': True
Словарь: {'d': 'четвертый', 'b': 'второй', 'a': 'первый'}
Элемент с ключом 'c': нет такого ключа!
Наличие элемента с ключом 'c': False
Ключи словаря: ['d', 'b', 'a']
Значения элементов словаря: ['четвертый', 'второй', 'первый']
```

```
Содержимое словаря: [('d', 'четвертый'), ('b', 'второй'),
('a', 'первый')]
Удаляется элемент со значением: второй
Словарь: {'d': 'четвертый', 'a': 'первый'}
Словарь: {}
```

В данном случае мы создаем два словаря `symbs=dict([["a","первый"],["b","второй"]])` и `more_symbs=dict([["c","третий"],["d","четвертый"]])`, после чего командой `symbs.update(more_symbs)` расширяем словарь `symbs` за счет словаря `more_symbs`. При этом словарь `symbs` меняется, а словарь `more_symbs` остается таким, как и прежде.

Длину словаря (количество ключей в словаре) определяем с помощью функции `len()`. Обращение к элементу словаря выполняется с помощью метода `get()`. Инструкция `symbs.get("c","нет такого ключа!")` возвращает в качестве результата значение элемента словаря `symbs` с ключом `"c"` если такой ключ есть, и значение `"нет такого ключа!"`, если ключа `"c"` в словаре `symbs` нет. Для проверки вхождения ключа `"c"` в словарь `symbs` использована инструкция `"c" in symbs`: значение этого выражения равно `True` если ключ `"c"` есть в словаре `symbs`, и равняется `False` в противном случае.

Чтобы удалить элемент с ключом `"c"` из словаря `symbs` используем команду `del symbs["c"]`. При удалении элемента словаря командой `symbs.pop("b")` не только удаляется элемент с ключом `"b"`, но еще и возвращается значение удаляемого элемента. Наконец, командой `symbs.clear()` выполняется полная очистка словаря: из него удаляются все элементы.

🕮 На заметку

Для создания копии словаря используют метод `copy()` и функцию `deepcopy()` из модуля `copy`. Методом `copy()` создается поверхностная копия словаря. Функцией `deepcopy()` создается полная копия словаря. То есть в данном случае наблюдается аналогия со списками.

Так же, как для списков, для словарей существуют генераторы. Если сравнивать *генератор словарей* с генератором списков, то есть некоторые отличия (в принципе, их два):

- В отличие от генератора списка, который заключается в прямоугольные скобки, генератор словаря заключается в фигурные скобки.
- При создании словаря с помощью генератора за каждую итерацию (цикл) приходится "генерировать" два параметра: ключ и элемент. Ключ и элемент разделяются двоеточием. Например, если у нас есть два списка `clr=["красный","желтый","зеленый"]` и `txt=["c`

тоим","ждем","движемся"], то командой A={clr[i]:txt[i] for i in range(len(clr))} создается словарь, в котором элементы первого списка clr играют роль ключей, а соответствующие им элементы второго списка txt - собственно элементов словаря A.

Далее рассмотрим некоторые особенности реализации текстовых значений в Python.

Текстовые строки

Если мы допустим беспорядок в документации, потомки нам этого не простят.
из к/ф "Гостья из будущего"

В наших программных кодах мы постоянно используем текстовые значения, хотя при этом особо серьезного внимания такому типу данных, как текст, не уделяли. Настало время устранить эту несправедливость.

О тексте (или текстовых строках) можно говорить и писать очень много - благо объект исследования дает к этому все основания. Чтобы не отвлекаться на второстепенные вопросы, мы выделим основные темы, которые представляют первоочередной интерес. На них и сконцентрируемся.

Нас будут интересовать:

- создание текстовых строк;
- основные операции со строками;
- использование в строках специальных символов;
- форматирование текстовых строк.

Но даже в рамках этих тем мы будем ограничиваться только наиболее интересными и показательными случаями.

Текстовые строки мы уже создавали: текстовое значение заключалось нами в двойные кавычки. На самом деле это не единственный способ создать текстовый литерал. С таким же успехом можно заключать текст в одинарные кавычки. В какие кавычки заключать текст (одинарные или двойные) - разницы нет. И так, и так - результат один и тот же. То есть если мы имеем дело с обычной текстовой строкой, не содержащей каких-то специальных символов, то выбор типа кавычек - вопрос сугубо эстетических предпочтений программиста. Однако если мы предполагаем использовать в тексте одинарные или двойные кавычки, то обычно используют следующий прием:

- если в тексте есть двойные кавычки, то весь текст заключается в одинарные кавычки;

- если в тексте есть одинарные кавычки, то весь текст заключается в двойные кавычки.

Эти правила удобные, но не обязательные. Например, мы можем в тексте, выделенном двумя кавычками, использовать двойные кавычки - но в этом случае двойным кавычкам в тексте должна предшествовать косая черта (слеш \). Это же замечание относится и к случаю, когда в тексте, выделенном одинарными кавычками, нужно использовать апостроф (одинарную кавычку).

На заметку

Если в текстовом значении нужно поместить символ \ не как часть некоторой специальной инструкции, а в качестве буквы (то есть если мы хотим, чтобы в тексте отображался символ \), то на самом деле нужно использовать две косые черты, то есть \\. Другими словами, комбинация \\ в тексте отображается как одинарная косая черта.

Будет полезной косая черта и в том случае, если мы захотим создать текстовый литерал (текстовое значение), занимающий несколько строк кода. Символ \ в конце строки кода внутри текстового литерала означает перенос строки.

На заметку

Символ переноса \ должен быть последним в строке команды - то есть внутри текстового литерала в строке с символом \ справа от него никаких других символов (букв) быть не должно. Также важно понимать, что в данном случае речь идет о размещении текстового литерала в нескольких строках в окне редактора кодов. При отображении такого текстового литерала переход к новой строке в месте размещения символа \ не осуществляется. Инструкцией перехода к новой строке является \n. Другими словами, если необходимо, чтобы при отображении текста в определенном месте этого текста выполнялся переход к новой строке, в этом месте вставляем инструкцию \n.

Можно поступить еще более радикально: заключить текстовый литерал в тройные двойные кавычки (то тесть в начале текста три раза двойные кавычки, и три раза двойные кавычки - в конце текста) или тройные одинарные кавычки (три одинарные кавычки в начале текста, и три одинарные кавычки в конце текста). Преимущество такого подхода состоит в том, что внутри соответствующего текстового значения можно свободно использовать одинарные и двойные кавычки, а также выполнять разбивку литерала на несколько строк. Причем при отображении литерала разбивка на строки остается такой, какой она была в окне редактора кодов. Небольшие примеры объявления текстовых значений приведены в листинге 5.11.

Листинг 5.11. Создание текста

```
txt="Знание языка 'Python' - залог успеха."
print(txt)
txt='Знание языка "Python" - залог успеха.'
print(txt)
txt="Знание языка \"Python\" - залог успеха."
print(txt)
txt='Знание языка \'Python\' - залог успеха.'
print(txt)
txt="Знание языка 'Python'\
 - залог успеха."
print(txt)
txt="Знание языка 'Python'\n - залог успеха."
print(txt)
txt="""Знание языка
          "Python"
       - залог успеха."""
print(txt)
```

Результат выполнения этого программного кода следующий:

Результат выполнения программы (из листинга 5.11)

```
Знание языка 'Python' - залог успеха.
Знание языка "Python" - залог успеха.
Знание языка "Python" - залог успеха.
Знание языка 'Python' - залог успеха.
Знание языка 'Python' - залог успеха.
Знание языка 'Python'
 - залог успеха.
Знание языка
          "Python"
       - залог успеха.
```

Хочется верить, что особых комментариев и код, и результат его выполнения, не требуют.

Поскольку текстовая строка представляет собой упорядоченный набор символов, то нет ничего удивительного в том, что к строкам применимы такие операции, как обращение к элементу строки (букве) по индексу и получение среза. Как и в случае других упорядоченных множественных типов данных, таких как кортежи и списки, индексация элементов (букв) в строке начинается с нуля.

Узнать количество букв (символов) в строке можно с помощью функции `len()`. Конкатенация (объединение) строк выполняется, как мы уже знаем, с помощью оператора сложения +.

На заметку

Помимо процедуры объединения строк обычным "сложением" (то есть с помощью оператора +), в Python существует так называемая неявная конкатенация строк. В этом случае строки просто указываются рядом одна с другой, без всякого оператора между ними. Например, так: `txt="Мы изучаем " "Python"`. То есть мы разместили рядом (через пробел) два литерала: `"Мы изучаем "` и `"Python"`. В результате переменная `txt` получает значение `"Мы изучаем Python"`.

Примеры обращения к символам текстовой строки через индекс, получения среза, а также конкатенации строк приведены в листинге 5.12.

Листинг 5.12. Текстовые строки

```
# Неявная конкатенация строк
txt="Мы изучаем " "Python"
print(txt)
print("Всего",len(txt),"букв")
# Использован символ табуляции \t
print("Индекс\tБуква")
# Отображаются индексы и буквы
for i in range(len(txt)):
    # Приведение (с помощью функции str())
    # целочисленного типа к текстовому,
    # обращение к букве по индексу
    print(str(i)+":  \t"+txt[i])
print(txt[11:])
# С помощью среза строка отображается
# в обратном порядке
print(txt[::-1])
# Текстовая строка
word="Java"
# Явная конкатенация строк и присваивание
# переменной txt нового значения
txt=txt[:3]+"не"+txt[2:11]+word
print(txt)
```

Ниже приведен результат выполнения этого программного кода:

Результат выполнения программы (из листинга 5.12)

```
Мы изучаем Python
Всего 17 букв
Индекс  Буква
0:   М
1:   ы
2:
3:   и
```

```
4:   з
5:   у
6:   ч
7:   а
8:   е
9:   м
10:
11:     P
12:     y
13:     t
14:     h
15:     o
16:     n
Python
nohtyP меачузи ыМ
Мы не изучаем Java
```

В приведенном программном коде, хотя он прост и очевиден, стоит обратить внимание на некоторые моменты. Так, исходная текстовая строка `txt` создается неявной конкатенацией текстовых литералов (два текстовых литерала подряд без оператора между ними). Для определения количества букв в текстовой строке `txt` используем инструкцию `len(txt)`. В текстовых литералах несколько раз используется специальный символ `\t` - это символ табуляции.

На заметку

Специальные символы, такие как `\n` (перевод строки) или `\t` (табуляция), не отображаются, а имеют некий "тайный смысл". Хотя формально символов два (в инструкции `\n` это `\` и `n`), рассматриваются такие специальные символы как один. Вставляются они прямо в текстовую строку.

В операторе цикла обращение к букве в текстовой строке `txt` выполняется в формате `txt[i]` (где `i` - индексная переменная). При этом при вычислении текстового выражения `str(i)+":  \t"+txt[i]`, представляющего собой конкатенацию трех текстовых фрагментов, для перевода целочисленного значения индексной переменной `i` в текстовый формат использована функция `str()`.

Также программный код содержит примеры выполнения среза для текстовой строки. Так, инструкцией `txt[11:]` возвращается часть текстовой строки `txt`, начиная с `12`-й буквы (этой букве соответствует индекс `11`, так как индексация начинается с нуля) и до конца текста. Выражение `txt[::-1]` - это текстовая строка `txt`, записанная в обратном порядке (поскольку шаг приращения по индексу для получения среза указан отрицательный).

В команде `txt=txt[:3]+"не"+txt[2:11]+word` при вычислении правой части выполняется конкатенация таких текстовых строк:

- срез `txt[:3]` (в тексте `txt` буквы от начала и до индекса 2 включительно);
- текст `"не"`;
- срез `txt[2:11]` (буквы в тексте `txt` от индекса 2 до индекса 10);
- текстовое значение переменной `word` (значение `"Java"`).

Полученный текст в качестве нового значение присваивается переменной `txt`.

На заметку

Текст (тип данных `str`) относится к неизменяемым типам данных, поэтому изменить текстовое значение нельзя. То есть мы не можем в текстовом значении взять и изменить, например, какую-то букву. Но мы можем на основе текста сформировать новое текстовое значение, а потом переменной, которая ссылается на исходный текст, присвоить новое значение. Благодаря тому, что в Python переменные ссылаются на значения (а не содержат их), такой прием возможен и часто используется на практике. Причем внешне создается иллюзия, как будто реально меняется значение переменной. На самом деле ссылка (содержащаяся в переменной и связывающая эту переменную с данными) перебрасывается с одного значения на другое. Именно таким приемом мы воспользовались выше.

Что касается методов (и функций) для работы с текстом, то их очень много и диапазон их назначения чрезвычайно широк. Например, есть несколько методов, предназначенных для управления регистром символов в текстовой строке:

- метод `upper()` позволяет получить строку, в которой все буквы прописные (большие буквы или верхний регистр);
- метод `lower()` позволяет получить строку, в которой все буквы строчные (маленькие буквы или нижний регистр);
- метод `capitalize()` позволяет получить строку, в которой первая буква прописная;
- метод `title()` позволяет получить строку, в которой первая буква каждого слова - прописная;
- метод `swapcase()` позволяет получить строку, в которой все строчные буквы заменены на прописные, а прописные - на строчные.

У всех этих методов аргументов нет. Методы вызываются из текстового объекта (и текстовой переменной). Исходную текстовую строку они не ме-

няют, а на основе этой строки формируют новую, которая и возвращается в качестве результата метода. Примеры использования методов приведены в листинге 5.13.

Листинг 5.13. Регистр символов

```
txt="мы изучаем язык PYTHON"
print(txt.upper())
print(txt.lower())
print(txt.capitalize())
print(txt.title())
print(txt.swapcase())
print(txt)
```

Результат выполнения данного программного кода такой:

Результат выполнения программы (из листинга 5.13)

```
МЫ ИЗУЧАЕМ ЯЗЫК PYTHON
мы изучаем язык python
Мы изучаем язык python
Мы Изучаем Язык Python
МЫ ИЗУЧАЕМ ЯЗЫК python
мы изучаем язык PYTHON
```

Важный класс задач связан с поиском символов или текстовых фрагментов в строке. Например:

- Методы `find()` и `index()` используются для поиска подстроки в строке. Каждый из методов вызывается из объекта строки (переменной, ссылающейся на строку), а аргументом передается подстрока, поиск которой выполняется в строке. Результатом является индекс позиции, начиная с которой подстрока входит в строку. Если строка содержит подстроку в нескольких местах, возвращается индекс первого вхождения. Если в строке нет подстроки, то метод `find()` возвращает значение `-1`, а метод `index()` генерирует ошибку (класс `ValueError`). Методам можно также передать второй и третий числовые аргументы. В этом случае поиск подстроки осуществляется в диапазоне индексов, определяемых этими аргументами. Аналогичным образом используются и методы `rfind()` и `rindex()`. Базовое отличие от методов `find()` и `index()` в том, что теперь выполняется поиск последнего вхождения подстроки в строку.
- С помощью метода `count()` можно подсчитать, сколько раз подстрока (аргумент метода) входит в строку (объект, из которого вызывается метод). Поиск может выполняться не по всей строке, а

только по фрагменту строки - второй и третий числовые аргументы метода (если они есть) определяют диапазон поиска.

- Методы `startswith()` и `endswith()` позволяют проверить соответственно, начинается ли и заканчивается ли строка (объект, из которого вызывается метод) подстрокой (аргумент метода).
- Метод `replace()` используется для выполнения замены текстовых фрагментов в текстовой строке. Метод вызывается из текстовой переменной, а аргументами ему передаются две текстовые подстроки. Методом в качестве результата возвращается текстовая строка, которая получается заменой в исходном тексте (объект, из которого вызывается метод) подстроки - первого аргумента метода, на подстроку - второй аргумент метода. Количество замен можно ограничить, передав методу третий числовой аргумент (максимальное количество замен).

Небольшие примеры использования некоторых из перечисленных выше методов приведены в листинге 5.14.

Листинг 5.14. Поиск и замена символов

```
txt="""И.В. Гете. "Фауст" (отрывок):
Бессодержательную речь
Всегда легко в слова облечь.
Из голых слов, ярясь и споря,
Возводят здания теорий.
Словами вера лишь жива.
Как можно отрицать слова?"""
word="слов"
print(txt,end='\n\n')
print("Подстрока встречается",txt.count(word),"раза")
print("Первая позиция:",txt.index(word))
print("Следующая позиция:",txt.find(word,69))
print("Последняя позиция:",txt.rindex(word))
print("В начале инициалы:",txt.startswith("И.В."))
print("В конце знак вопроса:",txt.endswith("?"),end='\n\n')
print(txt.replace(" ","_"))
```

Выполнение программного кода приводит к такому результату:

Результат выполнения программы (из листинга 5.14)

```
И.В. Гете. "Фауст" (отрывок):
Бессодержательную речь
Всегда легко в слова облечь.
Из голых слов, ярясь и споря,
```

```
Возводят здания теорий.
Словами вера лишь жива.
Как можно отрицать слова?

Подстрока встречается 3 раза
Первая позиция: 68
Следующая позиция: 91
Последняя позиция: 179
В начале инициалы: True
В конце знак вопроса: True

И.В._Гете._"Фауст"_(отрывок):
Бессодержательную_речь
Всегда_легко_в_слова_облечь.
Из_голых_слов,_ярясь_и_споря,
Возводят_здания_теорий.
Словами_вера_лишь_жива.
Как_можно_отрицать_слова?
```

На заметку

В некоторых командах с вызовом функции `print()` среди аргументов есть инструкции `end='\n\n'`. В этом случае при выводе текста в консоль в конце дважды выполняется переход к новой строке. Поэтому после выполнения такой команды в окне вывода появляется пустая строка.

Есть группа методов, которые позволяют выполнить проверку содержимого текстовой переменной. Скажем, нас может интересовать вопрос, в каком регистре находятся буквы в тексте, или есть ли в тексте цифры. На многие такие нетривиальные вопросы реально получить исчерпывающие ответы с помощью группы специальных методов. Название каждого метода начинается со слова *is*. Каждый из методов вызывается (без аргументов) из текстовой строки и возвращает значение `True` если проверяемое условие (свойство текста) истинно, и `False` - если оно ложно. Например:

- методом `isdigit()` в качестве значения возвращается `True`, если текст состоит только из цифр;
- метод `isalpha()` возвращает значение `True`, если текст состоит только из букв;
- метод `isalnum()` возвращает значение `True`, если текст не содержит ничего, кроме букв и цифр;
- метод `islower()` возвращает значение `True`, если текст состоит только из строчных (маленьких) букв;

- метод `isupper()` возвращает значение `True`, если текст состоит только из прописных (больших) букв;
- метод `istitle()` возвращает значение `True`, если каждое слово в тексте начинается с большой буквы.

Примеры использования этих методов собраны в листинге 5.15.

Листинг 5.15. Проверка текстовых значений

```
print("123".isdigit(),"12.3".isdigit())
print("abc".isalpha(),"abc123".isalpha())
print("ab12".isalnum(),"ab12\n".isalnum())
print("ABC".isupper(),"aBc".isupper())
print("abc".islower(),"aBc".islower())
print("Ab12 Ab12".istitle(),"Ab12 AB12".istitle())
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 5.15)

```
True False
True False
True False
True False
True False
True False
```

Есть и другие полезные методы, часто незаменимые при работе с текстовыми значениями. Они кратко перечислены ниже:

- Методы `strip()`, `lstrip()` и `rstrip()` используются для удаления определенных символов из текстовой строки - соответственно в начале и конце строки, только в начале строки и только в конце строки. Удаляемые символы (в виде текстовой строки) передаются аргументом методу. Если аргумент не указать, по умолчанию удаляются пробелы.
- Для разделения строки на подстроки используют метод `split()` (или `rsplit()`). Разделитель (в виде текста), который служит индикатором разбивки на строки, передается аргументом методу. Результатом метод возвращает список из подстрок, на которые разбивается исходная строка. Методом `split()` поиск подстроки-разделителя выполняется слева направо, а методом `rsplit()` - справа налево. Если разбивка на подстроки выполняется по символу перехода к новой строке `\n`, можно использовать метод `splitlines()`. Метод `partition()` выполняет похожую

процедуру. В строке, из которой вызывается метод, находится первое вхождение подстроки, переданной аргументом методу. То есть аргумент метода - это подстрока-разделитель. И исходная строка как бы разбивается на три части: то, что до подстроки-разделителя, подстрока-разделитель и то, что после подстроки-разделителя. Вся эти три фрагмента возвращаются в виде кортежа. В случае если подстроки-разделителя в исходной строке нет, кортеж (результат метода) будет содержать первым элементом исходную строку, а два других элемента - пустой текст. Аналогично используется метод `rpartition()`, но только поиск подстроки-разделителя выполняется с конца исходной строки.

- Метод `join()` позволяет создавать текстовые строки путем объединения текстовых подстрок, реализованных в виде списка. Список с объединяемыми текстовыми фрагментами передается аргументом методу. Между фрагментами можно добавлять тестовый разделитель - это тот текст, из которого вызывается метод `join()`.

Примеры использования некоторых методов приведены в листинге 5.16.

Листинг 5.16. Некоторые операции со строками

```
txt="_*_ABC_*_abc_*_"
print(txt.lstrip("_*_"))
print(txt.rstrip("_*_"))
print(txt.strip("_*_"))
print(txt.split("*"))
print(txt.rsplit("*"))
print(txt.partition("*"))
print(txt.rpartition("*"))
print("abc \n ABC \n ***".splitlines())
print("_*_".join(["AAA","BBB","CCC"]))
```

Ниже приведен результат выполнения этого кода:

Результат выполнения программы (из листинга 5.16)

```
ABC_*_abc_*_
_*_ABC_*_abc
ABC_*_abc
['_', '_ABC_', '_abc_', '_']
['_', '_ABC_', '_abc_', '_']
('_', '*', '_ABC_*_abc_*_')
('_*_ABC_*_abc_', '*', '_')
['abc ', ' ABC ', ' ***']
AAA_*_BBB_*_CCC
```

На заметку

Иногда приходится иметь дело с кодами символов. Функция `chr()` позволяет по коду символа восстановить сам символ: аргументом функции передается код, а результатом является символ. Обратная процедура (определение кода для символа) выполняется с помощью функции `ord()`: аргументом функции передается символ, а результатом функции является код этого символа.

Еще один важный вопрос, который нельзя обойти вниманием - это форматирование текстовых строк. Речь идет о явном определении способа и формы отображения текстовых значений. В Python эта задача может решаться по-разному. Например, методы `center()`, `ljust()` и `rjust()` позволяют производить выравнивание текстовой строки (соответственно - по центру, по левому краю и по правому краю) внутри поля заданной ширины. Ширина поля передается аргументом методу. Примеры использования этих методов приведены в листинге 5.17.

Листинг 5.17. Выравнивание текста

```
txt="ABCDEFGH"
print("*"+txt.center(20)+"*")
print("*"+txt.rjust(20)+"*")
print("*"+txt.ljust(20)+"*")
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 5.17)

```
*       ABCDEFGH       *
*              ABCDEFGH*
*ABCDEFGH              *
```

В данном случае текст выравнивается внутри поля шириной 20 символов, а для удобства в начале и конце текстовых значений добавлены символы `*`.

Достаточно интересный метод `format()`. Метод вызывается из текстовой строки, у него есть аргументы и в качестве результата метод возвращает текстовое значение. Если абстрагироваться от деталей, то общая схема такая: берем некоторую текстовую строку, выполняем с ней определенные манипуляции, и в результате получаем новую строку. За "манипуляции" как раз и "отвечает" метод `format()`.

Манипуляции выполняются над тем текстом, из которого вызывается метод. А характер манипуляций определяется аргументами метода `format()`. Текстовая строка, из которой вызывается метод `format()`, может содержать специальные символы форматирования - то есть встроенные в текст инструкции, которые имеют особый "смысл" для метода `format()` и обра-

батываются им. Поэтому обычно текстовую строку, из которой вызывается метод `format()`, называют *строкой формата*. Мы будем придерживаться именно такой терминологии.

Инструкции форматирования в строке формата заключаются в фигурные скобки. В наиболее простом варианте строка форматирования представляет собой текстовый шаблон, в определенные места которого вставляются текстовые параметры. Места для вставки текстовых параметров определяются инструкциями форматирования, а сами параметры - это аргументы метода `format()`.

Целочисленные значения в фигурных скобках соответствуют индексам аргументов метода `format()`. Индексация аргументов метода начинается с нуля (первому аргументу соответствует индекс ноль). Например, результатом выражения `"Один - это {0}, а два - это {1}.".format("one","two")` является текст `"Один - это one, а два - это two."`. Инструкция `{0}` в строке формата означает, что в этом месте должен быть вставлен первый аргумент метода `format()` (текст `"one"`). Инструкция `{1}` определяет место вставки второго аргумента метода `format()` (текст `"two"`).

Инструкции формата могут быть более замысловатыми, чем число в фигурных скобках, и содержать больше информации, чем просто индекс аргумент для вставки в текстовый шаблон. Через двоеточие после индекса аргумента можно указать минимальную ширину поля (целое число), которая выделяется для этого аргумента, а также способ выравнивания текста в этом поле. Способ выравнивания определяется специальными символами: < означает выравнивание по левому краю, > означает выравнивание по правому краю, а ^ означает выравнивание по центру. Например, инструкция `{0:>20}` означает, что в соответствующем месте нужно вставить первый (по порядку) аргумент метода `format()`, под этот аргумент нужно выделить поле шириной не меньше 20 символов, а вставляемый текст должен выравниваться по правому краю.

На заметку

Числовой параметр после двоеточия определяет минимальную ширину выделяемого поля. Если вставляемый текст занимает больше места, то данный параметр игнорируется.

По умолчанию используется выравнивание текста по левому краю.

Небольшой пример использования метода `format()` приведен в листинге 5.18.

Листинг 5.18. Строка формата

```
txt="{0}жды {0} - будет {1}"
print(txt.format("два","четыре"))
print(txt.format("три","девять"))
print("Текст '{0}': {0:<20}.".format("abcdef"))
print("Текст '{0}': {0:^20}.".format("abcdef"))
print("Текст '{0}': {0:>20}.".format("abcdef"))
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 5.18)

```
дважды два - будет четыре
трижды три - будет девять
Текст 'abcdef': abcdef              .
Текст 'abcdef':        abcdef       .
Текст 'abcdef':               abcdef.
```

Существуют и другие приемы выполнения форматирования, в том числе и с помощью метода `format()`. В случае необходимости читатель может обратиться к специальной справочной литературе или встроенной справке среды разработки Python.

Резюме

Вот ведь, на всех языках говоришь, а по-русски не понимаешь.
из к/ф "Покровские ворота"

1. Множество - это неупорядоченный набор уникальных элементов. Создается множество с помощью функции `set()` (аргументом передается список с элементами множества) или путем перечисления элементов в фигурных скобках.
2. Для проверки того, входит ли элемент в множество, используют оператор `in`.
3. Количество элементов в множестве определяют с помощью функции `len()`.
4. Для создания копии множества используется метод `copy()`.
5. Для добавления элемента в множество используют метод `add()`, для удаления - метод `remove()`. Для добавления в множество элементов другого множества используют метод `update()`.

6. Объединение множеств - множество, элементы которого состоят из элементов объединяемых множеств. Для объединения множеств используют метод `union()` или оператор `|`.

7. Пересечение множеств - множество, элементы которого состоят из элементов, общих для обоих множеств. Для вычисления пересечения множеств используют метод `intersection()` или оператор `&`.

8. Разность множеств - множество, элементы которого состоят из элементов первого множества, за вычетом тех элементов, которые входят во второе множество. Для вычисления разности множеств используют метод `difference()` или оператор `-`.

9. Симметрическая разность множеств - множество, элементы которого состоят из элементов первого и второго множеств, за вычетом тех элементов, которые входят в оба множества одновременно. Для вычисления симметрической разности множеств используют метод `symmetric_difference()` или оператор `^`.

10. Для сравнения множеств используют операторы == (равенство множеств), < (первое множество есть подмножество второго множества), <= (первое множество является подмножеством второго множества или множества равны), > (второе множество есть подмножество первого множества), >= (второе множество является подмножеством первого множества или множества равны).

11. Генератор множества подобен генератору списка, но вся конструкция заключается в фигурные скобки.

12. Словарь представляет собой неупорядоченный набор элементов. Доступ к элементам словаря выполняется по ключу.

13. Для создания словаря используют функцию `dict()`, аргументом которой передают список с элементами-списками. Каждый такой внутренний список состоит из двух элементов: значения ключа и значения элемента. Можно также создать словарь, заключив в фигурные скобки, разделенные двоеточием, ключи и значения соответствующих элементов.

14. Доступ к элементам словаря выполняется по ключу: после имени словаря в квадратных скобках указывается ключ элемента.

15. Метод `keys()` позволяет получить доступ к ключам словаря. Для доступа к значениям словаря используют метод `values()`. Метод `items()` возвращает кортежи с ключами и значениями элементов словаря.

16. Для удаления элемента из словаря применяют метод `pop()`. Для добавления нескольких элементов используют метод `update()`. Новый эле-

мент в словарь можно добавить, присвоив значение элементу с новым ключом: после имени словаря в квадратных скобках указывается ключ добавляемого элемента, и, после знака равенства, значение элемента.

17. Поверхностная копия словаря создается с помощью метода `copy()`, а полную копию можно создать с помощью функции `deepcopy()` из модуля `copy`.

18. Генератор словаря заключается в фигурные скобки, и одновременно нужно создавать два параметра: ключ и соответствующий ему элемент словаря.

19. Текстовая строка - упорядоченный набор символов. Текстовые литералы заключаются в одинарные или двойные кавычки.

20. Для добавления апострофа, двойных кавычек, обратной косой черты перед этими символами указывается обратная косая черта `\`. Обратная косая черта используется и в ряде специальных символов, таких как табуляция `\t` или инструкция перехода к новой строке `\n`.

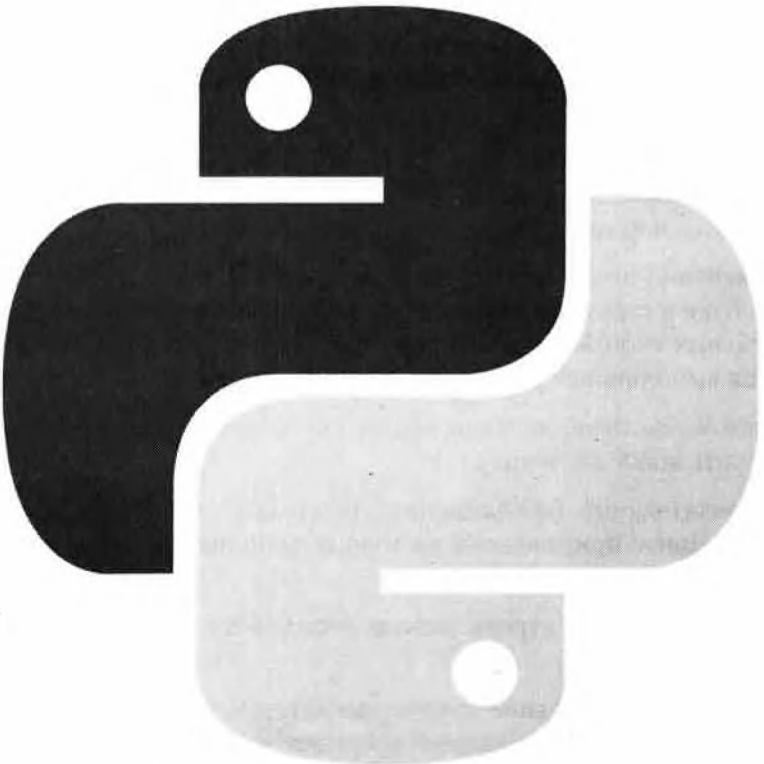
21. К элементам (символам) строки можно обращаться по индексу (индексация букв в строке начинается с нуля): после текстовой переменной в квадратных скобках указывается индекс буквы в строке. Также допускается выполнение срезов для текстовой строки.

22. Изменить текстовое значение нельзя (но можно текстовой переменной присвоить новое значение).

23. Для конкатенации (объединения) текстовых строк используют оператор +. Явное приведение к тестовому типу выполняется с помощью функции `str()`.

24. Количество букв в строке можно определить с помощью функции `len()`.

25. Существует значительное количество методов, которые позволяют выполнять самые разнообразные операции с текстовыми значениями, включая (но не ограничиваясь) следующее: преобразование регистра символов, поиск символов и подстрок, замена текстовых фрагментов, разбивка строк на подстроки, проверка содержимого текстовой строки, форматирование текстовой строки для вывода в консоль, и многое другое.



Глава 6

Основы объектно-ориентированного программирования

История, леденящая кровь. Под маской овцы скрывался лев!
из к/ф "Покровские ворота"

В этой главе мы познакомимся с основами объектно-ориентированного программирования (сокращенно ООП), а точнее, с тем, как принципы ООП реализуются в языке Python. В основе всей концепции ООП, без преувеличения, находятся такие понятия, как *класс* и *объект* (в отношении Python это *класс* и *экземпляр класса*). Поэтому мы начнем со знакомства именно с этими понятиями или их аналогами (хотя, безусловно, только ими дело не ограничится).

Классы, объекты и экземпляры классов

На то лед, чтоб скользить.
из к/ф "Покровские ворота"

Прежде, чем поближе познакомиться с классами и объектами, имеет смысл подчеркнуть несколько принципиальных моментов, которые в некотором смысле подготовят читателя к тому подходу, который будет использован в этой главе для объяснения фундаментальных понятий и принципов ООП. Для нас важно вот что: речь будет идти не просто о классах и объектах, а о том, как концепция классов и объектов реализуется в языке Python. Почему это важно? Важно потому, что сама по себе тема ООП и, более конкретно, классов и объектов, обычно достаточно сложна для понимания даже для тех, кто имеет опыт программирования. А в случае с языком Python проблемы, скорее всего, возникнут не только у новичков, но и у читателей, знакомых с методами ООП на примере таких языков, как C++, Java или C#.

На заметку

Понятно, что выше изложено только предположение. Надеемся, что для читателя все, что будет обсуждаться далее, окажется понятным или даже очевидным. Тем не менее, статистика и практика преподавания говорят об обратном. Мы тоже будем исходить из суровых реалий ООП. При этом приложим все усилия, чтобы изучение азов (и не только азов) ООП было наиболее удобным и наименее трудоемким.

Все дело в том, что механизмы и принципы реализации ООП в Python значительно отличаются от тех, что использованы в C++, Java и C#. Принципиальные различия существуют даже на уровне терминологии. Поэтому перед нами стоит двойная задача: с одной стороны, необходимо в доступной форме изложить принципы ООП для тех, кто с ними не знаком, а с другой стороны, важно вернуть "на путь истинный" тех, кто изучал ООП в контексте других языков программирования.

На заметку

Для тех, кто знаком с другими объектно-ориентированными языками: в Python класс сам является объектом. Это интригующее обстоятельство имеет весьма далеко идущие последствия. Более того, как мы уже знаем, переменные в Python не объявляются, а вводятся в программу путем присваивания значения. Это же правило остается справедливым при работе с классами и объектами. Отсюда получается, что процедура объявления полей, стандартная для многих языков программирования, в Python просто теряет смысл. Аналогично, многие привычные (по языкам программирования C++, Java и C#) в ООП моменты окажутся чуждыми для языка Python. Короче говоря, в экзотике недостатка не будет.

Главная идея ООП состоит в том, чтобы объединить в одно целое данные (то есть то, что хранится в переменных) и функции, предназначенные для обработки этих данных. Реализуется эта идея в классе. По большому счету *класс* - это некая конструкция, которая связывает или объединяет определенное количество переменных и определенное количество функций. На основе класса создаются *объекты*. Вообще, про класс удобно думать как про некоторый шаблон, на основе которого затем "по образу и подобию" создаются объекты. Но важно понимать, что здесь речь идет не о создании клонов. То есть класс и объект, созданный на основе этого класса - совершенно разные вещи (или сущности). Мы воспользуемся аналогией. Допустим, нам нужно сделать (собрать) автомобиль. Завод у нас есть, и мы можем произвести любую деталь. Но этого мало. Нам нужен некий план или чертеж, который бы давал нам четкое и однозначное представление, какие детали в автомобиле как и куда крепятся. Понятно, что на самом деле там очень много чертежей для разных блоков и механизмов. Но в данном случае это не важно. Мы можем думать, что чертеж один. Вот этот чертеж, на основе которого собирается автомобиль - это аналог класса. А автомобиль, который собирается в соответствии с чертежом - аналог объекта, созданного на основе класса.

Какую роль играет чертеж? В нем "прописано", какие детали есть у автомобиля и что это автомобиль "умеет делать": сколько дверей, сколько фар и как они расположены, размер колес, наличие или отсутствие кондиционера, радиоприемника, парковочного устройства, и многое другое. Все авто-

мобили, собранные по этому чертежу, будут однотипными в смысле набора опций и функциональных возможностей. Если мы возьмем другой чертеж и соберем автомобиль по этому чертежу, то, очевидно, что получим автомобиль с несколько иными характеристиками (все зависит от чертежа - что в нем заложено, то и получим).

То есть ситуация следующая:

- Есть чертеж автомобиля. Это аналог класса.
- На основе чертежа можно собирать автомобили. Автомобиль, собранный на основе чертежа - аналог объекта, который создан на основе класса.

Но это еще не все. Чертеж ведь должен быть "записан" на каком-то "носителе". Сейчас, конечно, "носителями" являются компьютеры, но нам удобнее думать, что чертеж выполнен на листе бумаги. Лист бумаги с чертежом - это тоже объект, - но другой, не такой, как автомобиль. То есть на самом деле, есть такой объект, как чертеж. На основе этого объекта можно создать другой объект - автомобиль. Здесь "спрятано" важное обстоятельство, специфичное именно для языка Python: в этом языке объектом является не только то, что мы создаем на основе класса, но и сам класс является объектом.

Точно так же, как бумажный лист с чертежом является объектом, отличным от автомобиля, объект, которым является класс, отличается от объекта, который создается на основе класса. И здесь мы подходим к очень важному месту: к *терминологии*. Тот объект, который создается на основе класса, мы будем называть *экземпляром класса*. Тот объект, через который реализуется класс, будем называть *объектом класса*.

На заметку

Для фанатов C++, Java и C# такой терминологический подход может показаться произволом (дело в том, что в этих языках объектом класса обычно называют тот объект, что создается на основе класса). Но мы так поступаем по необходимости и следуем той традиции именования классов и объектов, которая "исторически" сложилась среди разработчиков языка Python и программистов, использующих этот язык.

Итак, далее мы будем именовать объекты, которые создаются на основе класса *экземплярами класса* или просто *экземплярами*. Также в некоторых случаях используют термин *объект-экземпляр*. То, что класс в Python на самом деле является объектом - обстоятельство важное, но на данном этапе мы на этом зацикливаться не будем. Хотя впоследствии и вспомним о такой его особенности. Сейчас сконцентрируемся на двух вопросах:

- Как создать (описать) класс?
- Как на основе класса создать экземпляр класса?

Ответы на эти вопросы одновременно простые и сложные. Простые они, если смотреть на все это с формальной точки зрения. А сложными они становятся, как только мы начинаем "копать вглубь".

Итак, шаблон описания класса имеет следующий вид (жирным шрифтом выделены ключевые инструкции):

```
class имя_класса:
    # тело класса
```

Начинается все с ключевого слова `class`, после которого указывается *имя класса* и двоеточие. Затем описывается тело класса (напоминаем, что при описании инструкций в теле класса необходимо делать отступы - рекомендуется использовать четыре пробела). Что же пишут в теле класса? В теле класса, как правило, описывают *методы*. Метод - это та же функция, только вызываться она будет из экземпляра класса. Мы с методами уже имели дело неоднократно. Но только раньше мы использовали готовые методы, а теперь нам предстоит все это организовать своими руками.

Прежде, чем рассмотреть конкретный пример, отметим одно важное обстоятельство. Формально метод в теле класса описывается как обычная функция. Но методы, как мы уже знаем, должны вызываться из экземпляра класса. При описании метода экземпляр класса, из которого будет вызываться метод, должен быть явно указан как первый аргумент метода. При вызове метода этот аргумент методу явно не передается. Причина в том, что экземпляр класса, из которого вызывается метод, указывается явно (перед именем метода через точку). Получается такое своеобразное правило, которое условно можно назвать "минус один аргумент": при вызове метода из экземпляра класса у него на один аргумент меньше, чем это было при описании метода (это если нет аргументов со значением по умолчанию).

На заметку

Другими словами, при описании метода в теле класса у него должен быть, по крайней мере, один аргумент. Первый аргумент в списке аргументов метода обозначает тот экземпляр класса, из которого вызывается (или если точнее, будет вызываться) метод. Когда метод вызывается, то самый первый из аргументов ему не передается (точнее, в круглых скобках после имени метода не указывается) - в качестве первого аргумента автоматически используется ссылка на экземпляр класса, из которого вызван метод.

Ну и, разумеется, из того, что у метода есть аргумент, еще не следует, что этот аргумент обязательно должен быть задействован в программном коде метода.

Существует соглашение называть при описании метода первый его аргумент self. Мы тоже будем придерживаться этого соглашения - то есть в тех местах программного кода, где у метода объявлен аргумент с названием self, этот аргумент будет означать ссылку на экземпляр класса, из которого вызывается метод.

После того, как класс создан, возникает следующий вопрос: как на основе класса создать экземпляр класса? Делается это очень просто с помощью команды вида переменная=класс(). Другими словами, после имени класса указываем круглые скобки (пока что пустые) и всю эту конструкцию присваиваем в качестве значения некоторой переменной. Как результат будет создан экземпляр класса, а ссылка на этот экземпляр записана в переменную.

На заметку

Команда создания экземпляра класса может быть более замысловатой. Но эту ситуацию мы обсудим после того, как познакомимся с конструкторами.

Метод, описанный в классе как метод экземпляр класса (а мы пока других вариантов не знаем), вызывается из экземпляра класса с помощью "точечного синтаксиса": после экземпляра класса через точку указывается имя вызываемого метода. При этом методу в круглых скобках передаются нужные аргументы. Если аргументы методу передавать не нужно, пустые круглые скобки после имени метода все равно указываем. Небольшой пример создания класса и экземпляра класса (с последующим вызовом из экземпляра класса метода) приведен в листинге 6.1.

Листинг 6.1. Класс и экземпляры класса

```
# Создаем класс с названием MyClass
class MyClass:
    # Метод экземпляра класса.
    # Единственный аргумент метода self - ссылка
    # на экземпляр класса, из которого вызывается
    # метод
    def say_hello(self):
        # Методом отображается сообщение. Аргумент
        # метода (ссылка self) явно не используется
        print("Вас приветствует экземпляр класса!")
# Создаем экземпляр класса
obj=MyClass()
# Вызываем метод экземпляра класса.
# При вызове аргументы методу не передаются
obj.say_hello()
```

В результате выполнения данного программного кода в окне вывода появляются такое сообщение:

Результат выполнения программы (из листинга 6.1)

```
Вас приветствует экземпляр класса!
```

В данном случае мы создаем класс с названием `MyClass`. В теле класса описан всего один метод, который называется `say_hello()`, и у этого метода, как несложно заметить, один аргумент, который называется `self`. В теле метода имеется одна команда `print("Вас приветствует экземпляр класса!")`, в результате выполнения которой, как мы не без оснований ожидаем, в окне вывода появится текстовое сообщение.

На заметку

Хотя метод `say_hello()` объявлен с аргументом `self`, этот аргумент в теле метода на самом деле не используется. Здесь нет ничего "незаконного". Хорошего, правда, тоже мало. Комментарии - далее.

Экземпляр класса создается командой `obj=MyClass()`. В данном случае экземпляр класса создается непосредственно при выполнении инструкции `MyClass()`, а ссылка на этот экземпляр записывается в переменную `obj`. Но мы, если это не будет вызывать недоразумений, здесь и далее будем называть экземпляром класса переменную, которая на самом деле всего лишь ссылается на экземпляр класса.

Метод `say_hello()` из экземпляра класса (переменная `obj`) вызывается командой `obj.say_hello()`. В принципе, такого рода команды, когда метод вызывается из экземпляра класса, мы уже использовали во множестве. Обращает на себя внимание два обстоятельства.

Во-первых, хотя метод `say_hello()` описывался в классе с одним аргументом, вызывается он без передачи аргументов. То есть получаем правило "минус один аргумент" в действии. Во-вторых, результат выполнения метода `say_hello()`, в силу того, как он определен, не зависит от экземпляра класса, из которого вызывается метод. Если бы мы на основе класса `MyClass` создали еще один экземпляр класса и вызвали из него метод `say_hello()`, результат был бы точно таким же, как в рассмотренном выше примере.

Хотя ничего неправильного или некорректного в этом нет, такой подход граничит с дурным тоном. Почему? Как минимум потому, что если нас устраивает ситуация, когда все экземпляры класса "ведут" себя одинаково, то возникают сомнения в необходимости использования ООП для решения соот-

ветствующей задачи. Проще говоря, здесь имеет место не совсем адекватное использование объектно-ориентированного подхода. Поэтому обычно методы, которые описываются как методы экземпляра класса, в том или ином виде используют ссылку на экземпляр класса, из которого вызывается метод (первый аргумент с рекомендованным названием `self`).

Прежде, чем приступить к рассмотрению примера, в котором нам удастся избежать описанной выше досадной неприятности, расширим познания относительно содержимого экземпляров класса. Новость очень простая: у экземпляров класса могут быть не только методы, но и переменные, которые мы будем называть *полями экземпляра класса*. А все вместе, - поля и методы экземпляра класса, - будем называть *атрибутами экземпляра класса*. То есть атрибуты - это поля и методы.

На заметку

Ситуация с терминологией не очень однозначная. Термин *поле* скорее относится к таким языкам, как C++, Java или C#. В справочных ресурсах по языку Python то, что мы будем называть полем, обычно называют несколько иначе: *атрибуты-данные*, *переменные экземпляра класса*, реже - *свойство*, а иногда просто используют термин *атрибут*. Тем не менее, термин *поле* достаточно компактный и удобный. Им и будем пользоваться.

Фактически поле экземпляра класса - это некоторая переменная, которая "приписана" (или "прикреплена") к этому экземпляру класса. Проблема здесь вот в чем: переменные, как мы знаем, появляются тогда, когда им присваивается значение. Если мы говорим о переменной в контексте экземпляра класса, то естественным образом возникает вопрос: где, в каком месте, переменной (полю) можно присвоить значение? Логично предположить, что при выполнении метода экземпляра класса.

Небольшой пример, в котором реализуется такой подход, представлен в листинге 6.2.

Листинг 6.2. Поле экземпляра класса

```
# Создаем класс
class MyClass:
    # Метод для присваивания значения
    # полю экземпляра класса
    def set(self,n):
        print("Внимание! Присваивается значение!")
        # Полю присваивается значение
        self.number=n
    # Метод для считывания значения
    # поля экземпляра класса
```

```
    def get(self):
        # Отображаем значение поля
        print("Значение поля:",self.number)
# Создаем экземпляр класса
obj=MyClass()
# Вызывается метод экземпляра класса и
# полю экземпляра класса присваивается
# значение
obj.set(100)
# Вызывается метод экземпляра класса и
# отображается значение поля экземпляра
# класса
obj.get()
```

При выполнении этого программного кода в окне вывода появляется два сообщения, как показано ниже:

Результат выполнения программы (из листинга 6.2)

```
Внимание! Присваивается значение!
Значение поля: 100
```

Проанализируем программный код. Мы, как и в предыдущем примере, создаем класс с названием `MyClass`. Но теперь в этом классе описаны два метода экземпляра класса: метод `set()` предназначен для присваивания значения полю экземпляра класса. Поле называется `number`, и о его существовании мы узнаем исключительно из программного кода метода `set()`: в том месте, где командой `self.number=n` полю `number` экземпляра класса, на который ссылается переменная `self`, присваивается значение переменной `n`. И переменная `self` (первый аргумент), и переменная `n` (второй аргумент) объявлены как аргументы метода `set()`.

Переменная `self` представляет собой ссылку на экземпляр класса, из которого вызывается метод, а переменная `n` - это непосредственно тот аргумент, который передается методу при вызове. Впоследствии, когда командой `obj=MyClass()` будет создан экземпляр `obj` класса `MyClass` и затем командой `obj.set(100)` из этого экземпляра вызван метод `set()` с аргументом `100`, данное значение будет присвоено полю `number` экземпляра `obj`.

На заметку

Перед тем, как методом `set()` полю `number` присваивается значение, командой `print("Внимание! Присваивается значение!")` в теле метода выводится сообщение. Так что процесс присваивания значения полю незамеченным не проходит.

Метод экземпляра класса `get()` предназначен для считывания значения поля `number` экземпляра класса (точнее, для отображения значения этого поля в окне вывода). В теле метода всего одна команда `print("Значение поля:",self.number)`, которой и решается поставленная перед методом задача. Ссылка `self.number` на поле `number` экземпляра класса выполняется через переменную `self` - единственный аргумент метода экземпляра класса. При вызове из экземпляра класса методу `get()` аргументы не передаются. Так, в результате выполнения команды `obj.get()` в окне вывода отображается значение поля `number` экземпляра `obj`. Важно то, что команда `obj.get()` выполняется после команды `obj.set(100)`, поскольку прежде, чем значение поля получить, это значение полю нужно присвоить.

На заметку

На самом деле ситуация еще более "жесткая": при первом вызове метода `set()` из экземпляра класса полю `number` не просто присваивается значение - это поле создается. То есть до тех пор, пока не выполнена команда присваивания значения полю `number` экземпляра `obj` (а в данном случае такое присваивание выполняется методом `set()`) у экземпляра `obj` поля `number` как бы не существует. Совсем.

Значение полю экземпляра класса можно присвоить через прямое обращение к этому полю. Рассмотрим небольшой пример в листинге 6.3.

Листинг 6.3. Значение поля экземпляра класса

```
# Создаем класс без методов
class MyClass:
    pass
# Создаем экземпляр класса
obj=MyClass()
# Присваивается значение полю number
# экземпляра obj
obj.number=100
# Отображается значение поля number
# экземпляра obj
print("Значение поля:",obj.number)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 6.3)

```
Значение поля: 100
```

В данном случае мы создаем класс, в котором вообще ничего нет - никакие методы в теле `MyClass` класса не описываются. Там есть только инструк-

ция pass. С помощью этой инструкции мы выделяем тело класса. Проблема в том, что если там не написать вообще ничего, то такой синтаксис будет содержать ошибку. Поэтому даже если класс ничего не содержит, что-то там все равно должно быть. В таких случаях используем формальную ничего не значащую инструкцию pass.

Экземпляр класса, как и во всех предыдущих случаях, создаем командой obj=MyClass(). Затем командой obj.number=100 экземпляр obj получает поле number, и этому полю присваивается значение 100. После этого значение поля number экземпляра obj считывается и отображается в окне вывода командой print("Значение поля:",obj.number).

На заметку

Не нужно быть Шерлоком Холмсом, чтобы понять: в Python разные экземпляры одного и того же класса могут иметь разный набор полей. Такое положение дел является совершенно немыслимым в C++, Java, C#: в этих языках программирования полный набор характеристик экземпляров класса (выражаясь нашей терминологией) определяется раз и навсегда тем, что описано в классе, на основе которых создаются экземпляры. Поэтому важно понимать, что классы и экземпляры класса в Python - это далеко не то же самое, что классы и объекты в C++, Java и C#.

Ситуация, когда значения полям экземпляров класса присваиваются (явно или посредством специальных методов) уже после создания экземпляра класса не очень удобна. Особенно она неудобна, если приходится иметь дело со значительным количеством экземпляров. Существует механизм, который позволяет частично (или даже полностью) снять эту проблему. Речь идет об использовании *конструктора*.

Конструктор и деструктор экземпляра класса

Как говорит наш дорогой шеф, в нашем деле главное — этот самый реализм.
из к/ф "Бриллиантовая рука"

При создании класса можно описать специальный метод, который называется *конструктором экземпляра класса* (иногда также называют *методом инициализации*). Этот метод автоматически вызывается при создании экземпляра класса. Рецепт создания конструктора очень простой: необходимо создать метод с названием __init__() (два символа подчеркивания в начале и два символа подчеркивания в конце). Аргументов у конструктора может быть сколько угодно - но не меньше одного (ссылка на экземпляр, при создании которого вызывается конструктор).

На заметку

В Python есть группа специальных методов, которые позволяют составлять гибкие и эффективные программные коды. Название таких методов начинается с двойного подчеркивания и заканчивается двойным подчеркиванием. Со многими из этих методов и их назначением нам предстоит познакомиться. В данном случае мы обсуждаем конструктор `__init__()`.

В листинге 6.4 приведен пример создания класса, который содержит конструктор.

Листинг 6.4. Конструктор экземпляра класса

```
# Создаем класс
class MyClass:
    # Конструктор
    def __init__(self):
        # Присваивается значение полю
        self.number=0
        # Отображается сообщение
        print("Создан экземпляр класса!")
# Создается экземпляр класса
obj=MyClass()
# Проверяем значение поля экземпляра класса
print("Значение поля:",obj.number)
```

При выполнении программного кода получаем такой результат:

Результат выполнения программы (из листинга 6.4)

```
Создан экземпляр класса!
Значение поля: 0
```

В классе `MyClass` описан конструктор `__init__()` с одним аргументом `self` (в случае конструктора это ссылка на создаваемый экземпляр класса). В теле конструктора командой `self.number=0` полю `number` экземпляра класса присваивается нулевое значение, а затем командой `print("Создан экземпляр класса!")` в окне вывода отображается сообщение.

Эти два действия запрограммированы в конструкторе, который, напомним, автоматически вызывается при создании экземпляра класса. Поэтому каждый раз, когда создается экземпляр, этому экземпляру автоматически будет добавляться поле `number` с нулевым значением, и после этого, опять же автоматически, будет появляться сообщение в окне вывода. Поэтому при создании экземпляра `obj` командой `obj=MyClass()` у этого экземпляра появляется поле `number`, и этому полю присваивается значение `0`. Также в

окне вывода появится сообщение `Создан экземпляр класса!`. С помощью команды `print("Значение поля:",obj.number)` мы проверяем, справедливы ли все те утверждения относительно поля `number` экземпляра `obj`, которые были сделаны выше. Результат выполнения этой команды не дает поводов для сомнений.

У конструктора может быть несколько аргументов (во всяком случае, больше одного). В этом случае при создании экземпляра класса конструктору нужно передать необходимые аргументы. Как это делается, проиллюстрировано в листинге 6.5.

Листинг 6.5. Аргументы конструктора

```
# Создаем класс
class MyClass:
    # Метод для присваивания значения полю
    def set(self,n):
        # Полю number присваивается значение
        self.num=n
    # Метод для отображения значения поля
    def get(self):
        # Отображается значение поля number
        print("Значение поля:",self.num)
    # Конструктор с двумя аргументами.
    # У второго аргумента есть значение
    # по умолчанию
    def __init__(self,n=0):
        # Вызывается метод set() для присваивания
        # значения полю number
        self.set(n)
        # Отображается сообщение
        print("Создан экземпляр класса.")
        # Вызывается метод get() для отображения
        # значения поля number
        self.get()
# Создается экземпляр класса
a=MyClass()
# Создается еще один экземпляр класса
b=MyClass(100)
```

Здесь мы создаем класс `MyClass`, в теле которого описаны методы экземпляра класса `set()` и `get()`. Первый метод предназначен для присваивания значения полю `number` экземпляра класса, а второй метод нужен для отображения значения этого поля. У конструктора теперь два аргумента. Первый, как всегда, является ссылкой на экземпляр класса, а второй аргумент по нашей задумке, определяет значение поля `number` экземпляра

класса. Причем этот второй аргумент имеет нулевое значение по умолчанию: то есть если при создании экземпляра аргумент указать, то это будет значение поля number, а если значение не указать, то у поля number будет нулевое значение.

Возникает вопрос: а как передать аргумент конструктору? Очень просто: в команде создания экземпляра класса после имени класса в круглых скобках (которые до этого у нас всегда были пустыми) передаются аргументы конструктору. Другими словами, шаблон команды создания экземпляра класса с передачей аргументов конструктору такой: переменная=класс(аргументы). Что касается данного конкретного случая, то поскольку второй аргумент имеет значение по умолчанию, мы можем как передавать аргумент конструктору, так и не передавать.

Примером первой ситуации является команда b=MyClass(100) (экземпляр b создается со значением 100 для поля number), а второй - команда a=MyClass() (экземпляр a создается со значением 0 для поля number).

Результат выполнения приведенного выше кода такой:

Результат выполнения программы (из листинга 6.5)

```
Создан экземпляр класса.
Значение поля: 0
Создан экземпляр класса.
Значение поля: 100
```

Все сообщения отображаются при выполнении программного кода конструкторов, поскольку программа, кроме создания класса и двух экземпляров класса, никаких иных командных блоков не содержит.

Метод с названием __del__() (два символа подчеркивания в начале и два - в конце) автоматически вызывается при удалении экземпляра класса из памяти. Этот метод принято называть *деструктором*. У деструктора один и только один аргумент (ссылка на экземпляр класса self).

Небольшой пример с использованием деструктора приведен в листинге 6.6.

Листинг 6.6. Деструктор экземпляра класса

```
# Класс с конструктором и деструктором
class MyClass:
    # Конструктор
    def __init__(self):
        print("Всем привет!")
```

```
    # Деструктор
    def __del__(self):
        print("Всем пока!")
print("Проверяем работу деструктора.")
# Создаем экземпляр класса
obj=MyClass()
print("Экземпляр класса создан. Удаляем его.")
# Удаляем экземпляр класса
del obj
print("Выполнение программы завершено.")
```

Мы создаем класс, в котором описаны конструктор и деструктор. И в теле конструктора, и в теле деструктора выполняется по одной команде. И при вызове конструктора, и при вызове деструктора в окне вывода отображаются сообщения - только разные для конструктора и деструктора. При создании экземпляра класса появится сообщение `Всем привет!`, а при удалении экземпляра класса из памяти появляется сообщение `Всем пока!`. Чтобы удалить экземпляр класса (который до этого был создан командой `obj=MyClass()`) используем инструкцию `del obj`. Возможный результат выполнения программного кода представлен ниже:

Результат выполнения программы (из листинга 6.6)

```
Проверяем работу деструктора.
Всем привет!
Экземпляр класса создан. Удаляем его.
Всем пока!
Выполнение программы завершено.
```

Хотя вся эта схема с использованием деструктора выглядит довольно элегантно, у нее есть существенный недостаток, причем касается он не только рассмотренного примера, но деструкторов вообще. Дело в том, что очистка памяти в Python выполняется автоматически. Удаляются объекты, на которые в программе нет ссылок. Но сказать, когда конкретно они будут удалены практически невозможно. Поэтому если некоторый объект должен быть удален, то он рано или поздно будет удален. Но когда именно - вопрос сложный.

В контексте использования деструкторов для экземпляров класса это означает, что мы можем утверждать, что деструктор будет вызван (тогда, когда экземпляр реально удаляется из памяти), но не совсем точно можем сказать, когда именно это произойдет. Поэтому деструкторы в Python используются не очень часто.

Поле объекта класса

- Крупное дело?
- Давно такого не было!
из к/ф "Приключения Шерлока Холмса и доктора Ватсона"

Сейчас самое время вспомнить, что класс сам по себе является объектом. Точно так же, как реальным объектом является лист бумаги, на котором выполнен чертеж автомобиля. До этого мы, образно выражаясь, имели дело с постройкой "автомобилей". Сейчас более пристально посмотрим на тот "чертеж", с которого все началось.

А мы начнем с важного момента: программный код класса *выполняется*. До этого мы в теле класса размещали только объявления методов, и никаких других интересных команд там не было, хотя теоретически они там возможны (другой вопрос, нужны ли они там). Самое простое, что приходит на ум - разместить в теле класса команду присваивания значения переменной. В этом случае получим поле, но не экземпляра класса, а непосредственно самого класса (или *поле объекта класса*).

На заметку

О поле объекта класса можно думать как о переменной, которая "известна" только в теле класса. За пределами класса доступ к такой переменной осуществляется через явное указание класса, в котором она объявлена. Используется точечный синтаксис вида `класс.переменная`.

Если вернуться к аналогии с чертежом на листе бумаги, то поле объекта класса - это такая липучка с напоминанием или какой-то информацией, нацепленная на лист бумаги. Все, кто пользуются чертежом, видят и эту липучку. Но на процесс создания автомобилей по чертежу она не влияет. Это просто такой "довесочек" к важному документу, какая-то дополнительная информация.

С точки зрения прикладного программирования для нас на данный момент важны два обстоятельства:

- В теле класса можно присвоить значение переменной (поле класса).
- Обращаться к такой переменной следует так: указывается имя класса, и, через точку, имя переменной.

На заметку

Нередко в справочной литературе можно встретить утверждение, что поля класса играют роль статических переменных (как в языках C++, Java, C#). Для тех, кто

не знаком с этим термином: статическая переменная или статическое поле - это поле, общее для всех экземпляров класса. В некотором смысле можно конечно, относиться к полям класса как к статическим переменным. Но на самом деле это не совсем так - то есть, полной аналогии нет. С одной стороны, мы при определенных обстоятельствах (которые обсуждаются далее) и только для считывания значения можем обращаться к полю класса через экземпляр класса. В этом случае вместо имени класса указывается имя экземпляра класса (но и здесь не так все просто). Что касается присваивания значения полю класса, то через экземпляр класса сделать это крайне проблематично. Чтобы прояснить ситуацию мы далее рассмотрим несколько примеров.

В листинге 6.7 представлен программный код, который в некоторой степени позволят понять разницу между полем объекта класса и полем экземпляра класса.

Листинг 6.7. Поле объекта класса

```
# Создаем класс
class MyClass:
    # Поле класса
    name="Класс MyClass"
    # Метод для присваивания значения
    # полю экземпляра класса
    def set(self,n):
        self.nickname=n
    # Метод для отображения значения
    # поля экземпляра класса
    def get(self):
        print("Значение поля:",self.nickname)
    # Конструктор
    def __init__(self,n):
        # Полю экземпляра класса
        # присваивается значение
        self.set(n)
        # Отображается сообщение
        print("Создан экземпляр класса.")
        # Отображается значение поля экземпляра
        self.get()
# Создается первый экземпляр класса
green=MyClass("Зеленый")
# Обращение к полю класса через экземпляр класса
print("Принадлежность:",green.name)
# Создается второй экземпляр класса
red=MyClass("Красный")
# Обращение к полю класса через экземпляр класса
print("Принадлежность:",red.name)
# Полю класса присваивается значение
```

```
MyClass.name="Здесь могла быть Ваша реклама!"
# Обращение к полю класса через экземпляр класса
print("Спрашивает Красный:",red.name)
# Обращение к полю класса через экземпляр класса
print("Спрашивает Зеленый:",green.name)
```

При выполнении данного программного кода получаем следующий результат:

Результат выполнения программы (из листинга 6.7)

```
Создан экземпляр класса.
Значение поля: Зеленый
Принадлежность: Класс MyClass
Создан экземпляр класса.
Значение поля: Красный
Принадлежность: Класс MyClass
Спрашивает Красный: Здесь могла быть Ваша реклама!
Спрашивает Зеленый: Здесь могла быть Ваша реклама!
```

Проанализируем особенности программного кода. В теле класса `MyClass`, кроме объявления двух методов и конструктора, есть команда `name="Класс MyClass"`, которой фактически создается поле `name` класса `MyClass`, и этому полю присваивается текстовое значение `"Класс MyClass"`. Метод `set()` предназначен для присваивания значения полю экземпляра класса `nickname`. С помощью метода `get()` значение поля `nickname` экземпляра класса отображается в окне вывода. Также в классе описан конструктор. При выполнении кода конструктора полю `nickname` экземпляра класса присваивается значение аргумента, переданного конструктору, отображается сообщение о создании экземпляра класса и значение поля `nickname`. На этом код класса `MyClass` в принципе исчерпан. Далее в программе идут команды, в которых создаются экземпляры класса и выполняются различные манипуляции с полем `name` класса `MyClass`.

Первый экземпляр (переменная `green`) класса `MyClass` создается командой `green=MyClass("Зеленый")`. Для этого экземпляра поле `nickname`, очевидно, получает значение `"Зеленый"`. При вызове конструктора в окне вывода отображаются сообщения с текстом `"Создан экземпляр класса."` и `"Значение поля: Зеленый"`.

Сразу после создания экземпляра `green` следует команда `print("Принадлежность:",green.name)`. В ней обращение (инструкция `green.name`) к полю `name` объекта класса `MyClass` выполняется через экземпляр класса. В результате появляется сообщение с текстом `"Принадлежность: Класс MyClass"`. Таким образом, значением ин-

струкции green.name является текст "Класс MyClass" - то есть это значение поля name класса MyClass. Если бы мы вместо инструкции green.name использовали MyClass.name, получили бы точно такой же результат. В данной конкретной ситуации инструкции green.name и MyClass.name эквиваленты с точки зрения результата (но это не означает, что так будет всегда).

На заметку

Причины такой "эквивалентности" не так очевидны, как может показаться на первый взгляд. Все это мы обсудим позже.

Второй экземпляр (переменная red) класса MyClass создается командой red=MyClass("Красный"). Поле nickname экземпляра red получает значение "Красный". При вызове конструктора в окне вывода отображается текст "Создан экземпляр класса." и "Значение поля: Красный". При обращении к полю name класса MyClass через экземпляр класса в команде print("Принадлежность:",red.name) результатом инструкции red.name будет текстовое значение поля name класса MyClass, о чем свидетельствует сообщение Принадлежность: Класс MyClass в окне вывода.

Как и в предыдущем случае, обращение к полю класса через экземпляр класса дает такой же результат, как это было бы при использовании инструкции MyClass.name. Еще один вывод состоит в том, что какой бы экземпляр класса мы ни использовали, если мы обращаемся через экземпляр к полю класса, получаем один и тот же результат - значение поля класса.

На заметку

Читатель, знакомый с языками C++, Java или C# такую ситуацию найдет очень похожей на ситуацию с использованием статических полей.

Чтобы проверить наши подозрения, командой MyClass.name="Здесь могла быть Ваша реклама!" присваиваем новое значение полю name класса MyClass. Затем выполняются команды print("Спрашивает Красный:",red.name) и print("Спрашивает Зеленый:",green.name), в которых обращение к полю класса выполняется через экземпляры класса.

В результате выполнения этих команд в окне вывода появляются соответственно такие сообщения: Спрашивает Красный: Здесь могла быть Ваша реклама! и Спрашивает Зеленый: Здесь могла быть Ваша реклама!. Вывод простой: обе инструкции

green.name и red.name на самом деле возвращают значение поля name класса MyClass. Может показаться, что поле класса - это всего лишь общее поле для всех экземпляров класса. Но тогда возникает вопрос: а что будет, если мы попытаемся изменить поле класса, присвоив ему значение не через ссылку на объект класса (например, MyClass.name), а через ссылку на экземпляр класса (например, green.name или red.name)? По логике можно ожидать, что изменения "почувствуют" все экземпляры класса. Но это не так. Рассмотрим программный код в листинге 6.8.

Листинг 6.8. Поле объекта класса и поле экземпляра класса

```
# Создаем класс
class MyClass:
    # Поле name класса
    name="Класс MyClass"
    # Метод для присваивания значения
    # полю nickname экземпляра класса
    def set(self,n):
        self.nickname=n
    # Метод для отображения значения
    # поля nickname экземпляра класса
    def get(self):
        print("Значение поля:",self.nickname)
    # Конструктор
    def __init__(self,n):
        # Присваивается значение полю
        # nickname экземпляра класса
        self.set(n)
        # Отображается сообщение
        print("Создан экземпляр класса.")
        # Отображается значение поля
        # nickname экземпляра класса
        self.get()
# Первый экземпляр (переменная green)
green=MyClass("Зеленый")
# Проверяем значение поля name
# через экземпляр green
print("Принадлежность:",green.name)
# Второй экземпляр (переменная red)
red=MyClass("Красный")
# Проверяем значение поля name
# через экземпляр red
print("Принадлежность:",red.name)
# Изменяем значение поля name
# через экземпляр green
green.name="Здесь был Зеленый"
```

```
# Проверяем значение поля name
# через экземпляр red
print("Спрашивает Красный:",red.name)
# Проверяем значение поля name
# через экземпляр green
print("Спрашивает Зеленый:",green.name)
# Изменяем значение поля name
# через объект класса MyClass
MyClass.name="Здесь могла быть Ваша реклама!"
# Проверяем значение поля name
# через экземпляр red
print("Спрашивает Красный:",red.name)
# Проверяем значение поля name
# через экземпляр green
print("Спрашивает Зеленый:",green.name)
# Удаляем поле name экземпляра green
del green.name
# Проверяем значение поля name
# через экземпляр green
print("Спрашивает Зеленый:",green.name)
```

Результат выполнения этого программного кода приведен ниже:

Результат выполнения программы (из листинга 6.8)

```
Создан экземпляр класса.
Значение поля: Зеленый
Принадлежность: Класс MyClass
Создан экземпляр класса.
Значение поля: Красный
Принадлежность: Класс MyClass
Спрашивает Красный: Класс MyClass
Спрашивает Зеленый: Здесь был Зеленый
Спрашивает Красный: Здесь могла быть Ваша реклама!
Спрашивает Зеленый: Здесь был Зеленый
Спрашивает Зеленый: Здесь могла быть Ваша реклама!
```

Что касается класса `MyClass`, то он фактически такой же, как и в предыдущем примере. Изменились лишь те команды, "манипуляции", которые мы выполняем с полем класса `name` после создания экземпляров `green` и `red`. Обсудим соответствующую часть программного кода.

Пока мы после создания экземпляров `red` и `green` проверяем значение поля `name` класса `MyClass` с помощью инструкций вида `red.name` и `green.name`, все происходит совершенно ожидаемо: в обоих случаях получаем значение поля `name` класса `MyClass`. Эту ситуацию мы обсужда-

ли выше, в предыдущем примере. А далее выполняется команда green.name="Здесь был Зеленый", которой мы как-бы пытаемся изменить значение поля name класса MyClass, но обращаясь к полю не через объект класса MyClass, а через экземпляр green.

Ранее мы с помощью инструкции green.name считывали значение поля name, поэтому имеем некоторые основания рассчитывать на успех. Однако при выполнении команды print("Спрашивает Красный:",red.name) получаем в окне вывода сообщение Спрашивает Красный: Класс MyClass. То есть значение поля name, полученное по ссылке через экземпляр red не изменилось. Хотя при выполнении команды print("Спрашивает Зеленый:",green.name) в окне вывода получаем вполне ожидаемое сообщение Спрашивает Зеленый: Здесь был Зеленый. Поворот событий достаточно неожиданный: пока мы через инструкцию green.name значение полю name не присваивали, значения выражений green.name и red.name совпадали, но после присваивания значения полю эти инструкции возвращают разные значения. Объяснение такое.

При выполнении команды green.name="Здесь был Зеленый" на самом деле значение полю name класса MyClass не присваивается. Вместо этого создается поле с названием name у экземпляра green, и этому полю присваивается значение "Здесь был Зеленый". И после этого каждый раз, когда мы будем использовать инструкцию green.name, она будет означать не поле name класса MyClass, а поле name экземпляра green. Другими словами поле name экземпляра green "перекрывает" поле name класса MyClass. Но это только для экземпляра green. У экземпляра red поля name нет, поэтому инструкция red.name при считывании значения означает поле name класса MyClass.

🕮 На заметку

На самом деле принцип такой. Если при считывании значения поля через ссылку на экземпляр класса оказывается, что такое поле у экземпляра класса есть, то значение этого поля и возвращается. Если у экземпляра класса поля с таким названием нет, то начинается поиск одноименного поля среди полей класса.

При выполнении команды MyClass.name="Здесь могла быть Ваша реклама!" изменяется значение поля name класса MyClass. Поэтому когда в команде print("Спрашивает Красный:",red.name) используется ссылка red.name, то результатом этой инструкции будет новое значение поля name класса MyClass. А значение инструкции green.name в команде print("Спрашивает Зеленый:",green.name) - это значение "Здесь был Зеленый" поля name экземпляра green.

На следующем этапе командой `del green.name` у экземпляра `green` удаляется поле `name`. После этого у экземпляра `green` поля `name` уже нет, и значением инструкции `green.name` в команде `print("Спрашивает Зеленый:",green.name)` является значение поля `name` класса `MyClass`.

Добавление и удаление полей и методов

Ладно, все. Надо что-то делать. Давай-ка, может быть, сами изобретем.
из к/ф "Чародеи"

После наших смелых экспериментов с полями объекта класса и экземпляров класса читатель, скорее всего, уже догадался, что поля можно добавлять и удалять, причем это справедливо как для объекта класса, так и для экземпляров класса. Здесь, в этом разделе, мы обсудим и проанализируем две позиции:

- Поля экземпляра класса можно добавлять и удалять после создания экземпляра класса.
- Поля объекта класса можно добавлять и удалять после создания объекта класса.

С прикладной точки зрения ситуация в общем и целом несложная: для добавления поля экземпляра класса этому полю присваивается значение (разумеется, через ссылку на экземпляр класса). Примерно то же самое происходит при добавлении поля объекту класса: такому полю присваивается значение, и этого достаточно, чтобы у объекта класса появилось новое поле. Чтобы удалить поле у объекта класса, после оператора `del` указываем ссылку на поле объекта класса (в "точечном "формате: класс и, через точку, имя поля). Аналогичным образом удаляется и поле экземпляра класса, только теперь после оператора `del` указывается ссылка на поле экземпляра класса (экземпляр класса, точки и имя поля).

Небольшой пример манипулирования с полями объекта класса и полями экземпляров класса приведен в листинге 6.9.

Листинг 6.9. Добавление и удаление полей

```
# Создаем класс
class MyClass:
   pass
# Создаем экземпляр A
A=MyClass()
# Создаем экземпляр B
B=MyClass()
```

```
# Экземпляру A добавляем
# поле first
A.first="Экземпляр A"
# Экземпляру B добавляем
# поле second
B.second="Экземпляр B"
# Классу MyClass добавляем
# поле total
MyClass.total="Класс MyClass"
# Проверяем доступ к полям total и
# first через ссылку на экземпляр A
print(A.total,"->",A.first)
# Проверяем доступ к полю second
# через ссылку на экземпляр A
try:
    # Если поле second есть
    print(A.second)
# Если такого поля нет
except AttributeError:
    print("Такого поля у экземпляра A нет!")
# Проверяем доступ к полям total и
# second через ссылку на экземпляр B
print(B.total,"->",B.second)
# Проверяем доступ к полю first
# через ссылку на экземпляр B
try:
    # Если поле first есть
    print(B.first)
# Если такого поля нет
except AttributeError:
    print("Такого поля у экземпляра B нет!")
# Удаляем поле total класса MyClass
del MyClass.total
# Проверяем доступ к полю total
# через ссылку на экземпляр A
try:
    # Если поле есть
    print(A.total)
# Если поля нет
except AttributeError:
    print("Такого поля нет!")
# Проверяем доступ к полю total
# через ссылку на экземпляр B
try:
    # Если поле есть
    print(B.total)
```

```
# Если поля нет
except AttributeError:
    print("Такого поля нет!")
# Удаляем поле first экземпляра A
del A.first
# Проверяем доступ к полю first
# через ссылку на экземпляр A
try:
    # Если поле есть
    print(A.first)
# Если поля нет
except AttributeError:
    print("Такого поля у экземпляра A нет!")
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 6.9)

```
Класс MyClass -> Экземпляр A
Такого поля у экземпляра A нет!
Класс MyClass -> Экземпляр B
Такого поля у экземпляра B нет!
Такого поля нет!
Такого поля нет!
Такого поля у экземпляра A нет!
```

В этом примере создается класс `MyClass`, не содержащий, собственно, ничего. На основе этого класса создаются два экземпляра с названиями `A` и `B`. Затем командам и `A.first="Экземпляр A"`, `B.second="Экземпляр B"` и `MyClass.total="Класс MyClass"` добавляются (с присваиванием значений) поля `first` для экземпляра `A`, поле `second` для экземпляра `B` и поле `total` для класса `MyClass`. В команде `print(A.total,"->",A.first)` есть ссылки на поле `first` экземпляра класса и поле `total` объекта класса, причем обе ссылки выполняются через экземпляр `A`.

По результату выполнения команды (сообщение `Класс MyClass -> Экземпляр A` в окне вывода) видим, что проблем с доступом к полям не возникает. В принципе, нечто подобное мы наблюдали в предыдущих примерах, но здесь есть одна особенность. Связана она с тем, что поле `total` появилось у класса `MyClass` уже после того, как были созданы экземпляры `A` и `B`. Но, тем не менее, доступ к добавленному полю `total` через экземпляры класса имеется. А вот добавление поля одному экземпляру класса для другого экземпляра "остается незамеченным" (что, в принципе, вполне логично).

При попытке выполнить команду `print(A.second)` возникает ошибка класса `AttributeError`: поля `second` у экземпляра `A` нет. Эту ситуацию мы обрабатываем с помощью блока `try-except`. Результатом является сообщение `Такого поля у экземпляра A нет!`, которое отображается командой `print("Такого поля у экземпляра A нет!")` в `except`-блоке.

Аналогично обстоят дела с экземпляром `B`: доступ к полю `total` класса `MyClass` и полю `second` экземпляра через переменную `B` получаем без проблем, а к полю `first` экземпляра доступа нет, поскольку это поле добавлялось в экземпляр `A`, и экземпляр `B` к данному полю никакого отношения не имеет.

На следующем этапе командой `del MyClass.total` удаляем поле `total` объекта класса `MyClass`, после чего командой `print(A.total)` (в `try`-блоке) пытаемся прочитать значение этого поля. Судя по тому, что в этом случае возникает ошибка класса `AttributeError` (неверный атрибут) и в `except`-блоке выполняется команда `print("Такого поля нет!")`, поле `total` действительно удалено из объекта класса `MyClass`. Нет доступа к удаленному полю `total` и через переменную `B`.

Наконец, командой `del A.first` удаляем поле `first` экземпляра `A`. Как следствие, при попытке выполнить команду `print(A.first)` в `try`-блоке возникает ошибка. Поэтому в `except`-блоке выполняется команда `print("Такого поля у экземпляра A нет!")`.

На заметку

Чтобы легче было понять, что и как происходит при добавлении и удалении полей объекта класса и экземпляров класса, следует учесть, что поля и их значения для каждого экземпляра класса и объекта класса хранятся в словаре. Поле является ключом, а значение поля - элементом словаря. При обращении к полю экземпляра поиск сначала выполняется по словарю экземпляра класса, и если там нет соответствующего ключа (напомним, что это на самом деле название поля), то поиск перемещается к словарю с названиями и значениями полей объекта класса. Добавляя или удаляя поля, мы вносим изменения в соответствующий словарь. Поэтому экземпляры, созданные до внесения изменений в объект класса, "знают" об этих изменениях.

Методы и функции

Форму будете создавать под моим личным контролем. Форме сегодня придается большое содержание.
из к/ф "Чародеи"

В этом разделе мы обсудим методы. Тема эта нетривиальная, но мы постараемся выделить основные, так сказать, концептуальные моменты, которые позволят читателю более-менее свободно ориентироваться в том богатстве приемов и подходов, которые имеются в его распоряжении при программировании на Python. Начнем с небольшого примера, представленного в листинге 6.10.

Листинг 6.10. Метод экземпляра и функция класса

```
# Создаем класс
class MyClass:
    # Метод экземпляра
    def say(self):
        # Отображается сообщение
        print("Всем привет!")
# Создаем экземпляр класса
obj=MyClass()
# Вызываем метод экземпляра.
# Аргументов нет
obj.say()
# Вызываем функцию класса.
# Аргумент - экземпляр класса
MyClass.say(obj)
# Вызываем функцию класса.
# Аргумент - текст
MyClass.say("Какой-то текст")
```

Результат выполнения программного кода приведен ниже:

Результат выполнения программы (из листинга 6.10)

```
Всем привет!
Всем привет!
Всем привет!
```

Мы создаем класс `MyClass`, в котором описываем метод экземпляра класса `say()`. В теле метода всего одна команда `print("Всем привет!")`, которой, очевидно, в окне вывода отображается сообщение. Хотя у метода есть аргумент `self`, этот аргумент явно не используется.

В данном классе нет ничего необычного. Тем не менее, мы все же проведем небольшой эксперимент. Для этого командой `obj=MyClass()` создаем экземпляр `obj`. Затем из экземпляра вызываем метод `say()` (имеется в виду команда `obj.say()`). Результат вполне ожидаем: отображается сообщение `Всем привет!`. Затем последовательно выполняются команды `MyClass.say(obj)` и `MyClass.say("Какой-то текст")`. Результат точно такой же, как при выполнении команды `obj.say()`. Попробуем разобраться, почему же происходит именно так.

Начнем с команды `MyClass.say(obj)`. В известном смысле она является воплощением команды `obj.say()`. Дело в том, что когда мы описываем в классе `MyClass` метод экземпляра `say()`, на самом деле мы описываем *функцию* `say()`. Это, в общем-то, самая обычная функция, просто описывается в теле класса. Ранее мы такие *функции* называли *методами* и вызывали через экземпляр класса. Здесь мы вызываем функцию, указав вместо экземпляра класса сам класс. Так тоже можно делать.

На заметку

Чтобы не запутаться, будем называть `say()` функцией, если речь идет о команде `MyClass.say(obj)`, и методом, если речь идет о команде `obj.say()`.

С формальной точки зрения главная особенность функции `say()` в том, что при обращении к ней мы указываем еще и имя класса `MyClass`. Вызов функции `say()` через имя класса - это как если бы мы вызвали самую обычную функцию: командой `MyClass.say(obj)` вызывается функция `say()`, описанная в теле класса `MyClass`, а аргументом этой функции передается ссылка на экземпляр класса `obj`.

Когда мы говорим о вызове *метода* `say()`, то подразумеваем, что вызов осуществляется через ссылку на экземпляр класса. Как мы уже знаем, экземпляр класса неявно передается первым (и в данном случае единственным) аргументом в `say()`. Поэтому команда `obj.say()` эквивалента вызову функции `say()` из класса `MyClass` с аргументом `obj`. Следовательно, нет ничего удивительного в том, что выполнение команд `MyClass.say(obj)` и `obj.say()` приводит к одинаковым результатам. Что касается команды `MyClass.say("Какой-то текст")`, то в известном смысле это "хулиганство", хотя и вполне законное.

Мы вызываем функцию `say()` из класса `MyClass` с аргументом, который является текстовым значением, хотя по логике аргументом должна бы быть ссылка на экземпляр класса. Но поскольку в теле функции `say()` ссылка на экземпляр класса явно не используется (то есть аргумент у функции имеется, но в теле функции не используется), то не имеет особого значения,

что же мы передаем аргументом функции - главное, чтобы аргумент просто был.

Вообще, ситуация не такая простая, как может показаться на первый взгляд. Чтобы ее немного прояснить, имеет смысл напомнить, что функция в Python - это некоторый объект (в общем смысле этого термина), а если точнее, то объект типа `function`. Поэтому на функцию, описанную в классе, можно смотреть именно как на такой объект - по аналогии к тому, как мы описывали в теле класса переменные, которые играли роль полей объекта класса.

Имя функции, описанной в теле класса, сродни имени обычной переменной, описанной в теле класса. Как и в случае поля объекта класса, при ссылке на функцию, описанную в теле класса, имя класса через точку указывается перед именем функции. В контексте сказанного инструкция `MyClass.say` является ссылкой на объект типа `function`.

На заметку

Желающие могут провести такой эксперимент: в программный код, описанный выше, добавить команду `print(type(MyClass.say))`. В результате в окне вывода появится сообщение `<class 'function'>`. Результатом выполнения команды `print(type(obj.say))` будет сообщение `<class 'method'>`.

Мы можем рассматривать имя функции как некоторый атрибут класса, к которому обычно обращаются с указанием круглых скобок (и аргументов в них). В частности, если `функция`, описанная в теле `класса`, вызывается через ссылку на `класс` (то есть в формате `класс.функция(аргументы)`), то фактически подразумевается вызов `функции` с соответствующими `аргументами`.

По большому счету, здесь речь идет об обычной функции, но только "привязанной" к классу. В таком контексте мы можем описать в теле класса практически любую функцию и вызывать ее как обычную функцию, но только используя в названии явную ссылку на класс. С дугой стороны понятно, что обычно не для этого функции описываются в классе. По крайней мере, ранее мы вызывали функции, описанные в теле класса, через ссылку на экземпляр класса и называли такие функции *методами*. Здесь, разумеется, нет противоречия.

Функции, описанные в теле класса, можно вызывать не только через класс, но и через экземпляр класса. В известном смысле при этом мы можем говорить о вызове той же самой функции, что и при вызове через имя класса, но через экземпляр класса обращение к функции осуществляется не так "прямолинейно".

Что же происходит (в общих чертах), когда мы вызываем функцию через экземпляр класса (то есть вызываем метод)? Другими словами, как функция становится методом? А происходит примерно следующее.

- Среди названий функций, описанных в теле класса, выполняется поиск имени метода, который вызывается из экземпляра класса. Предполагаем, что функция с соответствующим названием найдена (если нет, то возникает ошибка).
- Создается объект метода (объект класса `method`). Этот объект соответствует функции, вызываемой с аргументами, которые передаются методу при вызове, но дополнительно первым аргументом добавляется ссылка на экземпляр класса, из которого вызывается метод.

На заметку

Проще говоря, команда вида `экземпляр.имя(аргументы)` эквивалентна команде `класс.имя(экземпляр,аргументы)`.

В дальнейшем для нас наиболее важными будут две позиции:

- При вызове метода первым аргументом неявно передается ссылка на экземпляр класса.
- Имя функции/метода является атрибутом, которому можно присвоить значение.

Что касается первого пункта, то он нам был известен и до этого. По поводу второго пункта потребуются некоторые пояснения. Представить их лучше всего на конкретном примере, что мы и сделаем. Например, совсем необязательно описывать метод экземпляра класса или функцию класса непосредственно в теле класса. Мы можем описать соответствующую функцию отдельно от описания класса, а затем "зарегистрировать" ее как метод экземпляра или как функцию класса. С помощью инструкции `del` метод экземпляра класса или функцию класса можно удалить. Чтобы понять, как выполняются такие операции, рассмотрим листинг 6.11.

Листинг 6.11. Добавление и удаление методов

```
# Создаем класс
class MyClass:
    pass
# Создаем экземпляры класса
A=MyClass()
B=MyClass()
C=MyClass()
```

```
# Создаем первую функцию
def hello():
    print("Метод экземпляра - 'hello'")
# Создаем вторую функцию
def hi():
    print("Еще один метод - 'hi'")
# Определяем метод экземпляра
A.say=hello
# Определяем метод экземпляра
C.say=hi
# Вызываем метод экземпляра
A.say()
# Вызываем метод экземпляра
try:
    B.say()
# Если такого метода нет
except AttributeError:
    print("Такого метода нет")
# Вызываем метод экземпляра
C.say()
# Вызываем функцию класса
try:
    MyClass.say()
# Если такой функции нет
except AttributeError:
    print("Такой функции нет")
# Удаляем метод экземпляра
del A.say
# Вызываем метод зкземпляра
try:
    A.say()
# Если такого метода нет
except AttributeError:
    print("Такого метода нет")
# Вызываем метод зкземпляра
C.say()
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 6.11)

```
Метод экземпляра - 'hello'
Такого метода нет
Еще один метод - 'hi'
Такой функции нет
Такого метода нет
Еще один метод - 'hi'
```

Мы создаем класс с традиционным названием `MyClass`. В классе в данном конкретном случае ничего не описано (ни полей, ни методов). Далее на основе этого класса создается три экземпляра: `A`, `B` и `C`. Также мы описываем две функции `hello()` и `hi()`, у которых нет аргументов и которые не возвращают результат.

При выполнении каждой из этих функций в области вывода отображается сообщение (для каждой функции свое). До этого момента не происходит ничего необычного. Откровенная "экзотика" начинается с команды `A.say=hello`, которой атрибуту `say` экземпляра `A` присваивается в качестве значения идентификатор `hello` - имя одной из созданных нами функций. Что все это означает и что при этом происходит?

Для ответа на эти вопросы следует вспомнить, что функция реализуется через специальный объект, а имя функции - это ссылка на данный объект. Поэтому в данном случае идентификатор `hello` является ссылкой на объект, через который реализуется функция `hello()`.

С другой стороны, если атрибуту `say` экземпляра `A` присваивается ссылка на объект, реализующий функцию, то указанный атрибут будет ссылаться на этот объект. Значит, атрибут `say` можно рассматривать как функцию, причем это та же функция, что и функция `hello()`. Поэтому, когда выполняется команда `A.say()`, на самом деле вызывается функция `hello()`.

Нечто похожее происходит при выполнении команды `C.say=hi`, с поправкой лишь на то, что речь идет об экземпляре `C` (а не `A`) и функции `hi()` (а не `hello()`). В результате при выполнении команды `C.say()` вызывается функция `hi()`. А вот при выполнении команды `B.say()` возникает ошибка (типа `AttributeError`), так как атрибута `say` у экземпляра `B` нет. Точно так же, как нет такого атрибута и у объекта класса `MyClass`. Поэтому команда `MyClass.say()` тоже приводит к ошибке.

Чтобы удалить атрибут `say` у экземпляра `A`, используем команду `del A.say`. После этого команда `A.say()` будет приводить к ошибке, а команда `C.say()` - нет, поскольку у экземпляра `C` атрибут `say` не удалялся.

🕮 На заметку

Утверждать, что мы, присваивая атрибутам экземпляров класса в качестве значений ссылки на функции, создавали тем самым методы экземпляров, будет не совсем корректно. Есть несколько важных обстоятельств, которые существенно охлаждают наш пыл. Во-первых, функции, ссылки на которые мы присваивали атрибутам, не имеют доступа к экземплярам класса. Другими словами, мы не сможем изменить программный код функции `hello()` так, чтобы при выполнении команды `A.say()` был бы прямой доступ к экземпляру `A`. Как бы ни хотелось нам рассматривать `say()` как метод экземпляра класса, данный метод фактически к экземпляру класса доступа не имеет. Это серьезный минус.

Далее, если бы мы взяли несколько экземпляров и атрибуту `say` каждого из этих экземпляров присвоили ссылку на функцию `hello()`, то все эти экземпляры через атрибут `say` ссылались бы на одну и ту же функцию. Получается "общая" функция для всех экземпляров. В том, что это действительно функция, а не метод, легко убедиться - достаточно после команды `A.say=hello` добавить команду `print(type(A.say))`. В окне вывода появится сообщение `<class 'function'>`. Напомним, что когда проверяется тип метода экземпляра, в аналогичной ситуации появляется сообщение `<class 'method'>`. Короче говоря, тем, что мы делали выше, лучше все же особо не увлекаться.

Более эффективным может быть присваивание ссылки на функцию в качестве значения атрибуту класса (разумеется, обратная процедура - удаление атрибута со ссылкой на функцию, - тоже допускается). Небольшой пример представлен в листинге 6.12.

Листинг 6.12. Добавление и удаление функции класса

```
# Создаем класс
class MyClass:
    def __init__(self,n):
        self.name=n
# Создаем экземпляры класса
A=MyClass("A")
B=MyClass("B")
# Создаем функцию с аргументом
def hello(self):
    print("Это экземпляр",self.name,"- hello")
# Создаем функцию с аргументом
def hi(self):
    print(self.name+": hi")
# Определяем функцию класса
MyClass.say=hello
# Вызываем метод экземпляра
A.say()
# Вызываем метод экземпляра
B.say()
# Вызываем функцию класса
MyClass.say(A)
MyClass.say(B)
# Меняем ссылку на функцию
MyClass.say=hi
# Вызываем метод экземпляра
A.say()
# Вызываем метод экземпляра
B.say()
# Вызываем функцию класса
MyClass.say(A)
```

```
MyClass.say(B)
# Удаляем функцию класса
del MyClass.say
# Вызываем метод экземпляра
try:
    A.say()
# Если метода нет
except AttributeError:
    print("Такого метода нет")
# Вызываем метод экземпляра
try:
    B.say()
# Если метода нет
except AttributeError:
    print("Такого метода нет")
# Вызываем функцию класса
try:
    MyClass.say(A)
# Если функции нет
except AttributeError:
    print("Такой функции нет")
```

Результат выполнения этого программного кода представлен ниже:

Результат выполнения программы (из листинга 6.12)

```
Это экземпляр A - hello
Это экземпляр B - hello
Это экземпляр A - hello
Это экземпляр B - hello
A: hi
B: hi
A: hi
B: hi
Такого метода нет
Такого метода нет
Такой функции нет
```

В классе `MyClass` описан конструктор. При создании экземпляра класса конструктору передается аргумент, который определяет значение поля `name` экземпляра класса. Мы создаем два экземпляра: `A` (со значением `"A"` для поля `name`) и `B` (со значением `"B"` для поля `name`). Также описываем две функции `hello()` и `hi()`. Причем у каждой из функций объявлено по одному аргументу (аргумент называется `self`), причем неявно предполагается, что это ссылка на экземпляр класса `MyClass`, поскольку в теле функций имеются ссылки `self.name` на поле `name` экземпляра `self`.

Вместе с тем, обе эти функции описаны вне тела класса MyClass, так что говорить о функции класса или методе экземпляра пока рано - рано, пока не выполнена команда MyClass.say=hello. В этом случае атрибуту say класса MyClass присваивается в качестве значения ссылка на функцию hello(). После этого можем вызывать метод say() для экземпляров класса командами A.say() и B.say(). При этом фактически вызывается функция hello(), аргументом которой передается ссылка на экземпляр класса, из которого вызывается метод say().

Аналогичные результаты получаем при выполнении команд MyClass.say(A) и MyClass.say(B). В них непосредственно вызывается функция say() класса MyClass с одним аргументом.

Мы можем изменить значение атрибута say класса MyClass, выполнив, например, команду MyClass.say=hi. После выполнения этой команды при вызове функции класса или метода экземпляра say() реально будет выполняться код функции hi(). В последнем легко убедиться с помощью команд A.say(), B.say(), MyClass.say(A) и MyClass.say(B). Наконец, мы можем удалить функцию say() класса MyClass командой del MyClass.say. После это выполнение перечисленных выше команд с вызовом функции/метода say() приводит к ошибке.

Возможны и другие варианты "действий" с функциями класса и методами экземпляров класса. Некоторые небольшие примеры приведены в листинге 6.13.

Листинг 6.13. Операции с методами и функциями

```
# Создаем класс
class MyClass:
    # Конструктор экземпляра класса
    def __init__(self,n):
        self.name=n
    # Метод экземпляра класса
    def say(self):
        print("Класс MyClass:",self.name)
# Создаем экземпляры класса
A=MyClass("A")
B=MyClass("B")
# Вызываем метод экземпляра
A.say()
B.say()
# Ссылку на метод записываем в переменную
F=A.say
# Вызываем функцию
F()
```

```
# Атрибуту экземпляра присваивается текст
A.say="Поле экземпляра A"
# Проверяем значение поля
print(A.say)
# Пытаемся вызвать метод экземпляра
try:
    A.say()
# Если такого метода нет
except TypeError:
    print("Неверная команда")
# Вызываем метод экземпляра
B.say()
# Вызываем функцию
F()
```

В классе `MyClass` описан конструктор экземпляра класса, в котором полю `name` экземпляра присваивается значение. Также в классе описан метод `say()`, отображающий текст и значение поля `name` экземпляра, из которого вызывается метод. При создании экземпляры `A` и `B` получают для своих полей `name` соответственно значения `"A"` и `"B"`. У каждого из экземпляров `A` и `B` есть метод `say()`, в чем несложно убедиться с помощью команд `A.say()` и `B.say()`.

Дальше мы будем рассуждать так. Инструкция `A.say` представляет собой ссылку на объект метода `say()` экземпляра `A` класса `MyClass`. Никто не запрещает нам присвоить ссылку на этот объект в некоторую переменную. Именно так мы и поступаем, когда используем команду `F=A.say`. В результате ее выполнения в переменную `F` записана ссылка на объект - тот самый, на который ссылается инструкция `A.say`. Поскольку речь идет об объекте метода, то вызвать этот метод можем инструкцией `A.say()` или `F()`. В обоих случаях вызывается один и тот же метод, поэтому мы получаем один и.тот же результат.

На следующем шаге командой `A.say="Поле экземпляра A"` атрибуту `say` экземпляра `A` присваивается текстовое значение. Хотя ранее атрибут `say` экземпляра `A` ссылался на метод экземпляра, теперь это фактически поле, которое ссылается на текст. Поэтому вызвать метод экземпляра командой `A.say()` больше не получится. Если мы воспользуемся командой `A.say()`, возникнет ошибка несогласования типов `TypeError`. Корректным является обращение `A.say` к текстовому полю экземпляра.

🕮 На заметку

Таким образом, у экземпляра `A` появляется поле `say`, и данное поле "перекрывает" метод `say()` экземпляра. Понятно, что это общее правило.

При этом у экземпляра B метод say() "остается" и без проблем может быть вызван командой B.say(). Но самое интересное, что инструкцией F() мы по-прежнему можем вызвать метод say() экземпляра A. Объяснение простое: переменная F ссылается на объект метода say() экземпляра A. Ранее на этот объект содержалась ссылка и в атрибуте A.say. И хотя после присваивания текстового значения атрибуту say инструкция A.say() для вызова метода использована быть не может, переменная F для этой цели вполне подойдет. В итоге после выполнения всего программного кода получаем такой результат:

Результат выполнения программы (из листинга 6.13)

```
Класс MyClass: A
Класс MyClass: B
Класс MyClass: A
Поле экземпляра A
Неверная команда
Класс MyClass: B
Класс MyClass: A
```

Хотя описанные выше возможности по манипулированию полями и методами достаточно уникальны (такие "трюки" можно делать далеко не в каждом языке программирования), картина была бы неполной без одного очень важного механизма, без которого вообще не было бы смысла рассматривать ООП. Речь о *наследовании*. Наследование, равно как и некоторые другие моменты (например, переопределение операторов) рассматривается в следующей главе.

Копирование экземпляров и конструктор создания копии

Хотите обмануть мага? Боже, какая детская непосредственность. Я же вижу Вас насквозь.
из к/ф "31 июня"

Конструкторы мы уже обсуждали. Мы знаем, что конструктору экземпляра класса могут передаваться аргументы. Это с одной стороны. С другой стороны есть в известном смысле "классическая" задача, которая состоит в том, что на основе одного экземпляра класса нужно создать точно такой же. Грубо говоря, актуальной является задача создания копии экземпляра класса. Проблема в том, что обычным присваиванием значения переменным, которые ссылаются на экземпляры классов, здесь не обойтись. Ведь при присваивании значения переменным просто "перебрасываются" ссылки. Так, если

некоторая переменная x ссылается на экземпляр класса, то после выполнения команды y=x переменная y будет ссылаться на тот самый экземпляр, что и переменная x. И это не то, что нам нужно. Нам нужно, образно выражаясь, чтобы переменная y ссылалась на экземпляр с точно такими же характеристиками, как у экземпляра, на который ссылается переменная x.

На заметку

То есть нам как бы нужен точно такой же экземпляр, но другой.

В принципе для создания копии экземпляра можно воспользоваться функциями `copy()` или `deepcopy()` из модуля `copy`. При использовании функции `copy()` создается *поверхностная копия* экземпляра, в то время как функция `deepcopy()` позволяет создавать *полные копии*. Разница между поверхностной и полной копиями проявляется, когда среди полей экземпляров имеются переменные, ссылающиеся на данные *изменяемых типов* (примером могут быть списки). Небольшая иллюстрация к использованию этих функций приведена в листинге 6.14.

Листинг 6.14. Создание копии экземпляра

```
# Импорт функций copy() и deepcopy()
# из модуля copy
from copy import copy, deepcopy
# Класс
class MyClass:
    # Конструктор
    def __init__(self,name,nums):
        # Значение поля name
        self.name=name
        # Значение поля nums
        self.nums=nums
    # Метод для отображения
    # значений полей экземпляра
    def show(self):
        # Поле name
        print("name ->",self.name)
        # Поле nums
        print("nums ->",self.nums)
# Создание экземпляра класса
x=MyClass("Python",[1,2,3])
print("Экземпляр x:")
# Отображение полей экземпляра x
x.show()
# Поверхностная копия экземпляра x
```

```
y=copy(x)
# Полная копия экземпляра x
z=deepcopy(x)
print("Экземпляр y:")
# Отображение полей экземпляра y
y.show()
print("Экземпляр z:")
# Отображение полей экземпляра z
z.show()
print("Поля экземпляра x изменяются!")
# Изменение значения поля name
# экземпляра x
x.name="Java"
# Изменение значения элемента в списке
# nums - поле экземпляра x
x.nums[0]=0
print("Экземпляр x:")
# Отображение полей экземпляра x
x.show()
print("Экземпляр y:")
# Отображение полей экземпляра y
y.show()
print("Экземпляр z:")
# Отображение полей экземпляра z
z.show()
```

В результате выполнения программного кода получаем такое:

Результат выполнения программы (из листинга 6.14)

```
Экземпляр x:
name -> Python
nums -> [1, 2, 3]
Экземпляр y:
name -> Python
nums -> [1, 2, 3]
Экземпляр z:
name -> Python
nums -> [1, 2, 3]
Поля экземпляра x изменяются!
Экземпляр x:
name -> Java
nums -> [0, 2, 3]
Экземпляр y:
name -> Python
nums -> [0, 2, 3]
Экземпляр z:
```

```
name -> Python
nums -> [1, 2, 3]
```

Импорт функций `copy()` и `deepcopy()` из модуля `copy` выполняется инструкцией `from copy import copy, deepcopy`. Класс, над экземплярами которого будут проводиться эксперименты по "клонированию", называется `MyClass`. В конструкторе экземпляра класса определяются поля `name` и `nums`. Присваиваемые полям значения передаются аргументами конструктору. Мы предполагаем, что поле `name` является текстовым, а поле `nums` представляет собой числовой список (хотя это, конечно, условности).

В классе описан метод экземпляра `show()`, которым выполняется отображение полей `name` и `nums` экземпляра класса. На этом описание класса заканчивается. Далее переходим к процедуре создания копий экземпляров. Для этой цели нам нужен исходный экземпляр класса, который будет копироваться. Экземпляр класса создаем командой `x=MyClass("Python",[1,2,3])`. В результате создается экземпляр `x` класса `MyClass`. У созданного экземпляра класса значением поля `name` есть текст `"Python"`, а значение поля `nums` - список `[1,2,3]`. Проверить значение полей экземпляра `x` можно с помощью команды `x.show()`.

Копии экземпляра `x` создаются так: командой `y=copy(x)` создается поверхностная копия (экземпляр `y`) экземпляра `x`, а командой `z=deepcopy(x)` создается полная копия (экземпляр `z`) экземпляра `x`. На этом этапе значения полей у экземпляров `y` и `z` такие же, как у экземпляра `x`: подтверждением служит результат выполнения команд `y.show()` и `z.show()`.

На следующем этапе командами `x.name="Java"` и `x.nums[0]=0` изменяется значение поля `name` экземпляра `x` и первый (с нулевым индексом) элемент в списке `nums`, являющемся полем этого же экземпляра. Командами `x.show()`, `y.show()` и `z.show()` проверяем, изменились ли (и если да, то как) поля экземпляров `x`, `y` и `z`. С экземпляром `x` все просто. Его поля изменились строго в соответствии с теми командами, которые были выполнены при присваивании значений.

Поля экземпляра `z` (полная копия) не изменились. Что касается экземпляра `y`, то у него не изменилось поле `name`, но изменилось поле-список `nums` (первый элемент стал нулевым, как и у экземпляра `x`). Объяснение в том, что при создании поверхностной копии для изменяемых типов, таких как списки, выполняется копирование ссылок, но не значений. Поэтому реально экземпляры `x` и `y` (поверхностная копия `x`) ссылаются через свои поля `nums` на один и тот же список, тогда как у экземпляра `z` (полная копия `x`) поле `nums` "персональное".

Хотя описанный выше подход легитимен, но все же не всегда приемлем (в силу разных причин). Существуют и иные варианты. В частности, мы можем посмотреть на ситуацию более широко и переформулировать задачу о создании копии экземпляра как задачу о создании экземпляра класса на основе уже существующего экземпляра. Если при этом требовать, чтобы созданный экземпляр имел такие же значения полей исходного экземпляра, то речь будет идти о создании копии.

С практической точки зрения такой подход означает, что в классе необходимо описать конструктор экземпляра, аргументом которому передается ссылка на другой экземпляр этого класса. По традиции означенный конструктор называется *конструктором создания копии* (хотя при этом может создаваться совсем не копия).

На заметку

Проблема в том, что в Python нет перегрузки методов. Поэтому мы не можем, как в C++ или Java, создать несколько конструкторов разных типов. В Python у экземпляра класса конструктор должен быть один. Это вносит некоторую интригу в процесс создания конструктора копии.

На практике очень удобно, если экземпляры класса могут создаваться разными способами. В первую очередь имеется в виду возможность передавать конструктору при создании экземпляра класса различные наборы аргументов. Это же замечание, кстати, относится и к обычным методам: хорошо, когда метод может вызываться с разными аргументами. В таких языках программирования, как C++, Java и C# проблема снимается с помощью механизма *перегрузки методов и функций.*

В языке Python как такового механизма перегрузки методов нет. Но, откровенно говоря, учитывая исключительную "гибкость" языка Python, особой необходимости в перегрузке не наблюдается. Существуют другие возможности. Одна из них - создание функций и методов с переменным количеством аргументов. Этот подход описывается в последней главе.

Здесь мы рассмотрим пример класса с конструктором, которому можно передавать различное количество аргументов, включая и случай, когда аргументом передается ссылка на уже существующий экземпляр класса.

Механизм перегрузки нам удастся "обойти" благодаря назначению аргументам значений по умолчанию и воспользовавшись проверкой типа переданного конструктору аргумента. Соответствующий программный код приведен в листинге 6.15.

Листинг 6.15. Конструктор создания копии экземпляра

```
# Класс
class ComplNum:
    # Конструктор создания экземпляра класса
    def __init__(self,x=0,y=0):
        # Если аргумент x - экземпляр
        # класса ComplNum
        if type(x)==ComplNum:
            # Значение поля Re
            self.Re=x.Re
            # Значение поля Im
            self.Im=x.Im
        # Если аргумент x - не экземпляр
        # класса ComplNum
        else:
            # Значение поля Re
            self.Re=x
            # Значение поля Im
            self.Im=y
    # Метод для отображения значений
    # полей экземпляра класса
    def show(self):
        print("Re =",self.Re)
        print("Im =",self.Im)
# Создается экземпляр класса
a=ComplNum(1,2)
# Создается копия экземпляра класса
b=ComplNum(a)
print("Экземпляр a:")
# Значения полей исходного экземпляра
a.show()
print("Экземпляр b:")
# Значения полей экземпляра-копии
b.show()
print("Поля экземпляра a изменяются!")
# Изменяем значения полей исходного
# экземпляра
a.Re=10
a.Im=20
print("Экземпляр a:")
# Значения полей исходного экземпляра
a.show()
print("Экземпляр b:")
# Значения полей экземпляра-копии
b.show()
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 6.15)

```
Экземпляр a:
Re = 1
Im = 2
Экземпляр b:
Re = 1
Im = 2
Поля экземпляра a изменяются!
Экземпляр a:
Re = 10
Im = 20
Экземпляр b:
Re = 1
Im = 2
```

В представленном примере мы создаем класс `ComplNum` со слабым намеком на реализацию комплексных чисел посредством экземпляров этого класса. Во всяком случае, у экземпляров класса есть два поля: `Re` и `Im` (как бы действительная и мнимая части комплексного числа). При создании экземпляра класса аргументами конструктору можно передать значения для полей `Re` и `Im`, а можно передать ссылку на экземпляр класса, на основе которого будет создаваться копия.

На заметку

Конструктор описан с тремя аргументами. Первый аргумент `self` является ссылкой на экземпляр класса (тот экземпляр, что создается). Еще два аргумента обозначены как `x` и `y`. У каждого из этих аргументов есть нулевые значения по умолчанию. Поэтому формально конструктор можно вызывать без аргументов, с одним аргументом или с двумя аргументами. Если конструктор вызывается без аргументов, то это все равно, как если бы он вызывался с двумя нулевыми аргументами. Если конструктору передан только один аргумент, то второй считается нулевым.

В теле конструктора в условном операторе проверяется тип первого аргумента. Точнее, проверяется условие `type(x)==ComplNum`, которое состоит в том, что тип данных, на которые ссылается аргумент `x`, является экземпляром класса `ComplNum`. Если это так, то аргумент `x` обрабатывается как экземпляр класса `ComplNum`.

Командами `self.Re=x.Re` и `self.Im=x.Im` значения полей `Re` и `Im` экземпляра `x` присваивается соответственно полям `Re` и `Im` экземпляра `self` (экземпляр, который создается при вызове конструктора).

Если условие `type(x)==ComplNum` ложно, то мы неявно предполагаем, что `x` и `y` - числовые аргументы. В этом случае командами `self.Re=x` и `self.Im=y` значения аргументов присваиваются полям `Re` и `Im` создаваемого экземпляра класса.

На заметку

Таким образом, если первым (но не обязательно единственным аргументом) конструктору передана ссылка на экземпляр класса `ComplNum`, то второй аргумент конструктора в вычислениях не используется - вне зависимости от того, передан он явно или нет.

Также в классе `ComplNum` описан метод экземпляра `show()`, который отображает в области вывода значения полей экземпляра. Мы этот метод используем для проверки "содержимого" экземпляров класса `ComplNum`.

Вне кода класса `ComplNum` командой `a=ComplNum(1,2)` создается экземпляр `a` класса `ComplNum`. Копия этого экземпляра создается командой `b=ComplNum(a)`. Здесь аргументом конструктору экземпляра класса `ComplNum` передана ссылка на ранее созданный экземпляр `a`. Непосредственной проверкой убеждаемся, что экземпляр `b` действительно является копией экземпляра `a`: после создания экземпляра `b` у него такие же значения полей, как и у экземпляра `a`, а после изменения значений полей экземпляра `a` значения полей экземпляра `b` не меняются.

Стоит отметить, что даже при описании в классе конструктора создания копии необходимо помнить об особенностях копирования данных изменяемых типов - таких, как списки. Так, программный код, рассмотренный нами при иллюстрации принципов использования функций `copy()` и `deepcopy()` из модуля `copy`, мог бы быть реализован несколько иначе, прибегни мы к помощи конструктора создания копии. Рассмотрим листинг 6.16, в котором показано, как может быть описан конструктор, позволяющий создавать копии экземпляра класса с полями-списками.

Листинг 6.16. Конструктор создания копии и поля-списки

```
# Класс
class MyClass:
    # Конструктор
    def __init__(self,arg,nums=None):
        # Если аргумент arg - ссылка на
        # экземпляр класса MyClass
        if type(arg)==MyClass:
            # Значение поля name
            self.name=arg.name[:]
```

```
            # Значение поля nums
            self.nums=arg.nums[:]
        # Если аргумент arg - не ссылка на
        # экземпляр класса MyClass
        else:
            # Значение поля name
            self.name=arg
            # Значение поля nums
            self.nums=nums
    # Метод для отображения
    # значений полей экземпляра
    def show(self):
        # Поле name
        print("name ->",self.name)
        # Поле nums
        print("nums ->",self.nums)
# Создание экземпляра класса
x=MyClass("Python",[1,2,3])
print("Экземпляр x:")
# Отображение полей экземпляра x
x.show()
# Копия экземпляра x
y=MyClass(x)
# Отображение полей экземпляра y
y.show()
print("Поля экземпляра x изменяются!")
# Изменение значения поля name
# экземпляра x
x.name="Java"
# Изменение значения элемента в списке
# nums - поле экземпляра x
x.nums[0]=0
print("Экземпляр x:")
# Отображение полей экземпляра x
x.show()
print("Экземпляр y:")
# Отображение полей экземпляра y
y.show()
```

Ниже приведен результат выполнения этого программного кода:

Результат выполнения программы (из листинга 6.16)

```
Экземпляр x:
name -> Python
nums -> [1, 2, 3]
name -> Python
```

```
nums -> [1, 2, 3]
Поля экземпляра x изменяются!
Экземпляр x:
name -> Java
nums -> [0, 2, 3]
Экземпляр y:
name -> Python
nums -> [1, 2, 3]
```

Здесь мы при описании конструктора экземпляра класса `MyClass` предусмотрели возможность передать конструктору два аргумента (текст и список), или один аргумент, который является ссылкой на уже существующий экземпляр класса `MyClass` (для создания копии).

🕮 На заметку

Для аргумента `nums` в описании конструктора указано значение по умолчанию `None`, что соответствует пустой ссылке. В известном смысле это такой формальный примем, поскольку совсем не указать значение по умолчанию мы не могли: если для аргумента `nums` не указать значение по умолчанию, то при вызове конструктора создания копии пришлось бы передавать и второй, ничего не значащий аргумент. А это не очень удобно.

Вообще, чтобы оценить разницу в подходах, желающие могут сравнить программные коды в листингах 6.14-6.16.

В конструкторе использован такой же прием, что и в предыдущем примере: проверяется, совпадает ли тип аргумента `arg` с классом `MyClass`. Способ обработки аргументов конструктора зависит от истинности или ложности этого условия. Принципиальное отличие, по сравнению с предыдущим примером, состоит в том, что при присваивании значений полям выполняется копирование значения с использованием *среза* (инструкция `[:]` после имени переменной, ссылающейся на текст или список). В этом случае неявно предполагается, что тип аргументов позволяет выполнять подобную операцию.

🕮 На заметку

Напомним, что с помощью получения срез создается поверхностная копия списка. У читателя могут возникнуть вопросы относительно тех примеров, что рассматривались в листинге 6.14 и листинге 6.16. Попробуем их предугадать. Для этого детальнее рассмотрим, что происходит при создании копии экземпляра (листинг 6.14). Там с помощью функции `copy()` создавалась поверхностная копия `y` экземпляра `x` класса `MyClass`. У этого экземпляра `x` было поле-список `nums`. При создании нового экземпляра `y` у него тоже будет поле `nums`. Значение этого поля такое же, как значение поля `nums` исходного экземпляра `x`. Но на самом деле значение поля исходного экземпляра `x` - это ссылка на список (в данном случае `[1,2,3]`). Поэтому в новом экземпляре `y` поле `nums` будет ссылаться на тот же

самый список, на который ссылается поле nums экземпляра x. Если мы через экземпляр x и его поле nums меняем элемент в соответствующем списке, то изменения "заметит" и экземпляр y (поскольку его поле ссылается на тот же самый список). Но если мы присвоим другое значение полю nums экземпляра x, то значение поля nums экземпляра y останется неизменным! Обратите внимание - речь идет о присваивании значения полю nums, а не отдельному элементу списка nums: желающие могут в программном коде из листинга 6.14 команду `x.nums[0]=0` заменить на команду `x.nums=0` и проверить результат выполнения кода. Почему так происходит? Потому что при присваивании значения полю nums экземпляра x ссылка в этом поле просто перебрасывается на новые данные. При этом поле nums экземпляра y продолжает ссылаться на список.

В листинге 6.16 в конструкторе экземпляра класса при создании копии полей name и nums использована процедура получения среза. В этом случае создаются поверхностные копии полей name и nums исходного экземпляра, и ссылки на эти копии записываются в поля name и nums экземпляра-копии. В отношении поля nums речь идет о поверхностной копии списка. Но поскольку в данном случае поле-список nums не содержит в качестве элементов других списков, проблем не возникает. Они могли бы быть, если бы в исходном экземпляре x список nums содержал внутренние списки. Например, если в листинге 6.16 команду `x=MyClass("Python",[1,2,3])` заменить на команду `x=MyClass("Python",[[1,2],3])`, а команду `x.nums[0]=0` - на команду `x.nums[0][0]=0`, то изменения в поле nums экземпляра x отразятся и на поле nums экземпляра y.

Резюме

1. В языке Python поддерживается парадигма объектно-ориентированного программирования (ООП). Программные коды, написанные на языке Python, могут содержать и оперировать классами и экземплярами классов.
2. В общем смысле класс - это некоторый шаблон, по которому создаются объекты. Объект, в свою очередь, объединяет в одно целое данные и функции (или методы) для обработки этих данных.
3. Объекты, создаваемые на основе класса, называют экземплярами класса. Сам класс в Python также является объектом.
4. Переменные, объявленные в классе, называются полями класса. Переменные, связанные с экземплярами класса, называются полями экземпляра класса. Функции, описанные в классе, называют функциями класса. Функции, связанные с экземплярами класса, называются методами экземпляра класса. Поля и методы (функции) называются атрибутами (класса или экземпляра класса соответственно).
5. Описание класса начинается с ключевого слова `class`, после которого указывается имя класса. Методы экземпляра класса описываются в теле класса как обычные функции. При этом первый аргумент метода

(рекомендуемое название `self`) по умолчанию является ссылкой на экземпляр, из которого вызывается метод. При вызове метода из экземпляра класса после имени экземпляра через точку указывается имя метода, а в круглых скобках - аргументы. Ссылка на экземпляр класса, из которого вызывается метод, передается автоматически, так что при вызове метода у него на один аргумент меньше, чем это было при описании метода в теле класса.

6. Для создания экземпляра класса переменной экземпляра присваивается выражение, состоящее из имени класса и круглых скобок. В круглых скобках могут передаваться аргументы конструктору экземпляра класса.

7. Конструктор экземпляра класса - это метод, который автоматически вызывается при создании экземпляра класса. Конструктор имеет название `__init__()`. Также существует деструктор `__del__()`, но он на практике используется редко.

8. Атрибуты объекта класса и экземпляра класса уже после их создания можно менять (добавлять и удалять). Для удаления атрибутов можно использовать оператор `del`. Добавление атрибутов выполняется путем присваивания им значения.

9. Если у класса и у экземпляра класса есть поля с одинаковыми названиями, поле экземпляра класса "перекрывает" поле класса. Обращение к такому полю по имени (через ссылку на экземпляр класса) будет означать обращение к полю экземпляра класса.

10. При обращении к функции, описанной в теле класса, через ссылку на экземпляр класса, создается объект метода экземпляра класса, который и вызывается. Вызов данного метода эквивалентен вызову соответствующей функции класса с такими же аргументами, что и у метода, но еще с добавлением дополнительного первого аргумента - ссылки на экземпляр класса, из которого вызывается метод.

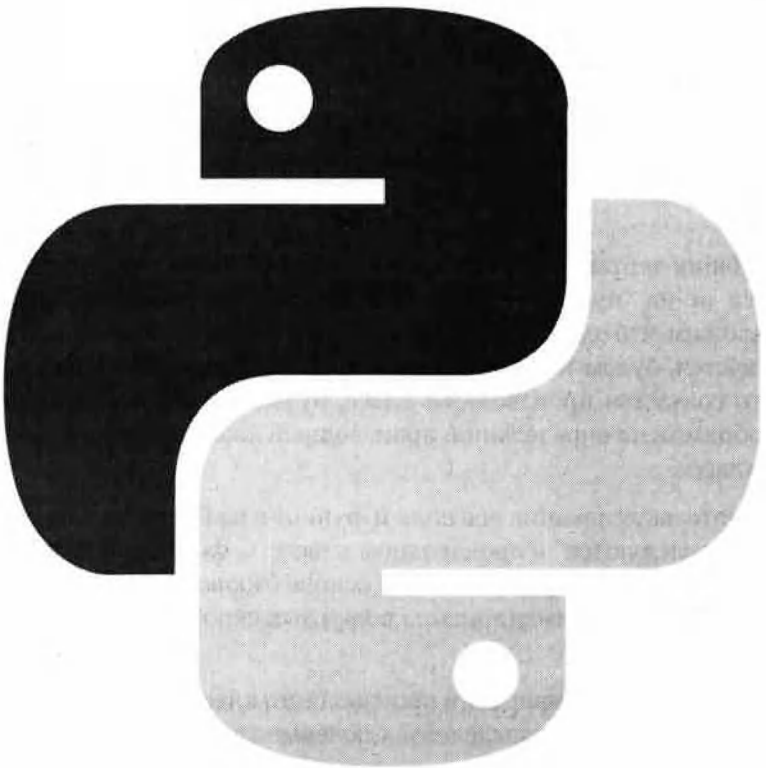
11. Имя функции, описанной в классе, является атрибутом этого класса. Как всякому атрибуту, этому может быть присвоено новое значение (имя функции), он может быть уделен или в класс может быть добавлен новый атрибут, ссылающийся на функцию.

12. Ссылка на функцию класса или метод экземпляра класса может быть записана в переменную, которая затем используется для вызова функции или метода.

13. Для создания копии экземпляра класса создают конструктор создания копии или используют функции `copy()` и `deepcopy()` из модуля `copy`.

Глава 7

Продолжаем знакомство с ООП



Лучший способ объяснить – это самому сделать.
Л. Кэрролл "Алиса в стране чудес"

Далее речь пойдет о некоторых достаточно нетривиальных механизмах, являющихся неотъемлемой частью ООП. В первую очередь это, конечно же, наследование. Кроме наследования, мы обсудим использование специальных методов и атрибутов при работе с классами, познакомимся с возможностями по переопределению операторов, а также затронем ряд других тем.

Наследование

Эврика! Царские шмотки! Одевайся, царем будешь!
из к/ф "Иван Васильевич меняет профессию"

Если в общих чертах, то идея *наследования* состоит в том, что новый класс создается не на "пустом месте", а на основе уже существующего класса. Предположим, что один класс создается на основе другого класса. Тот класс, что создается, будем называть *производным* классом. Тот класс, на основе которого создается производный класс, будем называть базовым *классом*. Таким образом, по определению производный класс создается на основе базового класса.

В результате наследования все поля и функции из базового класса неявным образом "наследуются" в производном классе. С формальной точки зрения чтобы создать производный класс на основе базового при описании производного класса после имени класса в круглых скобках указывается имя базового класса.

Другими словами при описании производного класса используем такой шаблон (жирным шрифтом выделены ключевые элементы шаблона):

```
class производный_класс(базовый_класс):
    # тело класса
```

Пример наследования проиллюстрирован программным кодом в листинге 7.1.

Листинг 7.1. Наследование классов

```
# Базовый класс
class BaseClass:
    # Поле базового класса
    name_base="Класс BaseClass"
    # Метод экземпляра базового класса
    def say_base(self):
        print("Метод say_base()")
# Производный класс
class NewClass(BaseClass):
    # Поле производного класса
    name_new="Класс NewClass"
    # Метод экземпляра производного класса
    def say_new(self):
        print("Метод say_new()")
# Экземпляр базового класса
obj_base=BaseClass()
# Экземпляр производного класса
obj_new=NewClass()
print("Класс BaseClass и экземпляр obj_base:")
# Поле базового класса
print(BaseClass.name_base)
# Метод экземпляра базового класса
obj_base.say_base()
print("\nКласс NewClass и экземпляр obj_new:")
# Поле производного класса
print(NewClass.name_base)
# Метод экземпляра производного класса
obj_new.say_base()
# Унаследованное из базового класса
# поле производного класса
print(NewClass.name_new)
# Унаследованный из базового класса
# метод экземпляра производного класса
obj_new.say_new()
```

В программном коде базовый класс называется `BaseClass`. Это самый обычный класс. Обстоятельство, что данный класс послужил базовым для создания нового класса, никак не отражено в классе `BaseClass`. В классе `BaseClass` описано поле `name_base` с текстовым значением `"Класс BaseClass"` и метод экземпляра с названием `say_base()`. У ме-

тода один аргумент и методом в окне вывода отображается простое сообщение.

Что касается производного класса, то он называется `NewClass`. При создании этого класса в круглых скобках после имени создаваемого производного класса указывается имя базового класса `BaseClass`. Непосредственно в теле класса `NewClass` описаны поле `name_new` с текстовым значением `"Класс NewClass"` и предназначенный для вывода сообщения метод экземпляра класса `say_new()` с одним аргументом. Но, поскольку класс `NewClass` создается на основе класса `BaseClass`, то "в наследство" от этого класса он получает поле `name_base` и метод `say_base()`. Таким образом, у класса `NewClass` два поля (`name_new` и `name_base`) и два метода (`say_new()` и `say_base()`).

Экземпляр базового класса создается командой `obj_base=BaseClass()`. Экземпляр производного класса создаем командой `obj_new=NewClass()`. Через класс `BaseClass` получаем доступ к полю `name_base`. Через экземпляр `obj_base` базового класса вызываем метод `say_base()`. У класса `NewClass`, как отмечалось, есть поля `name_base` и `name_new`, у экземпляра `obj_new` - методы `say_base()` и `say_new()`, которые, собственно, и вызываются. Результат выполнения описанного программного кода представлен ниже:

Результат выполнения программы (из листинга 7.1)

```
Класс BaseClass и экземпляр obj_base:
Класс BaseClass
Метод say_base()

Класс NewClass и экземпляр obj_new:
Класс BaseClass
Метод say_base()
Класс NewClass
Метод say_new()
```

Важно понимать, что при создании производного класса на основе базового класса последний используется не для того, чтобы просто создать на основе его атрибутов такие же атрибуты в производном класса: в производном классе используются те же самые атрибуты, что и в базовом классе. Поэтому если уже после создания производного класса изменить "содержимое" базового класса, эти изменения отразятся и на производном классе.

Пример такой ситуации проиллюстрирован в листинге 7.2.

Листинг 7.2. Изменение базового класса

```
# Базовый класс
class BaseClass:
    # Поле базового класса
    name="Поле name"
    # Метод экземпляра базового класса
    def say(self):
        print("Метод say()")
# Производный класс
class NewClass(BaseClass):
    pass
# Экземпляр производного класса
obj=NewClass()
# Поле производного класса
print(NewClass.name)
# Метод экземпляра производного класса
obj.say()
# Создаем функцию
def hello(self):
   print("Новый метод hello()")
# Изменяем ссылку на функцию
# в базовом классе
BaseClass.say=hello
# Изменяем значение поля
# в базовом классе
BaseClass.name="Новое значение поля name"
# Проверяем значение поля в производном классе
print(NewClass.name)
# Вызываем метод из экземпляра
# производного класса
obj.say()
```

Результат выполнения этого программного кода приведен ниже:

Результат выполнения программы (из листинга 7.2)

```
Поле name
Метод say()
Новое значение поля name
Новый метод hello()
```

В базовом классе `BaseClass` описано поле `name` со значением `"Поле name"` и метод `say()`, отображающий в окне вывода текст `"Метод say()"`. Производный класс `NewClass`, который создается на основе базового класса `BaseClass`, никакого дополнительного кода не содержит. Мы создаем экземпляр `obj` производного класса `NewClass`. Коман-

дами `print(NewClass.name)` и `obj.say()` проверяем, что поле `name` и метод `say()` успешно наследованы из базового класса. Затем определяется функция `hello()` и командой `BaseClass.say=hello` ссылка на эту функцию записывается в атрибут `say` базового класса `BaseClass`. Также командой `BaseClass.name="Новое значение поля name"` изменяем значение поля `name` в базовом классе. После этого проверяем значение поля `name` в производном классе с помощью команды `print(NewClass.name)`, а также вызываем метод `say()` из экземпляра производного класса с помощью команды `obj.say()`. Как видим по результату выполнения этих команд, в производном классе использованы новые значения атрибутов базового класса.

При наследовании существует возможность *переопределять* поля и методы, наследованные в производном классе из базового. Общая идея состоит в том, что поле или метод, которые наследуются из базового класса, в производном классе создаются заново. Как это делается на практике, иллюстрирует программный код из листинга 7.3.

Листинг 7.3. Переопределение полей и методов

```
# Базовый класс
class BaseClass:
    # Поле базового класса
    name="Поле name базового класса"
    # Метод экземпляра базового класса
    def say(self):
        print("Метод say() базового класса")
# Производный класс
class NewClass(BaseClass):
    # Поле производного класса
    name="Поле name производного класса"
    # Метод экземпляра производного класса
    def say(self):
        print("Метод say() производного класса")
# Экземпляр базового класса
obj_base=BaseClass()
# Экземпляр производного класса
obj_new=NewClass()
# Атрибуты экземпляра базового класса
print(obj_base.name)
obj_base.say()
# Атрибуты экземпляра производного класса
print(obj_new.name)
obj_new.say()
```

При выполнении данного программного кода получаем следующий результат:

Результат выполнения программы (из листинга 7.3)

```
Поле name базового класса
Метод say() базового класса
Поле name производного класса
Метод say() производного класса
```

В этом примере мы сначала создали базовый класс `BaseClass` с полем `name` и методом `say()`, а затем на основе этого класса создали производный класс `NewClass`. И хотя при наследовании поле `name` и метод `say()` становятся доступными и в производном классе, мы все равно в классе `NewClass` описываем поле `name` и метод `say()`, но уже другие (по сравнению с базовым классом). В результате экземпляр `obj_base` базового класса ссылается на поле и метод из базового класса, а экземпляр `obj_new` производного класса ссылается на поле и метод производного класса.

Обычно на практике прибегают к переопределению методов. Причем ситуация может быть далеко не такой простой, как было показано выше. Еще один пример переопределения методов при наследовании приведен в листинге 7.4.

Листинг 7.4. Переопределение методов

```
# Базовый класс
class BaseClass:
    def __init__(self,num):
        self.id=num
    def get(self):
        print("ID:",self.id)
    # Метод экземпляра базового класса
    def show(self):
        print("Поле экземпляра базового класса")
        self.get()
# Производный класс
class NewClass(BaseClass):
    def __init__(self,num,txt):
        super().__init__(num)
        self.name=txt
    def get(self):
        super().get()
        print("Name:",self.name)
# Экземпляр базового класса
obj_base=BaseClass(1)
```

```
print("Вызываем метод show() из экземпляра obj_base:")
# Вызов метода экземпляра базового класса
obj_base.show()
# Экземпляр производного класса
obj_new=NewClass(10,"десятка")
print("Вызываем метод show() из экземпляра obj_new:")
# Вызов метода экземпляра производного класса
obj_new.show()
```

Прежде, чем посмотреть, каков же будет результат выполнения этого программного кода, кратко проанализируем его.

Базовый класс `BaseClass` содержит конструктор, в котором полю `id` экземпляра класса присваивается значение (передается аргументом конструктору). Метод экземпляра `get()` отображает в окне вывода значение поля `id` экземпляра, из которого вызывается метод. Еще в классе `BaseClass` описан метод экземпляра `show()`, в котором командой `print("Поле экземпляра базового класса")` в окне вывода отображается сообщение, а затем командой `self.get()` вызывается метод экземпляра `get()` (который, напомним, отображает значение поля `id` экземпляра класса).

Производный класс `NewClass` создается на основе базового класса `BaseClass`. В данном классе много новых для нас (и, возможно, непонятных) инструкций. В первую очередь это конструктор. Мы предполагаем, что у экземпляра производного класса будет два поля: поле `id` и поле `name`. Необходимо, чтобы этим полям в конструкторе присваивались значения, и, соответственно, значения этих полей должны передаваться аргументами конструктору.

Наша идея состоит в том, чтобы в конструкторе экземпляра производного класса вызвать конструктор экземпляра базового класса. Проблема усугубляется обстоятельством, что все конструкторы во всех классах называются одинаково. Чтобы вызвать "правильный" конструктор, нам понадобится функция `super()`. Функция возвращает в качестве результата специальный прокси-объект, который делегирует вызов метода базовому классу. С практической точки зрения на эту функцию можно смотреть как на заместитель экземпляра базового класса, из которого получается вызвать исходную (определенную в базовом классе) версию метода - в том числе это касается и конструктора.

🕮 На заметку

Как бы там ни было, а вызвать конструктор базового класса можно инструкцией вида `super().__init__(аргументы)`. В круглых скобках для конструктора указываются `аргументы`, причем без ссылки на экземпляр класса, из которого вызывается конструктор.

Командой `super().__init__(num)` в теле конструктора экземпляра производного класса вызывается конструктор экземпляра базового класса. В результате поле `id` получит свое значение. Затем командой `self.name=txt` значение присваивается полю `name` экземпляра производного класса (переменные `self`, `num` и `txt` обозначают аргументы конструктора экземпляра производного класса).

В производном классе переопределяется метод `get()`. В теле метода командой `super().get()` вызывается версия метода из базового класса. При выполнении этого метода, напомним, в окне вывода отображается значение поля `id` экземпляра класса, из которого вызывается метод. Затем командой `print("Name:",self.name)` отображается значение поля `name` производного класса.

На заметку

Если имеется некоторый `метод`, который определен в базовом классе и переопределяется в производном классе, и нам необходимо в теле экземпляра производного класса вызвать эту исходную, базовую версию `метода` (а такое возможно), используем инструкцию вида `super().метод(аргументы)`.

Есть альтернатива к использованию функции `super()`. Например, если нам в производном классе нужно вызвать версию метода экземпляра из базового класса, можем выполнить явную ссылку на базовый класс, воспользовавшись командой вида `базовый_класс.метод(self,аргументы)` - это вместо команды `super().метод(аргументы)`. Например, команда `super().get()` могла бы выглядеть как `BaseClass.get(self)`.

Важный момент, на котором следует остановиться, связан с методом `show()`, который описан в базовом классе, а в производном не переопределяется. Поэтому производный класс наследует версию метода `show()` из базового класса. Но в методе `show()` (в базовом классе) вызывается метод `get()`, который, в свою очередь, переопределяется в производном классе. При вызове метода `show()` из экземпляра производного класса будет вызываться метод `get()`. Вопрос в том, какой именно это будет метод `get()` - из базового класса или из производного? Ответ на этот вопрос мы получим из результата выполнения программного кода.

Экземпляр базового класса создается командой `obj_base=BaseClass(1)`. Затем вызываем метод `show()` из экземпляра `obj_base` базового класса (команда `obj_base.show()`).

Экземпляр производного класса создается командой `obj_new=NewClass(10,"десятка")`. Командой `obj_new.show()` вызывается метод `show()` из экземпляра `obj_new` производного класса. Результат выполнения всего программного кода такой:

Результат выполнения программы (из листинга 7.4)

```
Вызываем метод show() из экземпляра obj_base:
Поле экземпляра базового класса
ID: 1
Вызываем метод show() из экземпляра obj_new:
Поле экземпляра базового класса
ID: 10
Name: десятка
```

Особый интерес представляют последние три строки в окне вывода: это результат выполнения команды `obj_new.show()`. Несложно понять, что такой результат может быть получен, если в теле наследованного из базового класса метода `show()` вызывается переопределенный в производном классе метод `get()`. Причем это общее правило: если в наследуемом методе есть команда вызова другого метода и этот другой метод переопределяется в производном классе, то будет вызвана эта новая (переопределенная) версия метода.

На заметку

В языке C++ такое свойство называется *виртувльностью*. Выражаясь терминологией языка C++, в Python все методы являются *виртуальными*.

До этого при создании производного класса базовый класс был всегда один. На самом деле в Python производный класс можно создавать на основе сразу нескольких базовых классов. Это называется *множественным наследованием*. При множественном наследовании у производного класса не один базовый класс, а несколько базовых классов. Производный класс наследует все атрибуты каждого из своих базовых классов. Небольшой пример с множественным наследованием представлен в листинге 7.5.

На заметку

В этом примере мы как бы условно "описываем" некоторую коробку или ящик. У этого ящика есть геометрические размеры (ширина, высота, глубина), масса (или вес - хотя, если честно, это далеко не одно и то же), а еще цвет. На основе линейных размеров вычисляется объем (произведение ширины, высоты и глубины). Все единицы измерений безразмерные.

В классе `BoxSize` предусмотрена возможность записывать линейные размеры и вычислять объем. Класс `BoxParams` предусматривает возможность запоминания таких параметров, как вес и цвет. На основе классов `BoxSize` и `BoxParams` путем наследования создается класс `Box`.

Листинг 7.5. Наследование нескольких базовых классов

```
# Первый базовый класс
class BoxSize:
    # Конструктор
    def __init__(self,width,height,depth):
        # Присваивание значений полям экземпляра
        self.width=width
        self.height=height
        self.depth=depth
    # Метод для вычисления объема
    def volume(self):
        # Результат - произведение полей экземпляра
        return self.width*self.height*self.depth
    # Метод для отображения значений полей экземпляра
    # и результата вызова метода volume()
    def show(self):
        # Поля экземпляра класса
        print("Размеры и объем ящика:")
        print("Ширина:",self.width)
        print("Высота:",self.height)
        print("Глубина:",self.depth)
        # Результат вызова метода volume()
        print("Объем:",self.volume())
# Второй базовый класс
class BoxParams:
    # Конструктор
    def __init__(self,weight,color):
        # Присваивание значений полям экземпляра
        self.weight=weight
        self.color=color
    # Метод для отображения значений полей экземпляра
    def show(self):
        # Отображение значений полей
        print("Дополнительные параметры ящика:")
        print("Вес (масса):",self.weight)
        print("Цвет:",self.color)
# Производный класс
class Box(BoxSize,BoxParams):
    # Конструктор
    def __init__(self,width,height,depth,weight,color):
        # Вызов конструктора первого базового класса
        BoxSize.__init__(self,weight,height,depth)
        # Вызов конструктора второго базового класса
        BoxParams.__init__(self,weight,color)
        # Вызов метода show() экземпляра класса
```

```
        self.show()
    # Переопределение метода show()
    def show(self):
        # Вызов метода show() из первого базового класса
        BoxSize.show(self)
        # Вызов метода show() из второго базового класса
        BoxParams.show(self)
# Создаем экземпляр производного класса
obj=Box(10,20,30,5,"зеленый")
```

В базовом классе `BoxSize` описан конструктор, в котором полям `width`, `height` и `depth` экземпляра класса присваиваются значения (тем самым при создании экземпляра класса этому экземпляру добавляются соответствующие поля).

На заметку

Обратите внимание, что аргументы конструктора `width`, `height`, `depth` и поля экземпляра класса называются одинаково. Если в теле конструктора мы просто пишем название `width`, `height` или `depth`, то имеются в виду аргументы конструктора. Для ссылки на поля экземпляра класса используем инструкции `self.width`, `self.height` и `self.depth` с явным указанием аргумента конструктора `self`, отождествляемого со ссылкой на экземпляр класса.

Для тех, кто знаком с C++, Java или C# подчеркнем: в Python нет сокращенной формы обращения к полям экземпляра класса.

Также в классе `BoxSize` описывается метод `volume()`, предназначенный для вычисления объема: результатом метода является произведение `self.width*self.height*self.depth` значений всех трех полей экземпляра.

Метод `show()` содержит команды по отображению в окне вывода значений полей экземпляра и результата вызова метода `volume()`.

В базовом классе `BoxParams` в конструкторе задаются значения двух полей, а метод `show()` в этом классе описан так, что при его вызове отображаются значения этих полей.

Производный класс `Box` создается на основе базовых классов `BoxSize` и `BoxParams`: названия этих классов указываются в круглых скобках после имени производного класса. В производном классе описывается конструктор, у которого достаточно много аргументов.

В теле конструктора последовательно вызываются конструкторы базовых классов: командой `BoxSize.__init__(self,weight,height,depth)` вызывается конструктор экземпляра класса `BoxSize`, а командой `BoxParams.__init__(self,weight,color)` вызывается конструктор экземпляра класса `BoxParams`. И в том и в другом случае конструктор вы-

зывается как функция класса с явным указанием имени класса и передачей первым аргументом конструктору ссылки `self` на экземпляр класса.

Также в конструкторе вызывается метод `show()`. Но здесь мы теоретически должны были бы столкнуться с проблемой: в каждом из базовых классов `BoxSize` и `BoxParams` есть метод с названием `show()`, и каждый из этих методов наследуется в производном классе `Box`. В принципе, в такой ситуации по умолчанию при обращении к методу `show()` вызывалась бы версия метода из класса `BoxSize`, поскольку в списке базовых классов этот класс указан первым.

На заметку

В экземпляре производного класса при обращении к полю или методу, если это поле или метод не описаны в теле производного класса, поиск начинается в базовых классах, причем строго в соответствии с тем порядком, в котором классы указаны в списке наследования (в круглых скобках после имени производного класса).

Но мы идем другим путем - путем переопределения метода `show()` в производном классе `Box`. В теле метода в классе `Box` командами `BoxSize.show()` и `BoxParams.show()` последовательно вызываются версии этого метода соответственно из класса `BoxSize` и `BoxParams`. Здесь действует тот же принцип, что и при переопределении конструктора класса `Box`: методы вызываются как функции класса через ссылку на объект класса и передачей аргументом ссылки на экземпляр класса.

Помимо описания классов, в программном коде есть всего одна команда `obj=Box(10,20,30,5,"зеленый")`, которой создается экземпляр производного класса `Box`. при этом вызывается конструктор, а в конструкторе вызывается метод `show()`, которым отображаются значения всех полей экземпляра `obj`. Так что результат получаем следующий:

Результат выполнения программы (из листинга 7.5)

```
Размеры и объем ящика:
Ширина: 5
Высота: 20
Глубина: 30
Объем: 3000
Дополнительные параметры ящика:
Вес (масса): 5
Цвет: зеленый
```

Кроме множественного наследования может использоваться *многократное наследование*. При многократном наследовании производный класс являет-

ся базовым для другого класса. Если учесть, что одновременно допустимо использовать и множественное наследование, становится очевидным, что структура "родственных отношений" между классами может быть очень сложной.

На заметку

Часто все сводится к определению того, как при многократном и одновременно множественном наследовании производным классом наследуются из разных классов атрибуты (методы/функции) с одинаковыми названиями. Речь идет вот о чем: представим, что из экземпляра производного класса вызывается метод или из производного класса вызывается функция, а в структуре наследования классов есть методы/функции с таким же именем. Вопрос в том, какая версия (из какого класса) метода или функции будет вызываться? В общем упрощенном виде ответ на этот вопрос сводится к тому, что последовательно перебирается цепочка наследования классов до первого "совпадения". Базовые классы перебираются в том порядке, как они указаны при создании производного класса. Причем базовые классы "просматриваются" вместе со своими базовыми классами. Хотя и тут не так все просто. Неприятности возникают в случае, если один и тот же класс оказывается "разными путями", прямо или опосредованно, несколько раз среди наследуемых классов. Например, класс `A` является базовым для классов `B` и `C`, а на их основе создается класс `D`. В результате получается, что класс `D` наследует класс `A` "дважды": через класс `B` и через класс `C`. А могут быть и более трагичные "хитро-

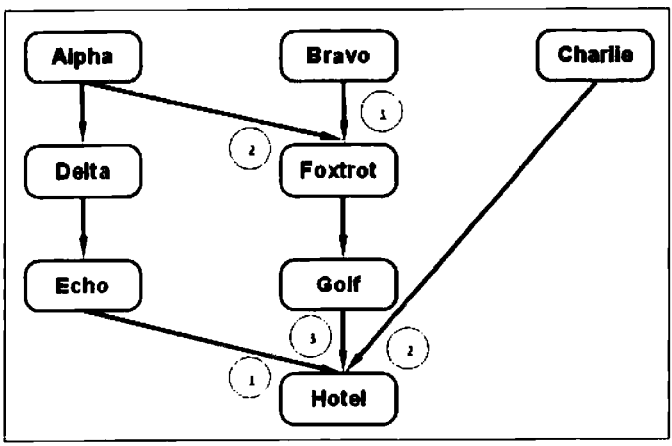


Рис. 7.1. Схема наследования классов. Наследование изображается направленной стрелкой (от базового класса к производному классу). Цифрами обозначен порядок наследования классов в случае множественного наследования. Классы с описанной функцией hi() выделены темным цветом

сплетения". В подобных случаях в Python применяется специальный алгоритм, которым определяется цепочка наследования классов, причем в этой цепочке каждый класс встречается только единожды.

Мы по этому поводу рассмотрим небольшой пример, представленный в листинге 7.6. В приведенном там программном коде описаны несколько классов, причем одни из них наследуют другие. Общая схема наследования классов представлена на рис. 7.1.

Три базовых класса (`Alpha`, `Bravo` и `Charlie`) содержат описание функции `hi()`, а остальные классы наследуют (различными путями) эти классы. Программный код такой:

Листинг 7.6. Многократное наследование

```
# Описываем классы
class Alpha:
    def hi():
        print("Класс Alpha")
class Bravo:
    def hi():
        print("Класс Bravo")
class Charlie:
    def hi():
        print("Класс Charlie")
class Delta(Alpha):
    pass
class Echo(Delta):
    pass
class Foxtrot(Bravo,Alpha):
    pass
class Golf(Foxtrot):
    pass
class Hotel(Echo,Charlie,Golf):
    pass
# Вызываем функции классов
Echo.hi()
Golf.hi()
Hotel.hi()
```

Результат выполнения программного кода следующий:

Результат выполнения программы (из листинга 7.6)

```
Класс Alpha
Класс Bravo
Класс Charlie
```

После описания классов в программном коде последовательно выполняются команды `Echo.hi()`, `Golf.hi()` и `Hotel.hi()`. Разберем, что происходит при выполнении каждой из этих команд.

При выполнении команды `Echo.hi()` выполняется поиск функции `hi()` в теле класса `Echo`. Непосредственно в этом классе функция не описана. Но класс `Echo` создается на основе класса `Delta`. В этом классе функция с именем `hi()` также не описана, поэтому поиск функции выполняется в классе `Alpha` - базовом классе для класса `Delta`. В результате вызывается функция из класса `Alpha` и в окне вывода появляется сообщение `Класс Alpha`. Здесь ситуация простая, поскольку простая цепочка наследования: Класс `Echo` наследует класс `Delta`, а класс `Delta` наследует класс `Alpha`. Поиск функции `hi()` осуществляется в соответствии с этой цепочкой наследования.

При выполнении команды `Golf.hi()` поиск функции `hi()` сначала происходит в теле класса `Golf`. Но там такая функция не описывалась. Поскольку класс `Golf` создавался на основе класса `Foxtrot`, поиск функции `hi()` выполняется в этом классе. В теле этого класса функция `hi()` также не описана. Поиск продолжается в базовых классах для класса `Foxtrot`: это классы `Bravo` и `Alpha` (в том порядке, как они указаны при наследовании). Так как в классе `Bravo` есть описание функции `hi()`, она и вызывается, в результате чего в окне вывода появляется сообщение `Класс Bravo`. Если бы в классе `Bravo` функции `hi()` не оказалось, поиск бы продолжился в классе `Alpha`.

Более сложная ситуация с командой `Hotel.hi()`. По логике для поиска функции `hi()` классы должны были бы проверяться в такой последовательности: сначала `Hotel`, `Echo`, `Delta`, `Alpha`, затем `Charlie`, а далее `Golf`, `Foxtrot`, `Bravo` и `Alpha` - пока не будет найден первый класс в этой последовательности, в котором описана функция `hi()`. В этой цепочке первый класс, в котором описана функция `hi()`, - класс `Alpha`. Поэтому если бы все так и было, появилось бы сообщение `Класс Alpha`. Но на самом деле появляется сообщение `Класс Charlie`. Почему так?

На самом деле полный и исчерпывающий ответ сводился бы к описанию алгоритма, используемого в Python при формировании цепочки наследования, когда одни и те же классы многократно наследуются в производном классе. Нам такие "технические" подробности вряд ли понадобятся, поэтому мы ограничимся общей идеей с апелляцией к нашему примеру. Итак, в формальной последовательности наследуемых классов класс `Alpha` встречается дважды. А нужно, чтобы он там встречался всего один раз. Причем важна позиция, на которой этот класс будет находиться в цепочке наследо-

вания (поскольку она определяет, в какой последовательности просматриваются классы при поиске функции `hi()`).

Базовый принцип состоит в том, чтобы избегать ситуаций, когда сначала просматривается базовый класс, а затем производный от него. В данном конкретном случае теоретически имеется два варианта цепочки наследования для класса `Hotel`: `Echo`, `Delta`, `Alpha`, `Charlie`, `Golf`, `Foxtrot`, `Bravo` или `Echo`, `Delta`, `Charlie`, `Golf`, `Foxtrot`, `Bravo`, `Alpha`. В первом случае класс `Alpha` просматривается перед классом `Foxtrot`, для которого класс `Alpha` является базовым.

Как мы упоминали выше, это плохой вариант. Во втором случае такой проблемы нет. Поэтому цепочка наследования классов для класса `Hotel` на самом деле выглядит так: `Echo`, `Delta`, `Charlie`, `Golf`, `Foxtrot`, `Bravo`, `Alpha`. Двигаясь по этой цепочки при поиске кода функции `hi()` (речь, напомним, идет о выполнении команды `Hotel.hi()`) "останавливаемся" на классе `Charlie`, в котором эта функция описана.

На заметку

Чтобы увидеть "цепочку наследования" для производного класса, разумно воспользоваться полем `__mro__` этого класса. В частности, если в рассмотренном выше примере добавить команду `print(Hotel.__mro__)`, увидим, в какой последовательности наследуются классы в классе `Hotel`.

Вообще, несложно придумать такую схему наследования классов, которая является недопустимой в том смысле, что не позволяет установить последовательность или цепочку наследования классов. Например, класс `C` наследует классы `A` и `B`, класс `D` наследует классы `B` и `A`, а класс `E` наследует классы `C` и `D`. В такой ситуации возникает ошибка типа `TypeError`, связанная с невозможностью установить цепочку наследования.

В следующем разделе речь пойдет о некоторых *специальных методах и полях*, которые позволяют существенно расширить возможности классов и экземпляров классов в Python.

Специальные методы и поля

Живьем брать демонов!
из к/ф "Иван Васильевич меняет профессию"

В Python есть группа методов, названия которых начинаются и заканчиваются двойным подчеркиванием. Такие методы предназначены для работы с классами и экземплярами классов, и позволяют выполнять некоторые весьма специфичные операции. Поэтому такие методы обычно называют *специальными*.

На заметку

По крайней мере, с двумя специальными методами мы уже знакомы: это конструктор `__init__()` и деструктор `__del__()`.

Также есть группа полей, которые в названии в начале и конце содержат по два символа подчеркивания. Эти поля доступны для каждого класса и/или экземпляра по умолчанию.

На заметку

На первый взгляд может показаться странным, что у классов, которые создаются пользователем, есть атрибуты, которые пользователем не описаны. Но здесь следует учесть, что начиная с версии Python 3 все классы (в том числе и созданные пользователем), в конечном счете, являются наследниками класса `object`. Другими словами, существует иерархия классов, и в вершине этой иерархии находится класс `object`.

В таблице 7.1 перечислены некоторые специальные поля, которые могут быть полезны при работе с классами.

Таблица 7.1. Специальные поля

Поле	Назначение
`__bases__`	Возвращается список базовых классов
`__dict__`	Возвращается словарь с атрибутами класса
`__doc__`	Возвращается текст документирования класса (текст, описывающий класс). Текст документирования можно непосредственно указать в теле класса первой строкой или присвоить полю значение
`__module__`	Возвращается модуль класса
`__mro__`	Возвращается цепочка наследования класса
`__name__`	Возвращается имя класса
`__qualname__`	Возвращается полное имя класса (в точечном формате, отображающем структуру вложенных классов)

На заметку

У экземпляров класса есть специальное поле `__class__`, которое позволяет определить класс, на основе которого создавался экземпляр. Поскольку класс сам является объектом, то у класса такое поле тоже есть. Поле `__class__` для объекта класса дает (при выводе значения функцией `print()`) значение `<class 'type'>`, что означает принадлежность объекта класса к типу `type`.

Также есть очень полезная функция `dir()`, которая позволяет получить список атрибутов (полей и методов) экземпляра класса. При вызове функции экземпляр класса передается аргументом функции.

Небольшой пример использования перечисленных полей приведен в листинге 7.7.

Листинг 7.7. Специальные поля

```
# Класс
class Alpha:
    "Класс Alpha и внутренний класс Bravo"
    def hello():
        pass
    # Внутренний класс
    class Bravo:
        pass
# Производный от Alpha класс
class Charlie(Alpha):
    pass
# Производный от Charlie класс
class Delta(Charlie):
    pass
# Создаем экземпляр класса
obj=Alpha()
# Поле __class__ экземпляра класса
print("Поле __class__ ")
print("Экземпляр obj:",obj.__class__)
# Поле __class__ класса
print("Класс Alpha:",Alpha.__class__)
print("Класс Alpha.Bravo:",Alpha.Bravo.__class__)
print("Класс Charlie:",Charlie.__class__)
# Поля __bases__ и __mro__
print("\nПоля __bases__ и __mro__")
print("Класс Delta, поле __bases__:",Delta.__bases__)
print("Класс Delta, поле __mro__:",Delta.__mro__)
print("Класс Alpha, поле __bases__:",Alpha.__bases__)
print("Класс Alpha, поле __mro__:",Alpha.__mro__)
# Поле __doc__
```

```
print("\nПоле __doc__")
print("Описание класса Alpha:",Alpha.__doc__)
Delta.__doc__="Класс Delta наследует класс Charlie"
print("Описание класса Delta:",Delta.__doc__)
# Поле __module__
print("\nПоле __module__")
print("Модуль класса Alpha:",Alpha.__module__)
# Поле __dict__
print("\nПоле __dict__")
print("Атрибуты класса Alpha:",Alpha.__dict__)
print("Атрибуты класса Alpha.Bravo:",Alpha.Bravo.__dict__)
print("Атрибуты класса Delta:",Delta.__dict__)
# Поля __name__ и __qualname__
print("\nПоля __name__ и __qualname__")
print("Класс Alpha, поле __name__:",Alpha.__name__)
print("Класс Alpha, поле __qualname__:",Alpha.__qualname__)
print("Класс Alpha.Bravo, поле __name__:",Alpha.Bravo.__
name__)
print("Класс Alpha.Bravo, поле __qualname__:",Alpha.Bravo.__
qualname__)
print("Класс Delta, поле __name__:",Delta.__name__)
print("Класс Delta, поле __qualname__:",Delta.__qualname__)
```

Результат выполнения этого программного кода приведен ниже:

Результат выполнения программы (из листинга 7.7)

```
Поле __class__
Экземпляр obj: <class '__main__.Alpha'>
Класс Alpha: <class 'type'>
Класс Alpha.Bravo: <class 'type'>
Класс Charlie: <class 'type'>

Поля __bases__ и __mro__
Класс Delta, поле __bases__: (<class '__main__.Charlie'>,)
Класс Delta, поле __mro__: (<class '__main__.Delta'>, <class
'__main__.Charlie'>, <class '__main__.Alpha'>, <class
'object'>)
Класс Alpha, поле __bases__: (<class 'object'>,)
Класс Alpha, поле __mro__: (<class '__main__.Alpha'>, <class
'object'>)

Поле __doc__
Описание класса Alpha: Класс Alpha и внутренний класс Bravo
Описание класса Delta: Класс Delta наследует класс Charlie
```

```
Поле __module__
Модуль класса Alpha: __main__

Поле __dict__
Атрибуты класса Alpha: {'__dict__': <attribute '__dict__' of
'Alpha' objects>, 'Bravo': <class '__main__.Alpha.Bravo'>,
'__weakref__': <attribute '__weakref__' of 'Alpha' objects>,
'hello': <function Alpha.hello at 0x023201E0>, '__module__':
'__main__', '__doc__': 'Класс Alpha и внутренний класс Bravo'}
Атрибуты класса Alpha.Bravo: {'__weakref__': <attribute '__
weakref__' of 'Bravo' objects>, '__dict__': <attribute '__
dict__' of 'Bravo' objects>, '__doc__': None, '__module__':
'__main__'}
Атрибуты класса Delta: {'__doc__': 'Класс Delta наследует
класс Charlie', '__module__': '__main__'}

Поля __name__ и __qualname__
Класс Alpha, поле __name__: Alpha
Класс Alpha, поле __qualname__: Alpha
Класс Alpha.Bravo, поле __name__: Bravo
Класс Alpha.Bravo, поле __qualname__: Alpha.Bravo
Класс Delta, поле __name__: Delta
Класс Delta, поле __qualname__: Delta
```

На заметку

Некоторые поля (а именно, `__bases__` и `__mro__`) в качестве значений возвращают кортежи (набор элементов в круглых скобках). Если кортеж состоит из одного элемента, по правилам синтаксиса Python после этого элемента стоит запятая, хотя следующего элемента в кортеже нет.

Далее мы рассмотрим специальные методы. Начнем в качестве иллюстрации с метода `__call__()`, который предназначен для выполнения достаточно благородной миссии: если в классе описан этот метод, то экземпляр класса можно вызывать как функцию.

Небольшой пример, иллюстрирующий, как определяется метод `__call__()`, представлен в листинге 7.8.

Листинг 7.8. Определение метода `__call__()`

```
# Класс
class Box:
    # Конструктор
    def __init__(self,width,height,depth):
        # Поля экземпляра класса
```

```
        self.width=width
        self.height=height
        self.depth=depth
    # Определение метода __call__()
    def __call__(self):
        # Вычисляется произведение полей экземпляра
        volume=self.width*self.height*self.depth
        # Отображается результат вычислений
        print("Объем равен",volume)
# Создаем экземпляр класса
obj=Box(10,20,30)
# Вызываем экземпляр класса как функцию
obj()
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 7.8)

```
Объем равен 6000
```

В этом примере мы эксплуатируем идею с классом, который описывает некоторый "ящик" или "коробку" (в геометрическом смысле этого понятия). Класс называется `Box`. В классе описан конструктор, при выполнении которого полям `width`, `height` и `depth` присваиваются значения (аргументы конструктора).

Для вычисления объема, вместо того, чтобы описывать метод экземпляра класса, определяем метод `__call__()`. Поскольку метод будет вызываться (неявно) через экземпляр класса, аргументом метода указана ссылка `self` на экземпляр класса. В теле класса командой `volume=self.width*self.height*self.depth` вычисляется произведение полей экземпляра класса (то есть объем "коробки") и записывается в локальную переменную `volume`, после чего значение этой переменной отображается в окне вывода. На этом описание класса `Box` заканчивается.

Вне программного кода класса `Box` создается экземпляр `obj` этого класса, а затем следует команда `obj()` - то есть мы обращаемся к экземпляру класса так, как если бы это была функция. Поскольку в теле класса `Box` определен метод `__call__()`, то автоматически вызывается этот метод. В результате в окне вывода появляется сообщение с информацией об объеме "коробки".

На заметку

Выражаясь более "технологичным" языком, команда `obj()` обрабатывается как команда `Box.__call__(obj)`.

Можно определить метод `__call__()` так, чтобы экземпляр класса не просто "вызывался", а "вызывался" с аргументами. Несложно догадаться, что для этого достаточно описать метод `__call__()` с дополнительными (кроме ссылки на экземпляр класса) аргументами. Пример такой ситуации приведен в листинге 7.9.

Листинг 7.9. Метод __call__() с аргументами

```
# Класс
class Box:
    # Конструктор
    def __init__(self,width,height,depth):
        print("Объем равен",self(width,height,depth))
    # Определение метода __call__()
    def __call__(self,width,height,depth):
        # Поля экземпляра класса
        self.width=width
        self.height=height
        self.depth=depth
        # Вычисляется произведение полей экземпляра
        volume=self.width*self.height*self.depth
        # Возвращаемый результат
        return volume
# Создаем экземпляр класса
obj=Box(10,20,30)
# Вызываем экземпляр как функцию
print("Новое значение",obj(1,2,3))
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 7.9)

```
Объем равен 6000
Новое значение 6
```

В данном случае мы так определяем метод `__call__()`, что при вызове экземпляра класса (в формате обращения к функции) необходимо передать три аргумента, которые определят значения полей экземпляра класса. Более того, теперь еще и возвращается результат - это объем (произведение полей экземпляра класса). Существенно изменился и программный код конструктора. Более конкретно, в теле конструктора выполняется всего одна команда `print("Объем равен",self(width,height,depth))`, в которой есть инструкция `self(width,height,depth)` вызова экземпляра класса `self` с аргументами `width`, `height` и `depth`.

Таким образом, неявный вызов метода `__call__()` используется уже в конструкторе - то есть при создании экземпляра класса. Поскольку

при вызове экземпляра класса возвращается результат, то у выражения `self(width,height,depth)` есть числовое значение - произведение полей экземпляра класса. Это значение будет отображаться в окне вывода при создании экземпляра класса.

Что касается непосредственно программного кода метода `__call__()`, то он описан с четырьмя аргументами: аргумент `self` представляет ссылку на экземпляр класса, а аргументы `width`, `height` и `depth` задают значения полей экземпляра. В теле метода командами `self.width=width`, `self.height=height` и `self.depth=depth` полям присваиваются значения, затем командой `volume=self.width*self.height*self.depth` вычисляется произведение полей, и, наконец, инструкцией `return volume` это значение возвращается как результат метода `__call__()`.

Поэтому когда командой `obj=Box(10,20,30)` создается экземпляр класса, то при вызове конструктора полям `width`, `height` и `depth` экземпляра присваиваются значения `10`, `20` и `30` соответственно, вычисляется произведение этих полей и отображается сообщение. При выполнении команды `print("Новое значение",obj(1,2,3))` указанные поля получают значения `1`, `2` и `3`, вычисляется новое значение для объема "коробки" и это значение отображается в окне вывода.

Помимо метода `__call__()`, существуют и другие полезные специальные методы, которые условно, в зависимости от их, так сказать, назначения, можно разбить на группы. Так, есть достаточно много методов, предназначенных для преобразования экземпляров класса к базовым типам - таким, например, как целые числа или текст. В таблице 7.2 представлены некоторые из таких методов. При описании все они имеют один аргумент - ссылку на экземпляр класса (это, фактически, тот экземпляр, который нужно преобразовать к базовому типу). Явно эти методы обычно не вызываются. При каких обстоятельствах они "вступают в игру", описано в таблице.

Таблица 7.2. Специальные методы приведения к типу

Метод	Описание
`__bool__()`	Метод для приведения экземпляра к логическому типу (тип `bool`). Вызывается при использовании функции `bool()` или в иных случаях, когда выполняется автоматическое приведение к логическому типу (например, когда экземпляр указан в том месте, где должно быть логическое значение - скажем, в условном операторе)

`__complex__()`	Метод для приведения к комплексному типу (тип `complex`). Метод вызывается при использовании функции `complex()`
`__float__()`	Метод для приведения к числовому типу `float` (формат числа с плавающей точкой). Метод вызывается при использовании функции `float()`
`__int__()`	Метод для приведения к целочисленному типу (тип `int`). Метод вызывается при использовании функции `int()`
`__str__()`	Метод для приведения экземпляра к текстовому формату (тип данных `str`). Метод вызывается при использовании функции `str()` или в тех случаях, когда выполняется автоматическое приведение к текстовому формату

Небольшой пример с использованием этих методов приведен в листинге 7.10. В этом примере мы описываем класс `MyClass`. Экземпляр, который создается на основе этого класса, будет иметь поле-список с числовыми элементами. Далее мы устанавливаем такие "правила" приведения экземпляра к разным типам:

- Если экземпляр приводится к целочисленному типу, то в качестве значения возвращается целое число - количество элементов в поле-списке.
- При приведении экземпляра к типу `float` в качестве значения возвращается среднее арифметическое значение элементов в поле-списке.
- Если экземпляр приводится к типу `complex`, то в качестве значения возвращается комплексное число, которое создается по такому алгоритму: действительная часть комплексного числа - это элемент из списка с максимальным значением, а мнимая часть комплексного числа - это элемент из списка с минимальным значением.
- К логическому типу `bool` экземпляр будет приводиться так: если в поле-списке нечетное количество элементов, при приведении экземпляра к типу `bool` возвращается значение `True`, а если количество элементов в списке четное, возвращается значение `False`.
- Наконец, при приведении экземпляра к текстовому типу будет возвращаться текст, в котором, кроме прочего, содержатся значения элементов из поля-списка.

Теперь приступим к рассмотрению программного кода.

Листинг 7.10. Методы приведения к типу

```
# Класс
class MyClass:
    # Конструктор
    def __init__(self,nums):
        # Создаем поле - пустой список
        self.nums=list()
        # Оператор цикла для перебора
        # элементов в аргументе nums
        # (это список или множество)
        for n in nums:
            # Добавляем новый элемент в список
            self.nums.append(n)
    # Метод для приведения к типу str
    def __str__(self):
        # Начальное значение текстовой строки
        txt="Значение поля-списка:\n| "
        # Перебираем элементы в списке nums- поле
        # экземпляра класса
        for n in self.nums:
            # Дополняем строку новым текстом
            txt+=str(n)+" | "
        # Результат метода
        return txt
    # Метод для приведения к типу int
    def __int__(self):
        # Результат метода (количество элементов
        # в списке nums - поле экземпляра класса)
        return len(self.nums)
    # Метод для приведения к типу float
    def __float__(self):
        # Среднее значение элементов в
        # списке nums - поле экземпляра
        avr=sum(self.nums)/int(self)
        # Результат метода
        return avr
    # Метод для приведения к типу bool
    def __bool__(self):
        # Если нечетное количество элементов
        if int(self)%2==1:
            # Результат метода
            return True
        # Если количество элементов четное
```

```
        else:
            # Результат метода
            return False
    # Метод для приведения к типу complex
    def __complex__(self):
        # Минимальное из чисел в списке
        mn=min(self.nums)
        # Максимальное из чисел в списке
        mx=max(self.nums)
        # Комплексное число
        z=complex(mx,mn)
        # Результат метода
        return z
# Создаем экземпляр класса
obj=MyClass({2.8,4.1,7.5,2.5,3.2})
# Выводим на печать экземпляр
# (преобразование к типу str)
print(obj)
# Приведение к типу int
print("Элементов в списке:",int(obj))
#  Приведение к типу bool
if obj:
    print("Нечетное количество элементов")
#  Приведение к типу float
print("Среднее значение:",float(obj))
#  Преобразование к типу complex
print("Минимум и максимум:",complex(obj))
```

В результате выполнения этого кода получаем такое:

Результат выполнения программы (из листинга 7.10)

```
Значение поля-списка:
| 4.1 | 2.8 | 2.5 | 3.2 | 7.5 |
Элементов в списке: 5
Нечетное количество элементов
Среднее значение: 4.02
Минимум и максимум: (7.5+2.5j)
```

У класса `MyClass` есть конструктор. У конструктора есть аргумент `nums`. Этот аргумент обозначает некоторый набор элементов для формирования поля-списка. Мы будем исходить из того, что это список или множество. Но при создании экземпляра класса элементы организуются в виде списка. Для этого в теле конструктора командой `self.nums=list()` у экземпляра `self` создаем поле `nums`, которое вначале является пустым списком. Затем запускается оператор цикла, в котором переменная `n` перебирает значе-

ния из списка или множества nums, переданного аргументом конструктору. За каждый цикл командой self.nums.append(n) в поле-список nums экземпляра self добавляется с помощью метода append() новый элемент (переменная n).

Метод __str__() для преобразования к типу str также содержит оператор цикла. В теле метода сначала командой txt="Значение поля-списка:\n| " определяется начальное текстовое значение для переменной txt, а затем в операторе цикла, в котором переменная n перебирает значения из поля-списка nums экземпляра self, командой txt+=str(n)+" | " добавляется текст со значением элемента и разделителем (в виде вертикальной черты). По завершении оператора цикла переменная txt командой return txt возвращается как результат метода.

Для возможности выполнять преобразование экземпляра к типу int в классе MyClass описывается метод __int__(). В теле метода всего одна команда return len(self.nums), которой как результат метода возвращается количество элементов в списке nums из экземпляра self. Здесь для вычисления количества элементов мы использовали встроенную функцию len().

Метод __float__() предназначен для приведения к типу float. В теле метода для значений элементов поля-списка nums экземпляра self вычисляется среднее арифметическое: сумму элементов вычисляем с помощью встроенной функции sum(), и полученное значение делим на количество элементов в списке. Причем для определения количества элементов мы используем инструкцию int(self) явного приведения экземпляра self к целочисленному типу (напомним, что эта операция переопределена нами так, что ее результатом является количество элементов в поле-списке экземпляра). Результат записывается в переменную avr, которая и возвращается как результат метода.

В теле метода __bool__(), который описывается, чтобы приводить экземпляр к типу bool, запускается условный оператор, в котором проверяется условие int(self)%2==1: остаток от деления на 2 количества элементов в поле-списке экземпляра self должно равняться 1. Такое случается, если количество элементов несчетное. Как и ранее, для определения количества элементов мы используем явное приведение экземпляра self к типу int с помощью функции int(). Нелишним будет подчеркнуть, что это возможно, поскольку в теле класса описан метод __int__().

Так вот, если количество элементов нечетное, в условном операторе инструкцией return True для метода __bool__() возвращается значение True. В противном случае возвращается значение False.

Метод для преобразования к типу complex называется __complex__(). В теле метода командой mn=min(self.nums) определяется минимальное значение из набора элементов в поле-списке nums экземпляра self, а командой mx=max(self.nums) определяется максимальное значение из набора элементов в поле-списке nums экземпляра self. Затем мы создаем комплексное число командой z=complex(mx,mn). Это комплексное число возвращается как результат метода __complex__().

На заметку

Мы описываем метод __complex__(), чтобы можно было использовать функцию complex(), передавая ей аргументом ссылку на экземпляр класса. При описании метода __complex__() мы вызываем функцию complex(), но здесь проблемы нет, поскольку мы эту функцию вызываем "в стандартной", "штатной" ситуации, с двумя аргументами типа float.

Экземпляр класса создается командой obj=MyClass({2.8,4.1,7.5,2.5,3.2}). Аргументом конструктору передается множество из числовых значений, хотя это мог бы быть и список. Как бы там ни было, экземпляр obj создан, у этого экземпляра есть поле-список nums, и еще этот экземпляр можно приводить к базовым типам. Именно эти возможности проверяем. Для начала пытаемся "напечатать" экземпляр, для чего используем команду print(obj).

Хотя формально мы здесь функцию str() с аргументом-экземпляром не вызываем, по контексту команды print(obj) запускаеттся автоматическое приведение к типу str. Поэтому при выполнении команды print(obj) в окне вывода фактически отображается текстовое значение, которое возвращается в качестве результата методом __str__() с аргументом - экземпляром obj.

Аналогичная ситуация с автоматическим приведением, только теперь к типу bool, имеет место при выполнении условного оператора, в котором условием указан экземпляр obj. Во всех остальных случаях приведение выполняется явным образом с вызовом соответствующей функции и передачей ссылки на экземпляр класса аргументом функции.

В рассмотренном выше примере для определения количества элементов в списке-поле экземпляра класса мы определяли метод __int__(), в котором вызывалась функция len(), которой, в свою очередь, определялось количество элементов в поле-списке экземпляра класса. Можно было пойти иным путем: описать в теле класса метод __len__(). Этот метод вызывается, если вызывается функция len() с аргументом - экземпляром соответствующего класса.

Другими словами, если описать в классе метод `__len__()`, то можно будет вызывать функцию `len()` для экземпляра этого класса. В листинге 7.11 приведен пример описания метода `__len__()`. Кроме этого метода, также описаны методы (из тех, что пока нам незнакомы) `__index__()` (вызывается при использовании функций `bin()`, `oct()` и `hex()`) и `__round__()` (вызывается при использовании функции `round()`).

Листинг 7.11. "Округление" экземпляра и другие операции

```
# Класс
class MyClass:
    # Конструктор
    def __init__(self,txt):
        # Присваивание значения полю
        # экземпляра класса
        self.name=txt
    # Метод для приведения к текстовому типу
    def __str__(self):
        # Результат метода - значение
        # поля name экземпляра класса
        return self.name
    # Метод для вычисления "длины"
    # экземпляра класса функцией len()
    def __len__(self):
        # Результат метода - количество
        # символов в поле name экземпляра класса
        return len(self.name)
    # Метод, который вызывается при использовании
    # функций bin(), oct() и hex()
    def __index__(self):
        # Количество пробелов плюс единица
        p=self.name.count(" ")+1
        # Результат метода
        return p
    def __round__(self):
        # Присваивание значения полю name
        self.name="Сброс значения"
        # Результат - ссылка на экземпляр класса
        return self
# Начальное текстовое значение
txt="Раз, два, три, четыре, пять. "
# Уточняем текстовое значение
txt+="\nВышел Зайчик погулять."
# Создаем экземпляр класса
obj=MyClass(txt)
# Экземпляр "печатается" в окне вывода
```

```
print(obj)
# Вычисляем количество символов в поле name
print("Количество букв (символов):",len(obj))
# Количество слов (пробелов) в поле name
print("Количество слов:",obj.__index__())
# Двоичный код
print("В двоичном коде:",bin(obj))
# Восьмеричный код
print("В восьмеричном коде:",oct(obj))
# Шестнадцатеричный код
print("В шестнадцатеричном коде:",hex(obj))
# Выполняем "округление" экземпляра класса
print(round(obj))
# Выводим "на печать" экземпляр класса
print(obj)
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 7.11)

```
Раз, два, три, четыре, пять.
Вышел Зайчик погулять.
Количество букв (символов): 52
Количество слов: 8
В двоичном коде: 0b1000
В восьмеричном коде: 0o10
В шестнадцатеричном коде: 0x8
Сброс значения
Сброс значения
```

Класс называется `MyClass` и в этом классе есть конструктор. Конструктору передается текстовый аргумент, ссылка на который присваивается в качестве значения полю `name` экземпляра класса. Также ради удобства в классе описан метод `__str__()`, благодаря чему функции `print()` можно передавать аргументом экземпляр класса. При этом будет отображаться значение, на которое ссылается поле `name` экземпляра класса.

Метод `__len__()` для вычисления "длины" экземпляра класса (если экземпляр передается аргументом функции `len()`) содержит всего одну инструкцию `return len(self.name)`: результатом метода возвращается длина (количество символов) текстового поля `name` экземпляра `self`. Поэтому каждый раз, когда мы будем вызывать функцию `len()` и передавать ей аргументом ссылку на экземпляр класса, результатом будет количество символов в поле `name` этого экземпляра.

Метод `__index__()`, который вызывается при использовании функций `bin()`, `oct()` и `hex()`, содержит команду `p=self.name.count(" ")+1`, которой в переменную `p` записывается количество пробелов в поле `name` экземпляра `self`, плюс единица. Количество пробелов подсчитывается с помощью встроенного метода `count()`. Значение переменной `p` возвращается как результат метода.

На заметку

Если исходить из того, что между словами в тексте находится пробел, то количество пробелов на единицу меньше, чем количество слов в тексте. В этом смысле результат, возвращаемый методом `__init__()` можно интерпретировать (с определенной долей условности, конечно) как количество слов в текстовом поле `name` экземпляра.

В теле метода `__round__()` командой `self.name="Сброс значения"` полю `name` присваивается значение, и затем командой `return self` ссылка на экземпляр `self`, из которого вызывается метод, возвращается в качестве результата метода.

На заметку

Как отмечалось ранее, метод `__round__()` задействуется, когда функция `round()` вызывается с экземпляром класса (в данном случае `MyClass`). Например, если ссылку на экземпляр обозначить `obj`, то при выполнении команды `round(obj)` на самом деле выполняется команда `obj.__round__()`. При выполнении этой команды (в силу того, как мы описали метод `__self__()`) полю `name` присваивается значение `"Сброс значения"`, а результатом выражения `round(obj)` является ссылка на экземпляр `obj`.

Для проверки функциональности описанных методов в переменную `txt` записывается текстовое значение, а затем на основе этого текстового значения командой `obj=MyClass(txt)` создаем экземпляр `obj` класса `MyClass`. Далее следует серия команд, в которых используется ссылка на этот экземпляр. Так, сначала выполняется команда `print(obj)`. В этом случае "в игру" вступает метод `__str__()`.

В результате отображается значение поля `name` экземпляра `obj`. Количество символов в поле `name` отображается при выполнении команды `print("Количество букв (символов):",len(obj))`. В этой команде использована инструкция `len(obj)`, которая вычисляется путем вызова метода `__len__()` из экземпляра `obj`. Количество пробелов отображаем командой `print("Количество слов:",obj.__index__())`. Мы здесь явно вызываем метод `__index__()`.

При вычислении инструкций `bin(obj)`, `oct(obj)` и `hex(obj)` тоже вызывается этот метод, но числовой результат (а это, с математической точки зрения, каждый раз один и тот же результат) отображается соответственно в двоичной, восьмеричной и шестнадцатеричной системах счисления.

Наконец, при выполнении команды `print(round(obj))` полю `name` экземпляра `obj` присваивается значение `"Сброс значения"`. Поскольку результатом `round(obj)` является ссылка на экземпляр `obj`, а также благодаря описанному в классе `MyClass` методу `__str__()`, при выполнении команды `print(round(obj))` появляется сообщение `Сброс значения`. Чтобы удостовериться, что экземпляр `obj` изменился, выполняем команду `print(obj)`.

Группа методов `__setitem__()`, `__getitem__()` и `__delitem__()` автоматически вызываются соответственно при присваивании значения экземпляру через индекс, считывании значения экземпляра через индекс и удалении значения экземпляра через индекс.

На заметку

Эти же методы используются для обработки операций присваивания значения, считывания значения и удаления значения по ключу.

В листинге 7.12 приведен пример использования этих методов. Там мы описываем класс `MyClass` для создания экземпляров, у которых есть поле `nums`, представляющее собой числовой список. Количество элементов в списке определяется полем класса `Nmax`.

Описывая методы `__setitem__()`, `__getitem__()` и `__delitem__()`, мы хотим добиться эффекта, чтобы можно было обращаться к элементам поля `nums` экземпляра (для присваивания значения, считывания значения и удаления значения), указывая индекс непосредственно для экземпляра класса. Причем при указывании индекса, если этот индекс выходит за допустимые пределы, выполняется циклическая перестановка.

Программный код выглядит так:

Листинг 7.12. Операции с экземплярами через индекс

```
# Класс
class MyClass:
    # Поле класса
    Nmax=5
    # Конструктор
    def __init__(self):
        # Количество элементов
```

```
        n=MyClass.Nmax
        # Список с нулевыми элементами
        self.nums=[0 for i in range(n)]
    # Метод приведения к текстовому типу
    def __str__(self):
        # Текстовая переменная
        txt="|"
        # Формируем текст
        for s in self.nums:
            # Добавляем к тексту фрагмент
            txt+=" "+str(s)+" |"
        # Результат метода
        return txt
    # Метод для присваивания значения по индексу
    def __setitem__(self,i,v):
        # Остаток от деления
        k=i % len(self.nums)
        # Присваивание значения
        self.nums[k]=v
    # Метод для считывания значения по индексу
    def __getitem__(self,i):
        # Остаток от деления
        k=i % len(self.nums)
        # Значение элемента
        return self.nums[k]
    # Метод для удаления значения по индексу
    def __delitem__(self,i):
        # Остаток от деления
        k=i % len(self.nums)
        # Новое значение элемента
        self.nums[k]="*"
# Создается экземпляр класса
obj=MyClass()
# Содержимое списка
print(obj)
# Новые значения элементов списка
obj[0]=100
obj[2]=-3
obj[24]=123
# Содержимое списка
print(obj)
# Считывание значений элементов списка
print("Элемент с индексом 4:",obj[4])
print("Элемент с индексом 7:",obj[7])
# Удаление элементов списка
del obj[0]
```

```
del obj[9]
# Содержимое списка
print(obj)
```

При выполнении этого кода получаем такой результат:

Результат выполнения программы (из листинга 7.12)

```
| 0 | 0 | 0 | 0 | 0 |
| 100 | 0 | -3 | 0 | 123 |
Элемент с индексом 4: 123
Элемент с индексом 7: -3
| * | 0 | -3 | 0 | * |
```

В классе `MyClass` полю `Nmax` присваивается значение `5`. Это означает, что поле-список экземпляра класса будет содержать `5` элементов. При создании экземпляра все элементы этого списка получают нулевые значения. В частности, в теле конструктора создается список и заполняется нулями (все это делается с помощью генератора списков). Метод `__str__()` описан так, чтобы при передаче функции `print()` ссылки на экземпляр класса в окне вывода отображалось содержимое списка `nums` - поля экземпляра класса.

Все, что описано в теле метода `__setitem__()` - это на самом деле то, что происходит при попытке выполнить команду вида `экземпляр[индекс]=значение`. Все три параметра (ссылка на `экземпляр` класса, `индекс` и присваиваемое `значение`) описаны как аргументы метода `__setitem__()`, - соответственно `self`, `i` и `v`. В теле метода командой `k=i % len(self.nums)` вычисляется остаток от деления индекса `i` на значение `len(self.nums)` (количество элементов в списке `nums`). Затем командой `self.nums[k]=v` элементу в этом списке с индексом `k` присваивается значение `v`.

Нечто похожее происходит при выполнении программного кода метода `__getitem__()`. В данном случае речь идет о вычислении значения выражения вида `экземпляр[индекс]`. Разница в сравнении с предыдущим случаем в том, что ранее этому выражению присваивалось значение, а теперь для этого выражения значение вычисляется (считывание значения). Отсюда у метода `__getitem__()` два аргумента: ссылка на экземпляр класса `self` и значение индекса `i`. В теле метода выполняются всего две команды: командой `k=i % len(self.nums)` вычисляется индекс `k` с учетом циклической перестановки и командой `return self.nums[k]` возвращается результат метода - значение элемента с индексом `k` из списка `nums` экземпляра `self`.

Метод `__delitem__()` автоматически вызывается при попытке удаления значения командой вида `экземпляр[индекс]`. У метода (при описании)

два аргумента: ссылка на экземпляр класса и индекс (удаляемого элемента). В теле метода традиционно командой k=i % len(self.nums) "уточняется" индекс, а затем выполняется присваивание self.nums[k]="*". Таким образом, если будет предпринята попытка удалить элемент, то на самом деле этому элементу в качестве значения будет присвоен текст "*".

На заметку

Когда мы говорим об удалении элементов, то фактически речь идет о команде вида del экземпляр[индекс]. Если мы будем удалять элементы, напрямую обращаясь к полю-списку экземпляра, удаление будет выполняться так, как и должно было бы быть. Это же замечание относится к присваиванию значений через индекс (команда экземпляр[индекс]=значение) и считыванию значения через индекс (команда экземпляр[индекс]). Кроме того, в принципе совсем необязательно, чтобы у экземпляра было поле-список. В принципе, обращение к экземпляру через индекс можно реализовать совершенно иначе, не привлекая встроенные в экземпляр списки.

После создания командой obj=MyClass() экземпляра obj класса MyClass, все элементы (их всего 5) списка nums экземпляра будут нулевыми. Значения элементов списка nums отображаются при выполнении команды print(obj). Затем несколькими командами выполняется присваивание значений элементам списка nums, причем обращение к элементам выполняется через указание индекса для экземпляра класса. Так, командой obj[0]=100 элементу nums[0] (поле-список nums экземпляра obj) присваивается значение 100, командой obj[2]=-3 присваивается значение -3 элементу nums[2], и, наконец, командой obj[24]=123 элементу nums[4] присваивается значение 123.

На заметку

Что касается команды obj[24]=123: остаток от деления 24 на 5 есть 4. Поэтому речь идет о присваивании значения элементу массива nums с индексом 4.

После внесения изменений проверяем содержимое списка nums командой print(obj). Можно считывать значения и поэлементно, обращаясь "персонально" к тому или иному элементу - как это делается в инструкциях obj[4] и obj[7] (элементы nums[4] и nums[2] соответственно). При попытке удаления элементов списка командами del obj[0] и del obj[9] элементы nums[0] и nums[4] получают значение "*". Результат проверяем командой print(obj).

На заметку

Инструкции obj[7] и obj[9] означают обращение к элементам nums[2] и

nums[4] соответственно: остаток от деления 7 на 5 равен 2, а остаток от деления 9 на 5 равен 4.

Благодаря описанию специальных методов __getattr__(), __getattribute__(), __setattr__() и __delattr__() можно существенно разнообразить операции обращения к атрибутам экземпляра класса. Метод __setattr__() вызывается при присваивании значения атрибуту экземпляра класса. Метод __delattr__() вызывается при удалении атрибута экземпляра класса. При обращении (для считывания значения) к несуществующему атрибуту вызывается метод __getattr__(). Наконец, при обращении к любому атрибуту вызывается метод __getattribute__().

Метод __getattr__() вызывается при обращении к несуществующему атрибуту экземпляра класса, если в классе не описан метод __getattribute__(). Если метод __getattribute__() в классе описан, то вызывается он, причем при обращении к любому атрибуту - как существующему, так и несуществующему. Метод __getattr__(), даже если он описан в классе, при этом игнорируется.

С методом __getattribute__() связана определенная проблема: если в теле этого метода выполняется обращение к атрибуту экземпляра класса с использованием точечного синтаксиса (то есть в формате экземпляр.атрибут), то получается бесконечный циклический вызов метода __getattribute__().

Действительно, если вызывается метод __getattribute__(), а при выполнении этого метода выполняется обращение к атрибуту экземпляра класса, то метод будет вызван снова. То есть в процессе выполнения метод фактически вызывает сам себя. Но на этом процесс не заканчивается. Вызванный метод снова вызывает этот же метод, и так до бесконечности.

На первый взгляд ситуация кажется безнадежной, но это, разумеется, не так. Просто обращение к атрибутам экземпляра класса следует выполнять, явно вызывая метод __getattribute__() как функцию из класса object с явным указанием ссылки на экземпляр класса и его атрибут. Поскольку речь идет об обращении к атрибуту в теле метода __getattribute__(), то ссылка на экземпляр класса будет называться self. Если речь идет об обращении к атрибуту этого экземпляра, то обращение к этому атрибуту будет выглядеть как object.__getattribute__(self,атрибут).

Нечто похожее происходит, когда в теле метода __setattr__() (который "отвечает" за присваивание значений атрибутам экземпляров) выполняется присваивание значения атрибуту с использованием точечного синтаксиса. Проблема такая: при выполнении метода __setattr__() если присваива-

ется значение атрибуту, автоматически вызывается метод `__setattr__()`, в теле которого вызывается метод `__setattr__()`, и так далее.

Чтобы этого не происходило, при присваивании значения атрибуту экземпляра класса в теле метода `__setattr__()` следует "добираться" до нужного атрибута через специальное поле `__dict__`, значением которого является словарь из названий атрибутов экземпляра и их значений. Если речь идет о присваивании значения атрибуту, то в теле метода `__setattr__()` соответствующая команда может выглядеть как `self.__dict__[атрибут]=значение` (здесь через `self` обозначена ссылка на экземпляр класса).

Формально в данном случае мы в словаре `__dict__` изменяем значение элемента с ключом `атрибут`. Но поскольку набор атрибутов и значений атрибутов экземпляра реализуется именно через этот словарь, то нужный эффект будет достигнут - атрибут экземпляра меняет значение.

Не следует удалять поля с помощью оператора `del` в теле метода `__delattr__()`. В этом случае метод `__delattr__()` будет вызывать сам себя. Если нужно удалить поле в теле метода `__delattr__()`, лучше сделать это, удалив соответствующий элемент из словаря `__dict__`.

Небольшой пример с описанием методов `__getattr__()`, `__setattr__()` и `__delattr__()` приведен в листинге 7.13. В этом примере создается класс `MyClass`. В классе описан конструктор - такой, что при создании экземпляра класса полю `name` экземпляра присваивается значение. Также в классе описан метод `__str__()`, благодаря чему при передаче экземпляра аргументом методу `print()` отображается значение поля `name` этого экземпляра.

Мы также хотим добиться такой реакции на действия с атрибутами экземпляра класса:

- Значение поля `name` можно изменять.
- При попытке создать (путем присваивания значения) новое поле, появляется сообщение `Операция не разрешена!`, и новое поле не создается.
- Если при считывании значения поля выполняется обращение к несуществующему полю, появляется сообщение `Такого поля нет!`.
- При попытке удалить поле экземпляра появляется сообщение `Удалять поля запрещено!`.

Собственно, для решения этих задач мы и описываем в классе `MyClass` методы `__getattr__()`, `__setattr__()` и `__delattr__()`. Как именно они описаны, показано ниже:

Листинг 7.13. Обращение к полям экземпляра

```
# Класс
class MyClass:
    # Конструктор
    def __init__(self,name):
        # Полю экземпляра
        # присваивается значение
        self.name=name
    # Метод для приведения экземпляра
    # к текстовому значению
    def __str__(self):
        # Результат метода
        return self.name
    # Метод для обработки ситуации, когда
    # атрибуту присваивается значение
    def __setattr__(self,attr,val):
        # Если значение присваивается
        # полю name
        if attr=="name":
            self.__dict__[attr]=val
        # Если значение присваивается не
        # полю name
        else:
            print("Операция не разрешена!")
    # Метод для обработки ситуации, когда
    # считывается значение атрибута
    def __getattr__(self,attr):
        # Результат метода
        return "Такого поля нет!"
    # Метод для обработки ситуации, когда
    # атрибут удаляется
    def __delattr__(self,attr):
        # Отображается сообщение
        print("Удалять поля запрещено!")
# Создается экземпляр класса
obj=MyClass("Исходное значение")
# Проверяем значение поля name
print(obj)
# Новое значение поля name
obj.name="Новое значение"
# Проверяем значение поля name
```

```
print(obj)
# Присваиваем значение полю number
obj.number=100
# Проверяем значение поля number
print(obj.number)
# Удаляем поле name
del obj.name
# Проверяем значение поля name
print(obj)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 7.13)

```
Исходное значение
Новое значение
Операция не разрешена!
Такого поля нет!
Удалять поля запрещено!
Новое значение
```

Рассмотрим программные коды методов `__getattr__()`, `__setattr__()` и `__delattr__()`. Самый простой код у методов `__getattr__()` и `__delattr__()`: в теле метода `__getattr__()` результат возвращается командой `return "Такого поля нет!"`, а в теле метода `__delattr__()` выполняется единственная команда `print("Удалять поля запрещено!")`. С методом `__setattr__()` все немного сложнее: в условном операторе проверяется условие `attr=="name"`, которое истинно, если запрашиваемый атрибут (аргумент метода `attr`) - это поле `name`. В этом случае выполнятся команда `self.__dict__[attr]=val`. Здесь в словаре `__dict__` для экземпляра `self` элементу с ключом `attr` присваивается значение `val` (аргумент метода `__setattr__()`). Если условие `attr=="name"` ложно (то есть атрибут `attr` не является полем `name`), выполняется команда `print("Операция не разрешена!")`.

После описания класса `MyClass` командой `obj=MyClass("Исходное значение")` создается экземпляр `obj`, поле `name` которого получает значение `"Исходное значение"`. Проверить значение поля `name` экземпляра `obj` можно с помощью команды `print(obj)`.

Командой `obj.name="Новое значение"` полю `name` присваивается новое значение. Эта операция проходит успешно. Но при попытке выполнить команду `obj.number=100` поле `number` у экземпляра `obj` не создается, и, разумеется, значение ему не присваивается (вместо этого появляется сообщение `Операция не разрешена!`). При попытке "прочитать" значение несуществующего поля `number` в команде `print(obj.number)` появля-

ется сообщение `Такого поля нет!`. Если попытаться удалить поле `name` у экземпляра `obj` командой `del obj.name`, поле `name` не удаляется, и появляется сообщение `Удалять поля запрещено!`.

Еще один способ обработки операций присваивания значения полям, считывая значения полей и удаления полей, представлен в следующем примере. Как и в предыдущем случае, мы описываем методы `__setattr__()` и `__delattr__()`, а вместо метода `__getattr__()` описываем метод `__getattribute__()`.

На заметку

Напомним, что метод `__getattr__()` вызывается при обращении для считывания знания к несуществующему полю, в то время как метод `__getattribute__()` вызывается при считывании значения любого поля - как существующего, так и не существующего.

Итак, в классе `MyClass` описываются методы `__setattr__()`, `__getattribute__()` и `__delattr__()`.

При выполнении метода `__setattr__()`:

- появляется сообщение о начале выполнения метода;
- появляется сообщение о том, какому полю какое значение присваивается;
- выполняется присваивание полю значения;
- появляется сообщение о завершении выполнения метода.

При выполнении метода `__getattribute__()`:

- появляется сообщение о начале выполнения метода;
- появляется сообщение о том, значение какого поля считывается;
- выполняется попытка прочитать значение поля;
- если поле существует, появляется сообщение о завершении метода и возвращается значение поля;
- если поля не существует, появляется сообщение о завершении метода и возвращается текстовое значение с информацией о том, что поля не существует.

При выполнении метода `__delattr__()`:

- появляется сообщение о начале выполнения метода;
- появляется сообщение о том, какое поле удаляется;

- выполняется попытка удалить поле;
- если поле существовало и его удаление проходит успешно, появляется сообщение о завершении выполнения метода;
- если поля не существовало, появляется сообщение о невозможности удалить поле, а затем сообщение о завершении метода.

Весь программный код представлен в листинге 7.14 и выглядит так:

Листинг 7.14. Считывание значения поля

```
# Класс
class MyClass:
    # Метод вызывается, если полю
    # присваивается значение
    def __setattr__(self,attr,val):
        print("Выполняется метод __setattr__():")
        txt="*\tПолю "+str(attr)
        txt+=" присваивается значение "+str(val)
        print(txt)
        # Присваивание значения полю
        self.__dict__[attr]=val
        print("Метод __setattr__() выполнен.")
    # Метод вызывается, если
    # считывается значение поля
    def __getattribute__(self,attr):
        print("Выполняется метод __getattribute__():")
        txt="*\tСчитывается значение поля "+str(attr)
        print(txt)
        # Результат метода
        try:
            # Значение поля - если поле существует
            res=object.__getattribute__(self,attr)
        except AttributeError:
            # Если поля не существует
            res="У экземпляра поля "+str(attr)+" нет!"
        print("Метод __getattribute__() завершает работу.")
        # Результат метода
        return res
    # Метод вызывается, если
    # поле удаляется
    def __delattr__(self,attr):
        print("Выполняется метод __delattr__():")
        txt="*\tУдаляется поле "+str(attr)
        print(txt)
        try:
```

```
            # Удаление поля - если поле существует
            del self.__dict__[attr]
        except KeyError:
            # Если такого поля не существует
            print("Нельзя удалить поле "+str(attr))
        print("Метод __delattr__() выполнен.")
# Создается экземпляр класса
obj=MyClass()
# Полю name присваивается значение
obj.name="Python"
# Проверяется значение поля name
print("Значение поля name:",obj.name)
# Удаляется поле name
del obj.name
# Проверяется значение поля name
print(obj.name)
# Повторно удаляется поле name
del obj.name
```

Результат выполнения программного кода представлен ниже:

Результат выполнения программы (из листинга 7.14)

```
Выполняется метод __setattr__():
*  Полю name присваивается значение Python
Выполняется метод __getattribute__():
*  Считывается значение поля __dict__
Метод __getattribute__() завершает работу.
Метод __setattr__() выполнен.
Выполняется метод __getattribute__():
*  Считывается значение поля name
Метод __getattribute__() завершает работу.
Значение поля name: Python
Выполняется метод __delattr__():
*  Удаляется поле name
Выполняется метод __getattribute__():
*  Считывается значение поля __dict__
Метод __getattribute__() завершает работу.
Метод __delattr__() выполнен.
Выполняется метод __getattribute__():
*  Считывается значение поля name
Метод __getattribute__() завершает работу.
У экземпляра поля name нет!
Выполняется метод __delattr__():
*  Удаляется поле name
Выполняется метод __getattribute__():
*  Считывается значение поля __dict__
```

```
Метод __getattribute__() завершает работу.
Нельзя удалить поле name
Метод __delattr__() выполнен.
```

Чтобы понять, почему появляются именно такие сообщения, во внимание нужно принять следующие обстоятельства.

- После создания экземпляра класса командой `obj=MyClass()` у этого экземпляра нет полей (тех, что созданы пользователем). При выполнении команды `obj.name="Python"` у экземпляра `obj` появляется поле `name` со значением `"Python"`. При этом вызывается метод `__setattr__()`, в теле которого выполняется обращение к специальному полю `__dict__`. Обращение к полю `__dict__` в свою очередь приводит к вызову метода `__getattribute__()`.
- При проверке значение поля `name` командой `print("Значение поля name:",obj.name)` вызывается метод `__getattribute__()`.
- При удалении поля `name` командой `del obj.name` вызывается метод `__delattr__()`. Поскольку в процессе удаления поля выполняется обращение к специальному полю `__dict__`, то автоматически вызывается и метод `__getattribute__()`.
- После удаления поля `name` значение этого поля проверяется командой `print(obj.name)`. Как следствие вызывается метод `__getattribute__()`.
- Попытка повторного удаления поля `name` командой `del obj.name` приводит к вызову метода `__delattr__()`, а при выполнении этого метода - к вызову метода `__getattribute__()`.

Далее мы познакомимся с одним очень интересным механизмом, который доступен при работе с классами и экземплярами классов в Python. Речь пойдет о *перегрузке операторов*.

Перегрузка операторов

Перезагрузка завершена. Счастливого пути!
из к/ф "Гостья из будущего"

В Python существует возможность определять для экземпляров пользовательских классов такие операции, как сложение, умножение, вычитание, деление, и ряд других. Иными словами, мы имеем возможность так описать класс, что экземпляры этого класса можно будет складывать, вычитать, умножать, делить и так далее - как, например, обычные числа. Что получа-

ется в результате - это вопрос другой, и ответ на него всецело зависит от нас и нашей фантазии.

Чтобы та или иная операция (например, сложение) стала доступна для выполнения с экземплярами класса, достаточно в классе описать определенный специальный метод (каждому оператору соответствует метод, который автоматически вызывается при обработке выражения с этим оператором). Если такой метод в классе описан, то операция, соответствующая данному методу, будет доступна для экземпляров класса.

Например, оператору сложения + соответствует метод `__add__()`. Если в классе описан метод `__add__()`, то при вычислении выражения, в котором к экземпляру класса что-то прибавляется, будет вызван (автоматически) метод `__add__()`.

Небольшой пример с классом, в котором описан метод `__add__()`, представлен в листинге 7.15.

Листинг 7.15. Перегрузка оператора сложения

```
# Класс с описанием метода __add__()
class Adder:
    # Конструктор
    def __init__(self,number):
        # Полю экземпляра присваивается
        # значение
        self.number=number
    # Метод для приведения к текстовому
    # типу
    def __str__(self):
        # Формируется текст
        txt="Значение поля number = "
        txt+=str(self.number)
        # Результат метода
        return txt
    # Описание метода для операции сложения
    def __add__(self,x):
        # Вычисляется числовое значение
        number=self.number+x
        # Создается экземпляр класса
        tmp=Adder(number)
        # Результат метода - ссылка на
        # экземпляр класса
        return tmp
# Создается экземпляр класса
a=Adder(10)
# К экземпляру класса добавляется число
```

```
b=a+5
# Проверяем поле number 1-го экземпляра
print(a)
# Проверяем поле number 2-го экземпляра
print(b)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 7.15)

```
Значение поля number = 10
Значение поля number = 15
```

В классе `Adder` есть конструктор, при выполнении которого полю `number` экземпляра присваивается значение. Метод `__str__()` описан так, что при передаче экземпляра класса аргументом методу `print()` отображается значение поля `number` экземпляра.

Интерес представляет метод `__add__()`, описанный с двумя аргументами: аргумент `self` традиционно обозначает ссылку на экземпляр класса, из которого вызывается метод, а аргумент `x` обозначает числовое (как мы предполагаем) значение, которое прибавляется к экземпляру класса.

В теле метода командой `number=self.number+x` вычисляется сумма поля `number` экземпляра `self` и аргумента `x`. Результат записывается в локальную переменную `number`. Данное значение передается аргументом конструктору при создании нового экземпляра класса командой `tmp=Adder(number)`. Этот экземпляр класса возвращается как результат метода командой `return tmp`.

Таким образом, при прибавлении к экземпляру класса числового значения создается новый экземпляр класса, поле `number` которого равняется сумме значений поля `number` исходного экземпляра и прибавляемого к этому экземпляру числа.

Вне тела класса командой `a=Adder(10)` создается экземпляр `a` класса `Adder` со значением `10` для поля `number`. Благодаря тому, что в классе `Adder` описан метод `__add__()`, становится возможной команда `b=a+5`, в которой вычисляется "сумма" экземпляра `a` и числа `5`, а результат, - ссылка на новый экземпляр класса, - записывается в переменную `b`. После этого переменные `a` и `b` ссылаются на экземпляры класса `Adder`. Проверить, что речь идет о разных экземплярах, легко: достаточно выполнить команды `print(a)` и `print(b)`.

На заметку

Если мы вместо того, чтобы к экземпляру класса прибавлять число, попытаемся к числу прибавить экземпляр (например, попробуем выполнить команду `b=5+a`), появится сообщение об ошибке. Дело в том, что метод `__add__()` "обрабатывает" только прибавление к экземпляру числа, но не наоборот. Чтобы можно было к числу прибавлять экземпляр, следует в классе `Adder` описать еще и метод `__radd__()`. Программный код этого метода мог бы быть таким:

```
def __radd__(self,x):
    return self+x
```

В данном случае мы в теле метода `__radd__()` неявно вызывается метод `__add__()`, поскольку именно этот метод будет задействован при вычислении выражения `self+x`, в котором к экземпляру класса прибавляется число.

В данном случае мы определяем операции сложения экземпляра с числом и числа с экземпляром так, что результаты идентичны. Но необходимости в этом не было - мы могли бы определить эти операции по-разному.

Операторов, которые можно перегружать, достаточно много. Их обычно разбивают на три группы: математические операторы, двоичные операторы и операторы сравнения. Далее в таблице 7.3 перечислены перегружаемые математические операторы. При этом мы используем такие обозначения: через `obj` обозначен экземпляр класса, в котором перегружается метод, а другой операнд (если такой имеется) обозначен через `x`.

Таблица 7.3. Перегружаемые математические операторы

Метод	Оператор	Пример	Описание
`__add__()`	`+`	`obj+x`	Сложение экземпляра класса `obj` с операндом `x`
`__radd__()`	`+`	`x+obj`	Сложение операнда `x` с экземпляром класса `obj`
`__iadd__()`	`+=`	`obj+=x`	Сложение экземпляра класса `obj` с операндом `x` и присваивание значения переменной экземпляра `obj`
`__sub__()`	`-`	`obj-x`	Вычитание из экземпляра класса `obj` операнда `x`
`__rsub__()`	`-`	`x-obj`	Вычитание из операнда `x` экземпляра класса `obj`

Метод	Оператор	Пример	Описание
`__isub__()`	`-=`	`obj-=x`	Вычитание из экземпляра класса `obj` операнда `x` и присваивание результата переменной экземпляра класса `obj`
`__mul__()`	`*`	`obj*x`	Вычисление произведения экземпляра класса `obj` и операнда `x`
`__rmul__()`	`*`	`x*obj`	Вычисление произведения операнда `x` и экземпляра класса `obj`
`__imul__()`	`*=`	`obj*=x`	Умножение экземпляра класса `obj` на операнд `x` и присваивание результата переменной экземпляра класса `obj`
`__truediv__()`	`/`	`obj/x`	Деление экземпляра класса `obj` на операнд `x`
`__rtruediv__()`	`/`	`x/obj`	Деление операнда `x` на экземпляр класса `obj`
`__itruediv__()`	`/=`	`obj/=x`	Деление экземпляра класса `obj` на операнд `x` и присваивание результата переменной экземпляра класса `obj`
`__floordiv__()`	`//`	`obj//x`	Целочисленное деление экземпляра класса `obj` на операнд `x`
`__rfloordiv__()`	`//`	`x//obj`	Целочисленное деление операнда `x` на экземпляр класса `obj`
`__ifloordiv__()`	`//=`	`obj//=x`	Целочисленное деление экземпляра класса `obj` на операнд `x` и присваивание результата переменной экземпляра класса `obj`

Метод	Оператор	Пример	Описание
`__mod__()`	`%`	`obj%x`	Остаток от деления экземпляра класса `obj` на операнд `x`
`__rmod__()`	`%`	`x%obj`	Остаток от деления операнда `x` на экземпляр класса `obj`
`__imod__()`	`%=`	`obj%=x`	Остаток от деления экземпляра класса `obj` на операнд `x` и присваивание результата переменной экземпляра класса `obj`
`__pow__()`	`**`	`obj**x`	Возведение экземпляра класса `obj` в степень операнда `x`
`__rpow__()`	`**`	`x**obj`	Возведение операнда `x` в степень экземпляра класса `obj`
`__ipow__()`	`**=`	`obj**=x`	Возведение экземпляра класса `obj` в степень операнда `x` и присваивание результата переменной экземпляра класса `obj`
`__neg__()`	`-`	`-obj`	Применение унарного (один операнд) оператора "минус" к экземпляру класса `obj`
`__pos__()`	`+`	`+obj`	Применение унарного (один операнд) оператора "плюс" к экземпляру класса `obj`
`__abs__()`	*модуль*	`abs(obj)`	Вычисление модуля экземпляра класса `obj`

В таблице 7.4 перечислены перегружаемые побитовые операторы. Как и в предыдущем случае через `obj` обозначен экземпляр класса, в котором перегружается оператор, а через `x` - другой операнд.

Таблица 7.4. Перегружаемые побитовые операторы

Метод	Оператор	Пример	Описание
`__invert__()`	`~`	`~obj`	Побитовая инверсия экземпляра класса `obj`
`__and__()`	`&`	`obj&x`	Побитовое *и* для экземпляра класса `obj` и операнда `x`
`__rand__()`	`&`	`x&obj`	Побитовое *и* для операнда `x` и экземпляра класса `obj`
`__iand__()`	`&=`	`obj&=x`	Побитовое *и* для экземпляра класса `obj` и операнда `x` с присваиванием результата переменной экземпляра класса `obj`
`__or__()`	`|`	`obj|x`	Побитовое *или* для экземпляра класса `obj` и операнда `x`
`__ror__()`	`|`	`x|obj`	Побитовое *или* для операнда `x` и экземпляра класса `obj`
`__ior__()`	`|=`	`obj|=x`	Побитовое *или* для экземпляра класса `obj` и операнда `x` с присваиванием результата переменной экземпляра класса `obj`
`__xor__()`	`^`	`obj^x`	Побитовое *исключающее или* для экземпляра класса `obj` и операнда `x`
`__rxor__()`	`^`	`x^obj`	Побитовое *исключающее или* для операнда `x` и экземпляра класса `obj`
`__ixor__()`	`^=`	`obj^=x`	Побитовое *исключающее или* для экземпляра класса `obj` и операнда `x` с присваиванием результата переменной экземпляра класса `obj`

Метод	Оператор	Пример	Описание
`__lshift__()`	`<<`	`obj<<x`	Сдвиг влево для экземпляра класса `obj` на операнд `x`
`__rlshift__()`	`<<`	`x<<obj`	Сдвиг влево для операнда `x` на экземпляр класса `obj`
`__ilshift__()`	`<<=`	`obj<<=x`	Сдвиг влево для экземпляра класса `obj` на операнд `x` с присваиванием результата переменной экземпляра класса `obj`
`__rshift__()`	`>>`	`obj>>x`	Сдвиг вправо для экземпляра класса `obj` на операнд `x`
`__rrshift__()`	`>>`	`x>>obj`	Сдвиг вправо для операнда `x` на экземпляр класса `obj`
`__irshift__()`	`>>=`	`obj>>=x`	Сдвиг вправо для экземпляра класса `obj` на операнд `x` с присваиванием результата переменной экземпляра класса `obj`

Таблица 7.5 содержит сведения о перегружаемых операторах сравнения (`obj` - экземпляр класса, а `x` - другой операнд).

Таблица 7.5. Перегружаемые операторы сравнения

Метод	Оператор	Пример	Описание
`__eq__()`	`==`	`obj==x`	Равенство экземпляра класса `obj` и операнда `x`
`__ne__()`	`!=`	`obj!=x`	Неравенство экземпляра класса `obj` и операнда `x`
`__lt__()`	`<`	`obj<x`	Экземпляр класса `obj` меньше операнда `x`

`__gt__()`	`>`	`obj>x`	Экземпляр класса `obj` больше операнда `x`
`__le__()`	`<=`	`obj<=x`	Экземпляр класса `obj` не больше операнда `x`
`__ge__()`	`>=`	`obj>=x`	Экземпляр класса `obj` не меньше операнда `x`
`__contains__()`	`in`	`x in obj`	Операнд `x` входит в экземпляр класса `obj`

Операторов, подходящих для перегрузки, достаточно много. Здесь важно понять простую штуку: мы в принципе можем перегружать операторы как угодно (в определенных пределах, конечно). Важно только то, какой это оператор: бинарный (тот, которому нужно два операнда) или унарный (у оператора один операнд). Пример, в котором выполняется перегрузка некоторых из перечисленных выше операторов, приведен в листинге 7.16.

🕮 На заметку

Далее описывается класс для реализации такого математического объекта, как вектор. Если абстрагироваться от геометрической интерпретации вектора, то с некоторой натяжкой можно утверждать, что вектор - это набор из трех параметров, которые называются координатами вектора. С векторами могут выполняться некоторые операции. Так, если имеется два вектора $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$, то можно вычислить сумму векторов $\vec{c} = \vec{a} + \vec{b}$. По определению это вектор $\vec{c} = (c_1, c_2, c_3)$ с координатами, равными сумме координат складываемых векторов: $c_k = a_k + b_k$ (индекс $k = 1,2,3$). Аналогично вычисляется разность векторов: $\vec{c} = \vec{a} - \vec{b}$, причем координаты вектора $\vec{c} = (c_1, c_2, c_3)$ равны $c_k = a_k - b_k$ (индекс $k = 1,2,3$).

Скалярным произведением векторов $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$ называется число $\vec{a} \cdot \vec{b} = a_1b_1 + a_2b_2 + a_3b_3$, то есть сумма произведений соответствующих координат векторов. При умножении вектора на число все его координаты умножаются на это число: если $\vec{a} = (a_1, a_2, a_3)$, то $\lambda\vec{a} = (\lambda a_1, \lambda a_2, \lambda a_3)$. Аналогично вектор делится на число: нужно поделить каждую координату вектора на это число - по определению $\frac{\vec{a}}{\lambda} = \left(\frac{a_1}{\lambda}, \frac{a_2}{\lambda}, \frac{a_3}{\lambda}\right)$. Существует такая характеристика, как модуль вектора. Модуль вектора - это число, равное корню квадратному из скалярного произведения вектора на самого себя. Для вектора $\vec{a} = (a_1, a_2, a_3)$ модуль $|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}} = a_1^2 + a_2^2 + a_3^2$.

Листинг 7.16. Перегрузка операторов

```
# Функция для вычисления квадратного корня
from math import sqrt
# Класс для реализации вектора
class Vector:
    # Конструктор
    def __init__(self,x=0,y=0,z=0):
        # Полям присваиваются значения
        self.x=x
        self.y=y
        self.z=z
    # Метод для приведения к текстовому типу
    def __str__(self):
        # Текстовое значение
        txt="<"+str(self.x)+"|"
        txt+=str(self.y)+"|"
        txt+=str(self.z)+">"
        # Результат метода
        return txt
    # Сложение векторов
    def __add__(self,obj):
        # Создается экземпляр класса
        t=Vector()
        # Значения полей экземпляра
        t.x=self.x+obj.x
        t.y=self.y+obj.y
        t.z=self.z+obj.z
        # Результат метода
        return t
    # Сложение векторов с присваиванием
    def __iadd__(self,obj):
        # Изменяем экземпляр
        self=self+obj
        # Результат метода
        return self
    # Умножение вектора на вектор или
    # вектора на число
    def __mul__(self,p):
        # Проверяем тип аргумента
        if type(p)==Vector:
            # Произведение векторов
```

```
            res=self.x*p.x+self.y*p.y+self.z*p.z
            # Результат метода
            return res
        # Произведение вектора на число
        else:
            self.x*=p
            self.y*=p
            self.z*=p
            # Результат метода
            return self
    # Умножение числа на вектор
    def __rmul__(self,p):
        # Результат метода
        return self*p
    # Минус перед вектором
    def __neg__(self):
        # Результат метода
        return Vector(-self.x,-self.y,-self.z)
    # Разность векторов
    def __sub__(self,obj):
        # Результат метода
        return -obj+self
    # Разность векторов с присваиванием
    def __isub__(self,obj):
        # Изменяем объект
        self=-obj+self
        # Результат метода
        return self
    # Модуль вектора
    def __abs__(self):
        # Результат метода
        return sqrt(self*self)
    # Деление вектора на число
    def __truediv__(self,p):
        # Результат метода
        return self*(1/p)
    # Равенство векторов
    def __eq__(self,obj):
        # Если равны значения полей
        if self.x==obj.x and self.y==obj.y and self.z==obj.z:
            return True
        # Если значения полей разные
        else:
            return False
    # Неравенство векторов
    def __ne__(self,obj):
```

```
        # Результат метода
        return not self==obj
    # Один вектор "меньше" другого
    def __lt__(self,obj):
        # Если модуль первого вектора меньше
        # модуля второго вектора
        if abs(self)<abs(obj):
            return True
        # Если это не так
        else:
            return False
    #  Один вектор "больше" другого
    def __gt__(self,obj):
        # Если модуль первого вектора больше
        # модуля второго вектора
        if abs(self)>abs(obj):
            return True
        # Если это не так
        else:
            return False
    #  Один вектор "не больше" другого
    def __le__(self,obj):
        # Если модуль первого вектора не больше
        # модуля второго вектора
        if abs(self)<=abs(obj):
            return True
        # Если это не так
        else:
            return False
    #  Один вектор "не меньше" другого
    def __ge__(self,obj):
        # Если модуль первого вектора не меньше
        # модуля второго вектора
        if abs(self)>=abs(obj):
            return True
        # Если это не так
        else:
            return False
    # Побитовая инверсия для вектора
    def __invert__(self):
        # Присваиваются значения полям
        self.x=10-self.x
        self.y=10-self.y
        self.z=10-self.z
        # Результат метода
        return self
```

```
    # Сдвиг влево (экземпляр класса - первый операнд)
    def __lshift__(self,n):
        # Выполняются циклические перестановки полей
        for i in range(n):
            self.x,self.y,self.z=self.y,self.z,self.x
        # Результат метода
        return self
    #  Сдвиг влево (экземпляр класса - второй операнд)
    def __rlshift__(self,n):
        # Результат метода
        return self>>n
    #  Сдвиг вправо (экземпляр класса - первый операнд)
    def __rshift__(self,n):
        # Выполняется циклическая перестановка полей
        for i in range(n):
            self.x,self.y,self.z=self.z,self.x,self.y
        # Результат метода
        return self
    #  Сдвиг вправо (экземпляр класса - второй операнд)
    def __rrshift__(self,n):
        # Результат метода
        return self<<n
# Векторы
print("Векторы:")
a=Vector(1,2,-1)
b=Vector(1,-1,3)
c=~Vector(9,8,11)
print("a =",a)
print("b =",b)
print("c =",c)
# Вычисление модуля
print("Модуль вектора.")
print("|a| =",abs(a))
print("|b| =",abs(b))
print("|c| =",abs(c))
# Сравнение векторов
print("Сравнение векторов.")
print("a==b ->",a==b)
print("a!=b ->",a!=b)
print("a==c ->",a==c)
print("a<b ->",a<b)
print("a>b ->",a>b)
print("a<=c ->",a<=c)
print("a>=c ->",a>=c)
# Операции с векторами
print("Сумма векторов:")
```

```
print("a+b =",a+b)
c+=a
print("c+=a ->",c)
print("Разность векторов:")
print("a-b =",a-b)
c-=a
print("c-=a ->",c)
print("Умножение векторов:")
print("a*b =",a*b)
print("Умножение и деление вектора на число:")
print("a*3 =",a*3)
print("a =",a)
print("2*b =",2*b)
print("b =",b)
print("-b =",-b)
print("b =",b)
print("a/3 =",a/3)
print("a =",a)
print("Циклические перестановки:")
v=Vector(1,2,3)
print("v =",v)
print("v<<1 =",v<<1)
print("v>>1 =",v>>1)
print("2>>v =",2>>v)
print("2<<v =",2<<v)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 7.16)

```
Векторы:
a = <1|2|-1>
b = <1|-1|3>
c = <1|2|-1>
Модуль вектора.
|a| = 2.449489742783178
|b| = 3.3166247903554
|c| = 2.449489742783178
Сравнение векторов.
a==b -> False
a!=b -> True
a==c -> True
a<b -> True
a>b -> False
a<=c -> True
a>=c -> True
Сумма векторов:
```

```
a+b = <2|1|2>
c+=a -> <2|4|-2>
Разность векторов:
a-b = <0|3|-4>
c-=a -> <1|2|-1>
Умножение векторов:
a*b = -4
Умножение и деление вектора на число:
a*3 = <3|6|-3>
a = <3|6|-3>
2*b = <2|-2|6>
b = <2|-2|6>
-b = <-2|2|-6>
b = <2|-2|6>
a/3 = <1.0|2.0|-1.0>
a = <1.0|2.0|-1.0>
Циклические перестановки:
v = <1|2|3>
v<<1 = <2|3|1>
v>>1 = <1|2|3>
2>>v = <3|1|2>
2<<v = <1|2|3>
```

Поскольку мы собираемся при вычислении модуля вектора использовать функцию `sqrt()` для нахождения квадратного корня, начинаем программный код инструкцией `from math import sqrt` импорта этой функции из модуля `math`.

Основу программного кода составляет описание класса `Vector`, через который реализуются векторы. У класса есть конструктор. Если при создании экземпляра класса конструктору передаются три аргумента, то это будут координаты вектора, который реализуется в виде экземпляра класса. Если при создании экземпляра аргументы не передаются, то это соответствует нулевым координатам вектора.

В классе описан специальный метод `__str__()`, так что экземпляры класса можно передавать аргументом функции `print()`.

Для реализации операции сложения векторов (то есть суммирования экземпляров класса) описываются методы `__add__()` (сложение экземпляров) и `__iadd__()` (сложение экземпляров с присваиванием).

🕮 На заметку

Метод `__add__()` вызывается в случае, когда первым операндом является экземпляр класса. В принципе, для случая, когда экземпляр класса является вторым операндом, нужно было бы описать метод `__radd__()`. Но поскольку в дан-

ном конкретном случае оба операнда будут экземплярами класса (мы рассматриваем только такие случаи), то описывать метод `__radd__()` необходимости нет.

В теле метода `__add__()` создается экземпляр `t` класса `Vector`, а затем значения полей этого экземпляра вычисляются как сумма полей экземпляров-операндов. После этого экземпляр `t` возвращается как результат метода `__add__()`.

В теле метода `__iadd__()` выполняется команда `self=self+obj`, которой к экземпляру `self` (первый операнд) прибавляется экземпляр `obj` (второй операнд) и результат записывается в переменную `self`. Эта переменная возвращается как результат метода.

🕮 На заметку

При вычислении выражения `self=self+obj` в правой части вычисляется сумма двух экземпляров `self` и `obj`. При вычислении суммы экземпляров вызывается метод `__add__()`. Таким образом, при описании одного специального метода мы используем неявный вызов другого описанного в классе специального метода.

Метод `__mul__()` предназначен для выполнения умножения вектора на вектор и вектора на число. В первом случае речь идет об умножении двух экземпляров, а во втором случае речь идет о вычислении произведения экземпляра на число.

В зависимости от того, к какому типу относится второй операнд, результат будет разным. Второй операнд обозначен в списке аргументов метода `__mul__()` как `p`. В теле метода в условном операторе проверяется условие `type(p)==Vector`, которое истинно, если переменная `p` ссылается на экземпляр класса `Vector`. В этом случае вычисляется значение выражения `self.x*p.x+self.y*p.y+self.z*p.z` (сумма произведений соответствующих полей экземпляров `self` и `p`), вычисленное значение записывается в переменную `res`, и эта переменная возвращается как результат метода.

Если условие `type(p)==Vector` ложно, мы предполагаем, что аргумент `p` - это число. На данное число умножаются все поля экземпляра `self`, после чего ссылка на экземпляр `self` возвращается как результат метода.

🕮 На заметку

Таким образом, при умножении экземпляра на экземпляр, исходные экземпляры не меняются, а результат произведения - число. Если же экземпляр умножается на число, то изменяется исходный экземпляр.

Также в классе `Vector` описан метод `__rmul__()`. Этот метод вызывается, если второй операнд при умножении является экземпляром класса `Vector`, а первый операнд таким экземпляром не является (мы исходим из того, что это число). Таким образом, метод "выходит на сцену" при попытке умножения числа (аргумент `p` метода) на экземпляр класса `Vector` (аргумент `self` метода). В теле метода выполняется всего одна инструкция `return self*p`, которой в качестве результата возвращается произведение экземпляра `self` на числовое значение `p`. Фактически, мы операцию умножения числа на экземпляр класса определяем как умножение экземпляра на число.

Метод `__neg__()` вызывается в случае, когда перед экземпляром указывается знак *минус*. Причем в данном случае речь идет о минусе как об унарной операции. Данный метод мы описываем так, что экземпляр класса меняться не будет, а результатом возвращается вновь созданный экземпляр, поля которого получаются из полей исходного экземпляра умножением на `-1`.

Разность векторов вычисляется методом `__sub__()`. Результатом метода является значение выражения `-obj+self`, в котором унарный минус применяется к экземпляру `obj` и к полученному в результате этой операции экземпляру прибавляется экземпляр `self`. Нечто похожее происходит в теле метода `__isub__()`, но только теперь результат присваивается переменной `self`.

Модуль вектора вычисляется методом `__abs__()` как значение выражения `sqrt(self*self)` (корень квадратный из произведения экземпляра `self` на себя же).

Деление вектора на число реализуется через метод `__truediv__()`. Здесь мы исходим из того, что деление экземпляра `self` на число `p` - это то же самое, что умножение экземпляра `self` на число `1/p`. Поэтому результат метода вычисляется выражением `self*(1/p)`.

При сравнении экземпляров на предмет равенства вызывается метод `__eq__()`. Метод описан так, что значение `True` возвращается, если совпадают значения полей сравниваемых экземпляров. Сравнение экземпляров на предмет неравенства реализуется посредством метода `__ne__()`. Результат метода (выражение `not self==obj`, где `self` и `obj` - аргументы метода) получается логическим отрицанием результата сравнения экземпляров на предмет равенства.

При сравнении экземпляров с помощью операторов больше, меньше, не больше и не меньше (соответственно методы `__gt__()`, `__lt__()`, `__le__()` и `__ge__()`) на самом деле сравниваются модули экземпляров.

В классе `Vector` описываются и некоторые методы для побитовых операторов. Так, "побитовая инверсия" для экземпляра класса выполняется методом `__invert__()`: "инвертируемый" экземпляр изменяется так, что новые значения полей получаются вычитанием из числа `10` текущих значений полей экземпляра.

Кроме этого, описываются методы `__lshift__()`, `__rlshift__()`, `__rshift__()` и `__rrshift__()`. Мы предполагаем, что один из аргументов в каждом из этих методов - экземпляр класса `Vector`, а другой операнд - это целое число. При вызове методов (а это происходит при использовании операторов << и >>) выполняется циклическая перестановка значений полей экземпляра класса.

Количество последовательных перестановок определяется значением числового операнда. Направление (влево или вправо) перестановки определяется оператором и порядком аргументов. Перестановки влево происходят для команд вида `экземпляр<<число` и `число>>экземпляр`, а перестановка вправо выполняется для команд вида `экземпляр>>число` и `число>>экземпляр`. Например, в теле метода `__lshift__()` запускается оператор цикла, в котором число итераций определяется числовым аргументом, а за каждый цикл выполняется команда множественного присваивания `self.x,self.y,self.z=self.y,self.z,self.x`.

В результате поле `x` получает значение поля `y`, поле `y` получает значение поля `z`, а поле `z` получает значение поля `x`.

На заметку

Чтобы понять, как вычисляется выражение `self.x,self.y,self.z=self.y,self.z,self.x`, следует учесть, что сначала вычисляются значения в правой части, а потом каждое из них присваивается соответствующей переменной в левой части.

Вне тела класса группа команд позволяет проверить функциональность описанных в классе `Vector` специальных методов.

Резюме

Приземленная ты субстанция.
из к/ф "Гостья из будущего"

1. Наследование позволяет создавать новые классы на основе уже существующих классов. Новый создаваемый класс называется производным, а тот класс, на основе которого создается производный класс, на-

зывается базовым. Поля и методы базового класса наследуются в производном классе.

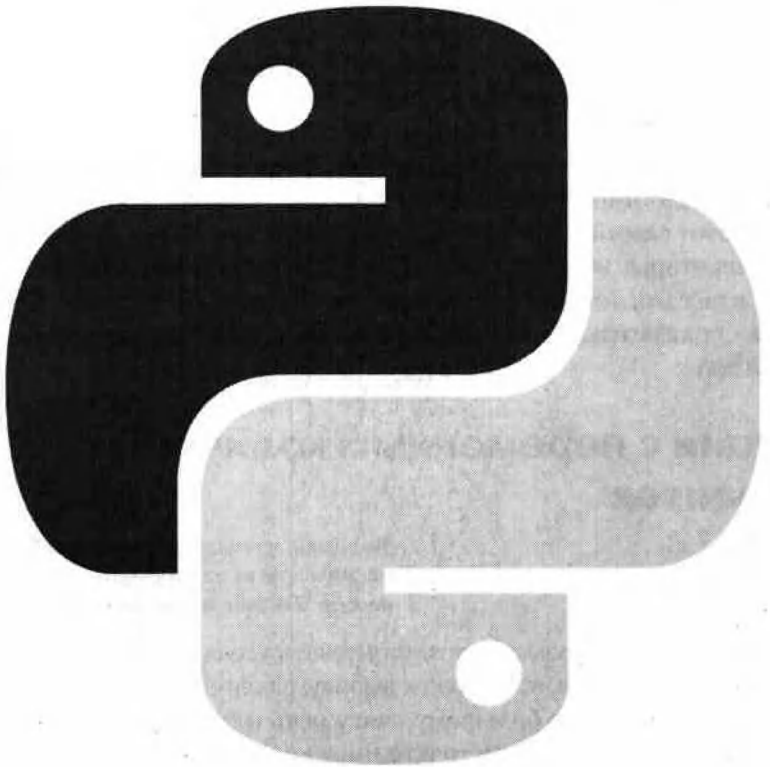
2. При создании производного класса базовый класс указывается в круглых скобках после имени производного класса. Базовых классов может быть несколько. В этом случае их имена в круглых скобках разделяются запятыми. Производный класс может в свою очередь быть базовым для другого класса.

3. Методы, унаследованные в производном классе из базового, можно переопределять (описывать заново в производном классе).

4. Существует группа специальных полей и методов, которые позволяют получать важную информацию о классе. Также с помощью специальных методов можно перегрузить операторы (такие, как оператор сложения, вычитания, умножения, побитовой инверсии или операторы сравнения) для использования их при работе с экземплярами класса.

Глава 8

Немного о разном



Всё время думать одну и ту же мысль нельзя! Это очень вредно! От этого можно соскучиться и заболеть.
из м/ф "38 попугаев"

Мы уже рассмотрели многие вопросы, касающиеся основных синтаксических конструкций языка Python, узнали разницу между списками, кортежами множествами и словарями, познакомились с управляющими инструкциями (такими, как условный оператор или оператор цикла), выяснили, как описываются функции, классы, а также как создаются экземпляры классов. Обсуждались и иные темы и вопросы. Вместе с тем, с нашей стороны было бы наивным полагать, что полученные сведения являются исчерпывающими. Понятно, что многое осталось "за бортом". Теперь наступило время заглянуть за этот самый "борт". Если более конкретно, то мы в этой главе рассмотрим некоторые вопросы, которые касаются в основном работы с функциями и классами (экземплярами класса). Также рассмотрим небольшие примеры - показательные с точки зрения методики программирования на языке Python.

Функции с переменным количеством аргументов

Ведь это же настоящая тайна! Ты потом никогда себе не простишь!
из к/ф "Гостья из будущего"

В Python существует возможность описывать (создавать) функции, которым может передаваться, от вызова к вызову, различное (произвольное) количество аргументов. По большому счету речь идет о том, что при вызове функции мы можем указывать то или иное количество аргументов, в зависимости от потребностей.

С подобной ситуацией мы уже сталкивались, когда имели дело с функциями, аргументы которых имеют значения по умолчанию. Например, если у функции описано пять аргументов, и у каждого есть значение по умолчанию, то при вызове функции ей можно не передавать аргументы вовсе или указать от одного до пяти аргументов. Но в данном подходе есть существенный недостаток: при описании все аргументы необходимо перечислить явно. Это как минимум накладывает ограничение на максимальное количество элементов, которые можно передать функции.

Кроме того, часто важное значение имеют не только сами переданные функции аргументы, но и их количество. То есть количество аргументов функции может играть роль неявного параметра или аргумента, используемого при выполнении функции или вычислении ее результата. В этом случае подход, основанный на использовании значений по умолчанию для аргументов функции, далеко не самый оптимальный.

Есть другой вариант - передавать аргументы функции в виде списка. Но это тоже не всегда удобно, поскольку в этом случае аргумент на самом деле один и обрабатывать его нужно как список. Организовать такую обработку достаточно просто, но основные трудности возникают при передаче аргументов во время вызова функции, поскольку эти аргументы предварительно придется организовать в виде списка. Хотя, конечно, и эта задача не является сложной. Пример такого подхода проиллюстрирован в листинге 8.1.

Листинг 8.1. Аргумент функции - список

```
# Функция с аргументом - списком
def show_me(args=[]):
    # Начальное значение индекса
    i=0
    print("Аргумент(ы) функции:")
    for s in args:
        # Новое значение индекса
        i+=1
        # Отображается элемент списка
        print(str(i)+"-й 'аргумент':",s)
    print("Выполнение функции завершено.")
# Примеры вызова функции
show_me()
show_me([100])
# Список - аргумент функции
nums=[10,55,2+3j,0.123]
show_me(nums)
show_me(["Python","Java"])
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 8.1)

```
Аргумент(ы) функции:
Выполнение функции завершено.
Аргумент(ы) функции:
1-й 'аргумент': 100
Выполнение функции завершено.
Аргумент(ы) функции:
1-й 'аргумент': 10
2-й 'аргумент': 55
3-й 'аргумент': (2+3j)
4-й 'аргумент': 0.123
Выполнение функции завершено.
Аргумент(ы) функции:
1-й 'аргумент': Python
2-й 'аргумент': Java
Выполнение функции завершено.
```

В данном случае мы описываем функцию `show_me()`, у которой один аргумент, причем у этого аргумента есть значение по умолчанию - пустой список. Сам аргумент, как мы предполагаем, также является списком. Во всяком случае, обрабатываем мы его именно как список. В частности, в теле функции запускается оператор цикла и в этом операторе цикла переменная `s` пробегает значения из списка `args` (аргумент функции). Индексная переменная `i` за каждый цикл увеличивает свое значение на единицу, нумеруя тем самым элементы списка `args`. Операции с этими элементами выполняются простые: они отображаются в окне вывода. При этом каждый элемент мы называем "аргументом" - как напоминание, что список `args`, указанный аргументом функции `show_me()`, "имитирует" набор аргументов функции, и количество этих аргументов не фиксировано.

Затем идут примеры вызова функции `show_me()`: без аргументов (это возможно, поскольку при описании функции для ее аргумента-списка задано значение по умолчанию), с аргументом-списком из одного элемента, из четырех элементов и из двух элементов. В каждом из этих случаем параметры, которые передаются в функцию, должны быть организованы в виде списка (список создается предварительно и ссылка на него присваивается переменной или список непосредственно передается аргументом функции).

На заметку

Формально у функции `show_me()` один аргумент (и мы исходим из того, что этот аргумент - список). Поэтому при вызове функции ей нужно передать один аргумент. Чтобы можно было вызывать функцию без передачи аргумента, мы задаем

значение аргумента по умолчанию (пустой список).

При вызове функции `show_me()`, если ей не передается аргумент, в теле функции в операторе цикла перебираются элементы пустого списка. Но поскольку в пустом списке элементов нет, то и перебирать особо нечего. Как следствие, ни одна из команд в теле оператора цикла не выполняется.

В принципе, нет ничего плохого в том, чтобы использовать означенный выше подход (с передачей функции аргументом списка), но имеются и другие возможности. Как иллюстрацию рассмотрим программный код в листинге 8.2. По сравнению с предыдущим примером здесь внесены минимальные изменения (которые, тем не менее, имеют существенные последствия).

Листинг 8.2. Функция с переменным количеством аргументов

```
# Функция с произвольным количеством аргументов.
# Единственный формальный аргумент обрабатывается
# как список
def show_me(*args):
    # Начальное значение индекса
    i=0
    print("Аргумент(ы) функции:")
    for s in args:
        # Новое значение индекса
        i+=1
        # Отображается элемент списка
        print(str(i)+"-й аргумент:",s)
    print("Выполнение функции завершено.")
# Примеры вызова функции
show_me()
show_me(100)
# Аргумент функции формируется на основе списка
nums=[10,55,2+3j,0.123]
show_me(*nums)
show_me("Python","Java")
```

Результат выполнения программного кода будет выглядеть следующим образом:

Результат выполнения программы (из листинга 8.2)

```
Аргумент(ы) функции:
Выполнение функции завершено.
Аргумент(ы) функции:
1-й аргумент: 100
Выполнение функции завершено.
```

```
Аргумент(ы) функции:
1-й аргумент: 10
2-й аргумент: 55
3-й аргумент: (2+3j)
4-й аргумент: 0.123
Выполнение функции завершено.
Аргумент(ы) функции:
1-й аргумент: Python
2-й аргумент: Java
Выполнение функции завершено.
```

Все, что мы сделали - это при описании функции `show_me()` перед аргументом `args` поставили звездочку `*`. При этом в теле функции переменная `args` обрабатывается как список (а если точнее, как *кортеж* - но в данном случае разница несущутима). А вот при вызове функции аргументы ей могут передаваться в любом количестве (включая "экзотический" случай, когда аргументов нет вовсе). В виде списка аргументы оформлять не нужно. Более того, если у нас имеется список (как, например, `nums=[10,55,2+3j,0.123]`), то его можно передать аргументом функции `show_me()`, указав перед именем списка звездочку `*` (как, например, в команде `show_me(*nums)`). Эффект будет такой, как если бы аргументами функции передавались элементы списка.

Таким образом, чтобы описать функцию с произвольным количеством аргументов, формально у функции указывается один аргумент, перед которым должна быть звездочка `*`. При этом в теле функции аргумент обрабатывается как список. При вызове функции аргументы передаются обычным образом, без формирования списка. Если при вызове функции ей передать аргументом список и перед этим списком указать звездочку `*`, то функция будет вызвана с аргументами, которые являются элементами данного списка.

На заметку

Последнее замечание относится не только к функциям с произвольным количеством аргументов, но ко всем функциям. Например, эквивалентом команды `pow(*[10,2])` является команда `pow(10,2)`. Напомним, что функцией `pow()` выполняется возведение в степень, поэтому результат обеих команд - это 10^2, то есть число 100.

Ситуация может быть не такой очевидной, как в рассмотренном примере. Вариантов достаточно много, и мы остановимся на наиболее значимых. Скажем, нередко случается, что необходимо описать функцию, в которой какое-то количество аргументов должно быть передано при вызове в обязательном порядке. Если так, то функция описывается с явным описанием необходимых аргументов (каждый из них в последовательности аргументов

обозначается отдельной переменной), а все остальные аргументы (количество которых не фиксировано) обозначаются одним аргументом со звездочкой. Пример такой ситуации приведен в листинге 8.3.

Листинг 8.3. Обязательные и необязательные аргументы

```
# Функция с двумя, как минимум, аргументами
def show_me(first,second,*other):
    # Первый аргумент
    print("Первый аргумент:",first)
    # Второй аргумент
    print("Второй аргумент:",second)
    # Количество необязательных аргументов
    n=len(other)
    print("Еще осталось",n,"аргумента:")
    # Кортеж из необязательных аргументов
    print(other)
# Вызов функции
show_me(10,20,30,40,50)
```

В результате выполнения данного программного кода в окне вывода появляются такие сообщения:

Результат выполнения программы (из листинга 8.3)

```
Первый аргумент: 10
Второй аргумент: 20
Еще осталось 3 аргумента:
(30, 40, 50)
```

В данном случае у функции `show_me()` первые два аргумента `first` и `second` описаны без звездочки, поэтому это самые обычные аргументы. Они обязательно должны быть указаны пари вызове функции. Третий аргумент функции, который называется `other`, указан со звездочкой. Поэтому при вызове функции `show_me()` первый переданный ей аргумент - это аргумент `first`, второй переданный функции аргумент - это аргумент `second`, а все прочие аргументы функции - это кортеж `other`. В теле функции аргумент `other` обрабатывается как кортеж (или как список - в данном случае это не принципиально). Так, командой `n=len(other)` вычисляется и в переменную `n` записывается количество элементов в кортеже `other`. Это, фактически, количество тех необязательных аргументов, которые переданы функции `show_me()` при вызове. Чтобы продемонстрировать, что `other` - это именно кортеж, в теле функции выполняется команда `print(other)`.

Вызывается функция `show_me()` с пятью аргументами: первые два - аргументы `first` и `second`, а три последние формируют кортеж `other`.

Но это еще не все. Читатель, наверное, помнит, что аргументы функции можно передавать не только простым перечислением через запятую (как мы обычно и делаем). Еще для аргументов можно явно указывать *ключ* или *имя* аргумента в формате `аргумент=значение`. Первый способ передачи аргументов функции называется *позиционным*, а второй - *по ключу* или *по имени*.

При вызове сначала указываются позиционные аргументы (передаются в строгом соответствии порядку, в котором аргументы описаны в функции), а уже затем те, что передаются по ключу (могут быть в произвольном порядке, поскольку название каждого аргумента указано явно). В Python существует возможность создавать функции с произвольным количеством аргументов, причем эти аргументы можно передавать не только позиционным способом, но и с явным указанием ключа. Для обозначения последних используют формальный аргумент, перед которым указывают двойную звездочку `**`.

Если более точно, то такой аргумент представляет собой словарь, ключи которого - это названия аргументов, а значения словаря - это значения аргументов функции. Например, в следующем примере описана функция `show_me()` с аргументами `first`, `second`, `*other` и `**byname`. Первые два аргумента `first` и `second` функции должны быть переданы обязательно (позиционным способом или через ключ), аргумент `other` представляет собой кортеж со списком тех аргументов (за исключением `first` и `second`), которые переданы функции позиционным способом, а аргумент `byname` является словарем с ключами и значениями аргументов (кроме `first` и `second`), которые переданы функции через ключ.

Описание функции `show_me()` и примеры ее вызова представлены в листинге 8.4.

Листинг 8.4. Передача аргументов по ключу

```
# Функция с произвольным количеством аргументов
def show_me(first,second,*other,**byname):
    # Отображаются значения аргументов
    print("first ->",first)
    print("second ->",second)
    print("other ->",other)
    print("byname ->",byname)
# Примеры вызова функции
print("1-й способ вызова функции.")
show_me(10,20,30,40,50,60,70)
print("2-й способ вызова функции.")
show_me(10,20,50,60,70,third=30,fourth=40)
```

```
print("3-й способ вызова функции.")
show_me(10,20,third=30,fourth=40)
print("4-й способ вызова функции.")
show_me(second=20,first=10)
print("5-й способ вызова функции.")
show_me(first=10,second=20,third=30,fourth=40)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 8.4)

```
1-й способ вызова функции.
first -> 10
second -> 20
other -> (30, 40, 50, 60, 70)
byname -> {}
2-й способ вызова функции.
first -> 10
second -> 20
other -> (50, 60, 70)
byname -> {'third': 30, 'fourth': 40}
3-й способ вызова функции.
first -> 10
second -> 20
other -> ()
byname -> {'third': 30, 'fourth': 40}
4-й способ вызова функции.
first -> 10
second -> 20
other -> ()
byname -> {}
5-й способ вызова функции.
first -> 10
second -> 20
other -> ()
byname -> {'third': 30, 'fourth': 40}
```

Если функция вызывается с передачей всех аргументов позиционными способом (как в команде `show_me(10,20,30,40,50,60,70)`), то первое и второе значения - это значения аргументов `first` и `second`, а все остальные аргументы формируют кортеж, являющийся значением аргумента `other`. Аргумент `byname` в этом случае является пустым словарем.

В команде `show_me(10,20,50,60,70,third=30,fourth=40)` кроме позиционных аргументов, есть аргументы, переданные по ключу (причем речь не идет об аргументах `first` и `second`). Первые два значения - ар-

гументы first и second, все остальные позиционные аргументы - элементы кортежа other, а словарь byname формируется на основе аргументов, переданных по ключу. Более конкретно, значения аргументов first и second - это 10 и 20, кортеж other состоит из трех элементов (50,60,70), а словарь byname содержит элементы со значениями 30 и 40 с ключами third и fourth соответственно. Похожая ситуация с командой show_me(10,20,third=30,fourth=40), только кортеж other в этом случае пустой.

На заметку

Обратите внимание, что аргументы third и fourth в описании функции show_me() отсутствуют. Другими словами, при вызове функции мы указываем ключи, которых как бы и нет. Тем не менее, так можно делать - язык Python предоставляет возможности по обработке таких ситуаций.

В команде show_me(second=20,first=10) первые обязательные аргументы указаны по ключу, а других аргументов нет. Поэтому и кортеж other, и словарь byname в этом случае пустые.

Наконец, в команде show_me(first=10,second=20,third=30,fourth=40) все аргументы указаны по ключу. Как следствие, аргумент other является пустым кортежем.

На заметку

Если при вызове функции аргументом ей передать словарь, перед которым указать две звездочки **, то этот словарь будет автоматически преобразован в список именованных аргументов: ключи словаря служат названиями аргументов, а значения словаря - значениями аргументов функции.

Декораторы функций и классов

Молодой человек, а вы неправильно одеты.
из к/ф "Гостья из будущего"

Чтобы понять суть и назначение *декораторов функций*, сделаем некоторые предварительные замечания. Так, результатом функции может быть функция.

На заметку

Пример функции, которая в качестве результата возвращает функцию, приведен в *главе 3* в разделе, посвященном описанию *лямбда-функций*, а также в разделе, специально посвященном этой теме.

Аргументом функции также может быть функция. Поэтому никто и ничто не запрещает описать нам функцию, которая аргументом принимает функцию и результатом возвращает функцию. Для большей конкретики обозначим через `f()` некоторую функцию, а `F()` - это будет функция, которая аргументом принимает функцию и результатом возвращает функцию. Функции `F()` в качестве аргумента может быть передана функция `f()` (аргументом указывается имя функции). Таким образом, выражение `F(f)` представляет собой функцию.

Результат этого выражения можно присвоить в качестве значения переменной, и эта переменная будет ссылаться на объект функции. Другими словами, такую переменную можно будет рассматривать как функцию. Мало того, этой переменной может быть `f` - мы можем воспользоваться командой `f=F(f)`. В этой команде нет ошибки. Ее выполнение приведет к переопределению функции `f()`.

На заметку

Команда `f=F(f)` выполняется так. Вначале переменная `f` ссылается на некоторый объект, который определяет функцию `f()`. При выполнении команды `f=F(f)` сначала рассчитывается результат выражения `F(f)`. При этом используется текущая ссылка на объект функции `f()`. Результатом выражения `F(f)` является новый объект, определяющий некоторую функцию. Ссылка на эту функцию записывается в переменную `f`. Внешний эффект от этих действий сводится к тому, что меняется функция `f()`.

Таким образом, с помощью функции `F()` мы можем выполнить преобразование функции `f()`. Для этого достаточно воспользоваться командой `f=F(f)`. Такого же эффекта можно добиться, если перед описанием функции `f()` поставить инструкцию `@F` (знак `@` и имя функции `F()`, на основе которой выполняется преобразование). То есть речь идет об инструкции вида

```
@F
def f(аргументы):
    # тело функции
```

Об инструкции `@F` говорят, что это *декоратор функции*. Пример использования декоратора функций приведен в листинге 8.5.

На заметку

Далее мы рассматриваем "математический" пример. В частности, мы рассматриваем выражение вида $\exp(-f(x)^2)$, где $f(x)$ есть некоторая функция. Если бы нам предстояло часто использовать такого рода выражения (для разных функций $f(x)$), то разумно было бы использовать декоратор функций. А именно, мы описываем

функцию $F(f)$ такую, что $F(f)(x) = \exp(-f(x)^2)$. Функцию $F(f)$ достаточно описать один раз. Затем используя ее как декоратор, можем определять функции для вычисления выражений вида $\exp(-f(x)^2)$, при этом определяя фактически только код для вычисления выражения $f(x)$.

Листинг 8.5. Декоратор функций

```
# Импорт математических функций
from math import exp,sin,cos,tan
# Функция для использования в декораторе
def F(f):
    # Лямбда-функция (анонимная функция)
    res=lambda x: exp(-f(x)**2)
    return res
# Функция для использования в декораторе
def Q(f):
    # Внутренняя функция
    def q(x):
        return tan(f(x))
    return q
# Функция с декоратором
@F # Декоратор
def f(x):
    return sin(x)
# Функция с декоратором
@F # Декоратор
def g(x):
    return cos(x)
# Функция с двумя декораторами
@Q # Второй декоратор
@F  # Первый декоратор
def h(x):
    return x
# Переменная
n=5
# Значения функций в разных точках
print("Функция f():")
for i in range(n+1):
    z=i/n
    print(f(z),"->",exp(-sin(z)**2))
print("Функция g():")
for i in range(n+1):
    z=i/n
    print(g(z),"->",exp(-cos(z)**2))
print("Функция h():")
```

```
for i in range(n+1):
    z=i/n
    print(h(z),"->",tan(exp(-z**2)))
```

При выполнении программного кода получаем такой результат:

Результат выполнения программы (из листинга 8.5)

```
Функция f():
1.0 -> 1.0
0.961299270288799 -> 0.961299270288799
0.8592918618974276 -> 0.8592918618974276
0.7270055824268487 -> 0.7270055824268487
0.5977397854322518 -> 0.5977397854322518
0.49259230319603176 -> 0.49259230319603176
Функция g():
0.36787944117144233 -> 0.36787944117144233
0.38268981631591364 -> 0.38268981631591364
0.4281193125221935 -> 0.4281193125221935
0.5060201050223136 -> 0.5060201050223136
0.6154508201347391 -> 0.6154508201347391
0.7468233644427067 -> 0.7468233644427067
Функция h():
1.5574077246549023 -> 1.5574077246549023
1.4307602577401106 -> 1.4307602577401106
1.143266474503948 -> 1.143266474503948
0.8383239288289558 -> 0.8383239288289558
0.582285681136045 -> 0.582285681136045
0.38542559176909813 -> 0.38542559176909813
```

В самом начале программного кода мы импортируем из модуля `math` функции `exp()` для вычисления экспоненты, `sin()` для вычисления синуса, `cos()` для вычисления косинуса и `tan()` для вычисления тангенса. Далее описывается функция `F()`, у которой, как предполагается, аргумент `f` - имя другой функции. В теле функции переменной `res` присваивается в качестве значения ссылка на анонимную функцию (лямбда-функцию), объект которой создается инструкцией `lambda x: exp(-f(x)**2)`. Переменная `res` (ссылка на объект функции) возвращается как результат функции `F()`.

Функция `Q()` также возвращает в качестве результат функцию (поэтому может и будет использована в декораторе). В теле функции описывается внутренняя функция `q()`, которая в качестве результата возвращает выражение `tan(f(x))`, где `x` - аргумент функции `q()`, а `f` - аргумент функции `Q()`. Имя функции `q` (ссылка на объект функции `q()`) возвращается как результат функции `Q()`.

На заметку

Таким образом, результатом выражения $Q(f)$ является функция, которая для аргумента x возвращает значение $Q(f)(x) = tg(f(x))$ (тангенс от значения $f(x)$).

Формально функция `f()` описана так, что для аргумента `x` возвращается значение `sin(x)`. Но поскольку перед описанием функции есть декоратор `@F`, то это все равно, как если бы после определения функции `f()` выполнялась команда `f=F(f)`. В результате функция `f()` определяется как такая, что значение выражения `f(x)` есть `exp(-sin(x)**2)`.

Похожая ситуация с функцией `g()`. Поскольку перед функцией есть декоратор `@F`, а в теле функции результат возвращается командой `return cos(x)` (через `x` обозначен аргумент функции `g()`), то на самом деле результатом вычисления инструкции `g(x)` является значение `exp(-cos(x)**2)`.

Что касается функции `h()`, то перед ее описанием есть два дескриптора: `@Q` и `@F`. Это эквивалентно выполнению команды `h=Q(F(h))` после определения функции `h()`. В итоге значение выражения `h(x)` эквивалентно значению выражения `tan(exp(-x**2))`.

Далее в программном коде для нескольких значений аргумента `x` (в диапазоне от `0` до `1`) вычисляются значения `f(x)`, `g(x)` и `h(x)`. Для сравнения в явном виде вычисляются также и соответствующие математические выражения.

Аналогично тому, как декораторы используются для функций, могут использоваться и декораторы для классов.

На заметку

Правда, на практике к этому прибегают не часто, но такая возможность имеется и о ней лучше знать.

Для создания декоратора класса необходимо иметь функцию, аргументом которой передается класс и результатом возвращается класс. На первый взгляд все это может показаться странным. Но тут нужно вспомнить, что в Python класс сам является объектом, а имя класса является ссылкой на этот объект. Таким образом, упомянутой функции в качестве имени должно передаваться имя класса, а в теле функции имя другого класса должно возвращаться как результат. Вариантов здесь может быть много, мы рассмотрим достаточно простой случай.

Итак, предположим, что функция F() для аргумента - объекта класса A, возвращает значение - объект класса F(A). Если класс A описан с декоратором @F, то это все равно, как если бы после описания класса A выполнялась команда A=F(A). Пример приведен в листинге 8.6.

Листинг 8.6. Декоратор классов

```
# Функция с аргументом - классом и
# результатом - объектом класса
def F(A):
    # Внутренний класс
    class Alpha(A):
        # Метод экземпляра внутреннего класса
        def hi(self):
            print("Класс Alpha!")
    # Результат функции - объект класса
    return Alpha
# Функция с аргументом - классом и
# результатом - объектом класса
def Q(A):
    # Внутренний класс
    class Bravo(A):
        # Метод экземпляра внутреннего класса
        def hi(self):
            print("Класс Bravo!")
    # Результат функции - объект класса
    return Bravo
# Класс с декоратором
@F # Декоратор класса
class First:
    # Метод экземпляра класса
    def hello(self):
        print("Класс First!")
# Класс с декоратором
@F # Декоратор класса
class Second:
    # Метод экземпляра класса
    def hello(self):
        print("Класс Second!")
# Класс с двумя декораторами
@Q # Второй декоратор класса
@F # Первый декоратор класса
class Third:
    # Метод экземпляра класса
    def hello(self):
        print("Класс Third!")
```

```
# Функция для вызова методов экземпляра
def show_obj(obj):
    # Класс экземпляра
    print("Класс экземпляра:",obj.__class__)
    # Вызов методов экземпляра
    obj.hi()
    obj.hello()
# Функция для отображения характеристик класса
def show_class(A):
    # Имя класса
    print("Имя класса:",A.__name__)
    # Базовый класс
    print("Баэовый класс:",A.__bases__)
    # Цепочка наследования
    print("Цепочка наследования:",A.__mro__)
# Создание экземпляров классов
one=First()
two=Second()
three=Third()
# Методы экземпляров
print("Экземпляры классов.")
for obj in [one,two,three]:
    show_obj(obj)
# Характеристики классов
print("Классы.")
for A in [First,Second,Third]:
    show_class(A)
```

Выполнение программного кода приводит к таким результатам:

Результат выполнения программы (из листинга 8.6)

```
Экземпляры классов.
Класс экземпляра: <class '__main__.F.<locals>.Alpha'>
Класс Alpha!
Класс First!
Класс экземпляра: <class '__main__.F.<locals>.Alpha'>
Класс Alpha!
Класс Second!
Класс экземпляра: <class '__main__.Q.<locals>.Bravo'>
Класс Bravo!
Класс Third!
Классы.
Имя класса: Alpha
Базовый класс: (<class '__main__.First'>,)
Цепочка наследования: (<class '__main__.F.<locals>.Alpha'>,
<class '__main__.First'>, <class 'object'>)
```

```
Имя класса: Alpha
Базовый класс: (<class '__main__.Second'>,)
Цепочка наследования: (<class '__main__.F.<locals>.Alpha'>,
<class '__main__.Second'>, <class 'object'>)
Имя класса: Bravo
Базовый класс: (<class '__main__.F.<locals>.Alpha'>,)
Цепочка наследования: (<class '__main__.Q.<locals>.Bravo'>,
<class '__main__.F.<locals>.Alpha'>, <class '__main__.Third'>,
<class 'object'>)
```

В этом примере мы описываем две функции, которые используются в декораторах класса. Также с использованием декораторов создаются три класса. Есть и другой вспомогательный код. Рассмотрим все это поэтапно.

Функции `F()` аргументом, как мы предполагаем, передается класс (обозначен как `A`). В теле функции описан внутренний класс, который называется `Alpha`. Класс `Alpha` создается на основе класса `A` путем наследования. Класс `Alpha`, таким образом, наследует атрибуты класса `A`, и при этом в теле класса `Alpha` описан метод `hi()`. Действие метода сводится к отображению в окне вывода текстового значения. Ссылка на внутренний класс возвращается в качестве результата функции `F()`.

Аналогичным образом описана функция `Q()`, только ее внутренний класс называется `Bravo`. Он тоже создается наследованием того класса, что передан аргументом функции `Q()`. У внутреннего класса `Bravo` также, как и у класса `Alpha` из функции `F()`, имеется метод `hi()`. Этот метод также отображает сообщение - только другое. Результатом функция `Q()` возвращает ссылку на класс `Bravo`. Поэтому функция `Q()`, как и функция `F()`, может использоваться в декораторе класса.

Класс `First` описан с декоратором `@F`. В теле класса описан метод `hello()`. При создании класса `First` происходит следующее:

- Сначала создается объект класса `First` в соответствии с тем, как непосредственно описан код класса, а ссылка на объект класса записывается в переменную `First`. На этом этапе класс содержит лишь метод экземпляра `hello()` (плюс специальные поля и методы).
- Далее вызывается функция `F()`, аргументом которой передается ссылка `First` на объект соответствующего класса. При выполнении кода функции на основе класса `First` создается внутренний (локальный) класс `Alpha`. У этого класса, помимо метода экземпляра `hi()`, появляется еще и метод экземпляра `hello()`.
- Ссылка на объект внутреннего класса, созданного при вызове функции `F()`, присваивается в качестве значения переменной `First`. В результате переменная ссылается на объект класса, у которого есть

методы экземпляра `hi()` (метод из класса `Alpha` функции `F()`) и `hello()` (описан непосредственно в классе `First`).

В этой схеме есть несколько важных для понимания моментов. Во-первых, хотя класс мы называем `First`, на самом деле переменная с таким названием будет ссылаться на класс, который создавался при вызове функции `F()`. А это локальный класс `Alpha`. Поэтому если воспользоваться инструкцией `First.__name__`, то в качестве значения получим имя класса `Alpha`. А вот базовым для этого класса является класс с именем `First`. И это именно тот класс, который был создан в самом начале, и ссылка на который первоначально записывалась в переменную `First`. Во-вторых, если создать экземпляр для класса, на который ссылается переменная `First` (например, командой `one=First()`), то этот экземпляр, на самом деле, будет экземпляром упомянутого выше локального класса `Alpha` (класса, созданного при вызове функции `F()`). В этом легко убедиться, обратившись к полю `__class__` экземпляра.

Класс `Second` также описан с декоратором `@F`. Ситуация сходна с предыдущей:

- Создается объект класса `Second` так, как описан код этого класса. Класс содержит метод экземпляра `hello()`.
- Вызывается функция `F()`. Аргументом функции передается ссылка на объект класса `Second`. При этом наследованием класса `Second` создается внутренний класс `Alpha` (но это не тот класс, который создавался при создании класса `First`).
- Ссылка на объект созданного внутреннего класса присваивается переменной `Second`.

Таким образом, переменная `Second` ссылается на объект класса, который был создан наследованием "исходного" класса `Second`. Если запросить имя класса `Second` через специальное поле `__name__`, то оно будет `Alpha`.

🕮 На заметку

Хотя значения поля `__name__` для классов `First` и `Second` совпадают (речь об имени `Alpha`), на самом деле `First` и `Second` - это разные классы. Просто значением поля `__name__` является "локальное" имя класса. В данном случае классы создаются при вызове функции `F()`. Каждый раз создается новый класс, с "локальным" именем `Alpha`. Но поскольку речь идет о локальных пространствах имен при вызове функции, конфликта по причине совпадения названий не возникает (то есть классы с одинаковыми локальными названиями попадают в разные пространства имен).

Объект класса, на который ссылается переменная `Second`, имеет методы экземпляра `hi()` из локального класса `Alpha` и `hello()`, описанный непосредственно в теле класса `Second`.

Более сложная ситуация с классом `Third`. Этот класс описан с двумя декораторами: `@F` и `@Q`. Происходит следующее:

- Создается объект класса `Third`.
- Ссылка на объект класса `Third` передается в функцию `F()`. В результате создается новый объект класса: класс `Third` наследуется в локальном классе `Alpha`.
- Ссылка на указанный объект передается в функцию `Q()`. В теле этой функции наследованием переданного аргументом класса создается новый класс с локальным названием `Bravo`.
- Ссылка на новый объект класса записывается в переменную `Third`.

Таким образом, локальное имя класса, на который ссылается переменная `Third`, есть `Bravo`. У экземпляра этого класса есть два метода: метод `hi()` из класса `Bravo`, созданного при вызове функции `Q()`, и метод `hello()`, описанный в самом классе `Third`.

На заметку

В описанной выше схеме сначала создается объект класса с методом `hi()` из локального класса `Alpha`, а затем этот метод фактически переопределяется или перекрывается методом `hi()` из локального класса `Bravo`. Тем не менее, через экземпляр `three` и переменную `Third` можно "добраться" и до версии метода `hi()` из класса `Alpha`. Для этого достаточно воспользоваться командой `super(Third,three).hi()`, которой с помощью функции `super()` из экземпляра `three` вызывается метод `hi()`, описанный в классе, являющемся базовым для класса, на который ссылается переменная `Third`.

Функция `show_obj()` предназначена для вызова методов экземпляров классов (тех, на которые ссылаются переменные `First`, `Second` и `Third`). Для отображения характеристик самих классов (название класса, базовый класс и цепочка наследования) предназначена функция `show_class()`.

Документирование и аннотации в функциях

В вашем возрасте уже пора научиться врать как следует.
из к/ф "Гостья из будущего"

Есть некоторые полезные свойства функций, которое можно использовать на практике для повышения читабельности программного кода. В первую очередь следует сказать об особой роли первой (после названия функции) текстовой строки в описании функции. Эта строка служит для документирования - создания особого комментария, доступ к которому можно получить программными методами. Более конкретно, у каждой функции есть специальное поле `__doc__` (два подчеркивания в начале, два подчеркивания в конце). Если после имени функции через точку указать поле `__doc__`, то значением будет текстовая строка, указанная в самом начале в теле функции. Небольшой пример приведен в листинге 8.7.

Листинг 8.7. Документирование функции

```
# Функция
def hi():
    "Эта функция просто выводит сообщение."
    print("Документирование функции.")
# Вызываем функцию
hi()
# Описание функции
print(hi.__doc__)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 8.7)

```
Документирование функции.
Эта функция просто выводит сообщение.
```

Помимо этого, при описании функции можно создавать "аннотации" для аргументов функции. Речь идет о специальных комментариях, которые добавляются в описании аргументов функции. Также можно обозначить результат (тип результата), который ожидается получить в результате выполнения функции.

Аннотации для аргументов помещается после этих самых аргументов, в качестве разделителя используется двоеточие. Аннотацией может быть любое корректное с точки з рения синтаксиса Python выражение. Если у аргумен-

тов есть значения по умолчанию, то они указываются через знак равенства после аннотации к аргументу.

Аннотация для результата функции указывается через стрелку -> после круглых скобок с описанием аргументов функции, но перед двоеточием, которым завершается строка описания заголовка функции. Общий шаблон функции с аннотациями для аргументов и результата такой (жирным шрифтом выделены ключевые элементы):

```
def имя_функции(аргумент: аннотация=значение,
аргумент: аннотация=значение,...)->аннотация:
    # код функции
```

Для получения доступа к аннотациям, использованным в функции, из объекта функции вызывается специальное поле `__annotations__`. Значение этого свойства - словарь, ключами которого являются названия аргументов, а значения - аннотации к аргументам. Аннотация к результату функции соответствует ключу `return`. Пример создания и использования функции с аннотациями приведен в листинге 8.8.

Листинг 8.8. Функция с аннотациями

```
# Функция с аннотациями
# для аргументов и результата
def show(a:"первый аргумент",b:int=0)->None:
    print("a =",a)
    print("b =",b)
# Вызываем функцию
show(10)
show(10,20)
# Использованные аннотации:
print(show.__annotations__)
# Аннотацияч для аргумента a
annt=show.__annotations__["a"]
print("Аргумент a:",annt)
# Аннотация для результата
res=show.__annotations__["return"]
print("Возвращаемый результат:",res)
```

При выполнении программного кода получаем следующее:

Результат выполнения программы (из листинга 8.8)

```
a = 10
b = 0
a = 10
b = 20
```

```
{'b': <class 'int'>, 'a': 'первый аргумент', 'return': None}
Аргумент a: первый аргумент
Возвращаемый результат: None
```

Важно понимать, что наличие аннотаций не влияет алгоритм выполнения кода функции, равно как наличие аннотации для результата функции реально не ограничивает свободу программиста в плане типа возвращаемого функцией значения. Другими словами, аннотации - это такая "декоративная" составляющая в описании функции. Тем не менее, некоторую пикантность в определение функции они могут привнести. В листинге 8.9 приведен еще один небольшой пример функции, в которой в качестве аннотации для аргумента и результата функции указаны команды отображения текста в окне вывода.

Листинг 8.9. Аннотации как команды

```
def demo(arg:print("Здесь есть аргумент."))->print("Результата
не будет."):
    print("Вас приветствует demo!")
    print(arg)
print("Вызываем функцию demo():")
demo("Код функции выполнен.")
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 8.9)

```
Здесь есть аргумент.
Результата не будет.
Вызываем функцию demo():
Вас приветствует demo!
Код функции выполнен.
```

Чтобы понять смысл происходящего, нужно учесть, что аннотации не просто "принимаются к сведению" - они выполняются. Выполняются при создании объекта функции. А объект функции создается при выполнении кода с описанием функции (не путать с выполнением кода при вызове функции). Другими словами, аннотации будут выполнены в том месте и в то время, когда интерпретатор "доберется" до программного кода с описанием функции. В рассматриваемом примере команды `print("Здесь есть аргумент.")` и `print("Результата не будет.")`, которые мы использовали как аннотации для аргумента и результата функции, будут выполнены при "просмотре" интерпретатором кода с описанием функции. Это происходит только раз.

При вызове функции эти команды уже выполняться не будут. Поэтому текст `"Здесь есть аргумент."` и `"Результата не будет."` ото-

бражается еще до того, как была вызвана функция `demo()`. А после вызова функции `demo()` эти сообщения уже не появляются, зато отображается текст `"Вас приветствует demo!"` и `"Код функции выполнен."` как следствие выполнения команд в теле функции `demo()`.

На заметку

Порядок выполнения аннотаций может отличаться от того порядка, в котором они указаны в описании функции.

Исключения как экземпляры классов

Отрицательный результат - это тоже результат.
из к/ф "Гостья из будущего"

С обработкой исключительных ситуаций мы немного знакомы. Пришел черед расширить наши познания в этой области. А именно, более пристальное внимание мы уделим тому обстоятельству, что типы исключений описываются классами, а сами исключения являются экземплярами классов. Кроме того, нам предстоит познакомиться с некоторыми полезными механизмами - такими, например, как генерирование исключительных ситуаций и создание пользовательских классов исключений.

Хотя кое-что об обработке ошибок (исключительных ситуаций) мы уже знаем, здесь нелишним будет напомнить основные моменты. Итак, контролируемый код помещается в блок `try`. Код, который предназначен для обработки ошибки, размещается в блоке `except`. Блоков, помеченных ключевым словом `except`, может быть несколько. Каждый блок соответствует ошибке определенного типа. Тип (или, точнее, класс) ошибки, которая обрабатывается в соответствующем `except`-блоке, указывается после ключевого слова `except`.

Для описания различных ошибок предусмотрены специальные классы, которые называют классами встроенных исключений. Эти классы образуют иерархию, основанную на наследовании. Общая иерархическая структура классов встроенных ислючений представлена на рис. 8.1.

На заметку

Как видим, классов достаточно много. В вершине иерархии находится класс `BaseException`. У этого класса всего четыре производных класса (`GeneratorExit`, `KeyboardInterrupt`, `SystemExit` и `Exception`), причем только у класса `Exception` есть производные классы. Поэтому не будет преувеличением сказать, что подавляющее большинство классов встроенных ошибок - потомки (прямые или опосредованные) класса `Exception`.

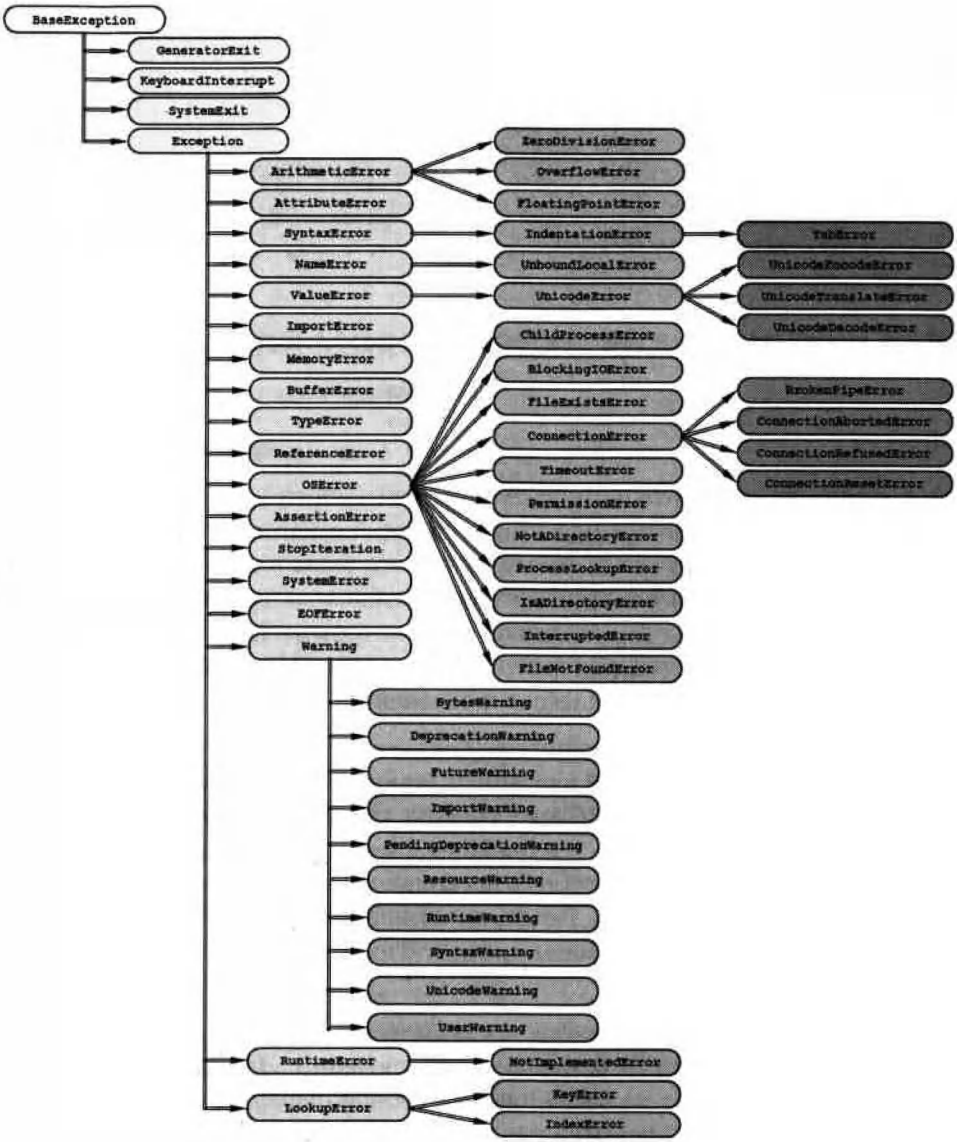


Рис. 8.1. Иерархия классов встроенных исключений

Когда при выполнении программного кода возникает ошибка, автоматически определяется класс, которому эта ошибка соответствует, и для этого класса создается экземпляр. Образно выражаясь, созданный таким образом экземпляр содержит основную информацию о возникшей ошибке. К этому экземпляру можно получить доступ и использовать его в явном виде в процессе обработки исключения.

Мы уже знаем из предыдущих глав, как перехватывать исключения разных типов. Здесь мы рассмотрим еще несколько вопросов, связанных с обработкой исключительных ситуаций. Практически все они, так или иначе, подразумевают использование парадигмы ООП и обращение к классам встроенных ошибок. В частности, нас будет интересовать:

- как выполнять в одном `except`-блоке обработку для исключений нескольких типов;
- как получить доступ к экземпляру класса, описывающему исключение (доступ к экземпляру исключения);
- как генерировать исключения;
- как создавать пользовательские классы исключений.

Допустим, необходимо реализовать обработку исключительных ситуаций так, чтобы нескольких разных типов ошибок обрабатывались одинаково. Что нужно учесть в этом случае? Есть два момента. Во-первых, если в `except`-блоке указан определенный класс исключения, то перехватываться будут исключения не только данного класса, но и исключения, относящиеся к производным классам.

Например, в `except`-блоке для исключений класса `Exception` будут перехватываться практически все исключения, поскольку класс `Exception` является базовым (по цепочке наследования) для всех классов, кроме `GeneratorExit`, `KeyboardInterrupt`, `SystemExit` и `BaseException` (см. рис. 8.1). Во-вторых, в `except`-блоке можно указать не только один класс исключения, а целый кортеж, элементами которого являются названия классов исключений. В таком `except`-блоке будут перехватываться все исключения из кортежа. Например, предположим, что для перехвата и обработки исключения используется следующий шаблон:

```
try:
    # контролируемый код
except (ArithmeticError,TypeError,ValueError):
    # код для обработки
except Warning:
    # код для обработки
```

Что это означает? Это означает, что при возникновении ошибки в try-блоке она будет перехвачена и обработана в первом except-блоке, если тип ошибки относится к классу ArithmeticError (включая производные классы FloatingPointError, OverflowError и ZeroDivisionError), TypeError или ValueError. Если тип ошибки иной, "в игру" вступает второй except-блок. В этом блоке обрабатываются ошибки, которые относятся к классу Warning и всем его производным классам.

Мы уже знаем, что если есть класс, то на его основе, скорее всего, можно создать экземпляр класса. По отношению к ошибкам мы говорим о классе, который соответствует той или иной исключительной ситуации. Но где же экземпляры таких классов?

В принципе, никто не запрещает нам создать экземпляр класса обычным способом, как мы до этого создавали экземпляры прочих классов - через имя класса, указав в круглых скобках (если нужно) аргументы для конструктора и присвоив результат такой инструкции некоторой переменной (ссылка на экземпляр класса). Хотя обычно (но не всегда) используют те экземпляры, что автоматически создаются при возникновении исключительной ситуации. То есть такой экземпляр всегда существует, если уж возникла ошибка. Другое дело, как мы этот экземпляр использует (или не используем).

В случае если в except-блоке мы все же решили явно использовать экземпляр исключения, шаблон командного кода будет таким (жирным шрифтом выделены ключевые элементы кода):

```
try:
    # контролируемый код
except класс_исключения as экземпляр:
    # код для обработки
```

Если коротко, то после названия класса исключения через ключевое слово as указывается формальное название для экземпляра класса исключения, который будет автоматически создан при возникновении ошибки. Именно черед это имя в блоке обработки исключения следует обращаться к экземпляру класса исключения. В качестве иллюстрации к использованию экземпляра класса исключения рассмотрим небольшой пример, представленный в листинге 8.10.

Листинг 8.10. Экземпляр класса исключения

```
# Импорт функций из модуля random
from random import seed, randint
print("Перехват исключений.")
# Список из двух элементов
```

```
nums=[1,2]
# Слово из шести букв
txt="Python"
# Целочисленная переменная
a=10
# Список из трех элементов
names=[nums,a,txt]
# Инициализация генератора случайных чисел
seed(123)
# Оператор цикла
for i in range(10):
    # Контролируемый код
    try:
        # Случайное целое число в диапазоне от 0 до 2
        n=randint(0,2)
        print("Сгенерировано число:",n)
        # Количество элементов в списке или тексте.
        # При попытке вызвать функцию len() для целого
        # числа возникает ошибка (некорректный тип данных)
        print("Количество элементов:",len(names[n]))
        # Неверный индекс или деление на ноль
        names[n+1]//=n
    # Обработка ошибки, связанной с
    # некорректным типом данных
    except TypeError as err:
        # Экземпляр исключения передан
        # аргументом в функцию print()
        print("Ошибка:",err)
    #  Обработка ошибок, связанных с некорректным индексом
    # или попыткой деления на ноль
    except (LookupError,ArithmeticError) as err:
        print("Проблема с вычислениями:")
        # Определяем класс ошибки по экземпляру класса
        print("Класс ошибки -",err.__class__)
    # Строка из 45 символов формируется с помощью
    # оператора повторения
    print("-"*45)
print("Работа завершена.")
```

Результат выполнения программного кода может быть таким (а может быть и несколько иным, поскольку многое зависит от использованного генератора случайных чисел):

Результат выполнения программы (из листинга 8.10)

```
Перехват исключений.
Сгенерировано число: 0
```

```
Количество элементов: 2
Проблема с вычислениями:
Класс ошибки - <class 'ZeroDivisionError'>
-----------------------------------------
Сгенерировано число: 1
Ошибка: object of type 'int' has no len()
-----------------------------------------
Сгенерировано число: 0
Количество элементов: 2
Проблема с вычислениями:
Класс ошибки - <class 'ZeroDivisionError'>
-----------------------------------------
Сгенерировано число: 1
Ошибка: object of type 'int' has no len()
-----------------------------------------
Сгенерировано число: 1
Ошибка: object of type 'int' has no len()
-----------------------------------------
Сгенерировано число: 0
Количество элементов: 2
Проблема с вычислениями:
Класс ошибки - <class 'ZeroDivisionError'>
-----------------------------------------
Сгенерировано число: 0
Количество элементов: 2
Проблема с вычислениями:
Класс ошибки - <class 'ZeroDivisionError'>
-----------------------------------------
Сгенерировано число: 1
Ошибка: object of type 'int' has no len()
-----------------------------------------
Сгенерировано число: 2
Количество элементов: 6
Проблема с вычислениями:
Класс ошибки - <class 'IndexError'>
-----------------------------------------
Сгенерировано число: 2
Количество элементов: 6
Проблема с вычислениями:
Класс ошибки - <class 'IndexError'>
-----------------------------------------
Работа завершена.
```

В этом примере мы используем генератор случайных чисел, поэтому прежде всего командой `from random import seed, randint` импортируем из модуля `random` функции `seed()` и `randint()`. Функция `randint()` предназначена для генерирования целых случайных (псевдос-

лучайных) чисел. Функция `seed()` используется для инициализации генератора случайных чисел.

На заметку

Хотя с генератором случайных чисел мы уже имели дело, нелишним будет кое-что напомнить.

Как бы мы ни пытались сгенерировать случайное число, на самом деле это неподъемная задача. Максимум, чего мы можем добиться - создать иллюзию того, что число случайное. Поэтому генерируемые "случайные" числа правильнее было бы называть "псевдослучайными". Так, собственно, их и называют. Технически процесс генерирования псевдослучайных чисел состоит в том, что по некоторой формуле или алгоритму вычисляются числа. Для начала вычислений необходимо, по крайней мере, одно начальное, "затравочное" число. Когда это число задано, запускается процесс вычислений. За все это отвечает группа утилит, которую мы называем генератором случайных чисел. Действия, связанные с определением "начального" состояния генератора случайных чисел, называются инициализацией генератора случайных чисел. Для выполнения инициализации следует указать число и передать его аргументом функции `seed()`. Поскольку, как мы уже знаем, вычисление случайных (псевдослучайных) чисел - процесс строго детерминированный, а "начальная точка" в этом процессе определяется используемым для инициализации числом, то используя в качестве "инициализатора" одно и то же число, будем получать одну и ту же последовательность чисел. Если инициализация генератора случайных чисел в явном виде не выполнена, обычно выполняется неявная инициализация с использованием системного времени.

Командой `nums=[1,2]` создается список из двух элементов, командой `txt="Python"` создаем текст, а числовая переменная определяется командой `a=10`. Наконец, командой `names=[nums,a,txt]` создается список из трех элементов, один из которых - список, другой - целое число, а третий - текст. Первый и третий элементы могут быть указаны аргументом функции `len()`, которая, как известно, в качестве результата возвращает количество элементов для списка и количество букв для текста.

Инициализация генератора случайных чисел выполняется командой `seed(123)`. В данном случае аргумент функции `seed()` играет формальную роль - можно указать и другое число.

Затем запускается оператор цикла (10 итераций). За каждый цикл выполняется группа команд, которые размещены в блоке `try`. А именно, выполняются такие команды: командой `n=randint(0,2)` генерируется случайное целое число в диапазоне от `0` до `2` включительно, и это число записывается в переменную `n`. Далее в инструкции `len(names[n])` выполняется попытка вычислить количество элементов в элементе списка `names` с индексом `n`.

Если значение переменной `n` равно `0` или `2`, то проблем с выполнением инструкции не возникает, поскольку элемент с индексом `0` - это список, а эле-

мент с индексом 2 - текст. Но если значение переменной n равняется 1, то возникает ошибка класса TypeError, так как элемент с индексом 1 - целое число. Для перехвата данной ошибки есть специальный except-блок, в котором экземпляр ошибки класса TypeError обозначен как err (инструкция TypeError as err в except-блоке).

В этом except-блоке выполняется команда print("Ошибка:",err). Здесь экземпляр ошибки err передан аргументом функции print(), что приводит к автоматическому приведению экземпляра err к текстовому формату (экземпляры классов исключений поддерживают такую операцию). Экземпляр err будет замещен текстом "object of type 'int' has no len()" (что означает *объект типа 'int' не имеет (функции) len()*).

Все это происходит, напомним, если выполнение инструкции len(names[n]) приводит к ошибке. Но если выполнение инструкции к ошибке не приводит, то ошибка точно возникнет при выполнении команды names[n+1]//=n. Здесь сделана попытка изменить значение элемента names[n+1], вычислив результат целочисленного деления текущего значения этого элемента на значение переменной n. Необходимо учесть, что уж если дело дошло до выполнения команды names[n+1], то значение n равно 0 или 2 (при значении 1 ошибка возникает на предыдущем шаге). При значении 0 индекс n+1 дает значение 1. Элемент с таким индексом в списке names имеется. Но беда в том, что значение этого элемента мы пытаемся поделить на ноль. Такой самонадеянный проступок приводит к ошибке класса ZeroDivisionError.

Если значение переменной n равно 2, то деления на ноль нет, но индекс n+1 равен 3, и элемента с таким индексом в списке names нет. В результате возникает ошибка класса IndexError. Для перехвата и той, и другой ошибки предназначен except-блок с кортежем из классов исключений LookupError и ArithmeticError. Экземпляр класса ошибки обозначен как err.

На заметку

Класс исключения ZeroDivisionError является производным классом от класса ArithmeticError. Класс исключения IndexError является подклассом класса LookupError. Поэтому в except-блоке с кортежем из классов исключений LookupError и ArithmeticError перехватываются исключения классов ZeroDivisionError и IndexError.

В блоке при обработке ошибки выполняется обращение к полю __class__ экземпляра ошибки err. Значением этого поля является название класса, к которому относится экземпляр err.

На заметку

В программном коде есть команда `print("-"*45)`. В данном случае отображается строка, которая получается повторением символа "-" 45 раз.

Как отмечалось ранее, исключения можно не только перехватывать и обрабатывать, но и генерировать. Зачем генерировать исключения - вопрос отдельный. Например, метод генерирования исключений можно рассматривать как некий способ имитации условного оператора или выполнение аналога безусловного перехода в программном коде. Но как бы там ни было, генерирование исключений как механизм программирования существует, и мы с ним кратко познакомимся.

С технической точки зрения для генерирования исключения (ошибки) достаточно после инструкции `raise` указать экземпляр класса исключения. Экземпляр класса исключения можно или создать, или воспользоваться "готовым" - например тем, что создастся автоматически, когда ошибка реально возникает, и затем передается в блок обработки исключения. Пример в листинге 8.11 иллюстрирует обе эти ситуации.

Листинг 8.11. Генерирование исключения

```
# Создаем экземпляр исключения
err_one=ZeroDivisionError("Ошибка деления на ноль!")
print("Сейчас \"возникнет\" ошибка.")
print("Первая:")
# Описание экземпляра исключения
print(err_one)
# Контролируемый код
try:
    # Генерирование ошибки
    raise err_one
# Обработка ошибки
except ZeroDivisionError as err_two:
    print("Вторая:")
    # Описание ошибки
    print(err_two)
    # Контролируемый код
    try:
        # Повторное генерирование ошибки
        raise err_two
    # Обработка ошибки
    except ZeroDivisionError as err_three:
        print("Третья:")
        # Описание ошибки
        print(err_three)
```

```
        # Контролируемый код
        try:
            # Попытка деления на ноль
            a=1/0
        # Обработка ошибки
        except ZeroDivisionError as err_four:
            print("Четвертая:")
            # Описание ошибки
            print(err_four)
print("Больше никаких ошибок.")
```

При выполнении программного кода получаем такой результат:

Результат выполнения программы (из листинга 8.11)

```
Сейчас "возникнет" ошибка.
Первая:
Ошибка деления на ноль!
Вторая:
Ошибка деления на ноль!
Третья:
Ошибка деления на ноль!
Четвертая:
division by zero
Больше никаких ошибок.
```

В этом программном коде командой `err_one=ZeroDivisionError ("Ошибка деления на ноль!")` мы создаем экземпляр `err_one` исключения класса `ZeroDivisionError`. Текст, который передается аргументом конструктору при создании экземпляра, служит описанием ошибки. Именно этот текст будет результатом приведения экземпляра исключения к текстовому формату: например, если передать экземпляр `err_one` в качестве аргумента методу `print()`, в результате в окне вывода появится текст `"Ошибка деления на ноль!"`.

Затем в `try`-блоке выполняется команда `raise err_one`, в результате чего генерируется ошибка, реализованная через экземпляр `err_one` класса `ZeroDivisionError`.

На заметку

Обратите внимание, что создание экземпляра исключения не означает возникновения ошибки. Для генерирования ошибки используем инструкцию `raise`.

Для обработки сгенерированной ошибки предназначен `except`-блок, и экземпляр обрабатываемой ошибки обозначен в нем как `err_two`. В самом бло-

ке выполняется команда print(err_two), которой в окне вывода отображается описание ошибки. В данном случае следствие выполнения этой команды - появление в окне вывода сообщения Ошибка деления на ноль!. Это то же самое сообщение, которое появлялось при выполнении команды print(err_one). Ничего удивительного здесь нет, поскольку переменная err_two ссылается фактически на тот же экземпляр, на который ссылалась переменная err_one. При перехвате сгенерированного командой raise err_one исключения экземпляр исключения передается в except-блок, и в этом блоке ссылка на экземпляр ошибки выполняется через переменную err_two.

В except-блоке обработки ошибки имеется свой try-блок, в котором командой raise err_two выполняется повторное генерирование ошибки. Новая обработка выполняется в except-блоке, и в этом блоке ссылка на экземпляр ошибки выполняется через переменную err_three. И снова речь идет о том же самом экземпляре, что и в предыдущих случаях. Свидетельством тому - результат выполнения команды print(err_three) (результат такой же, как при выполнении команд print(err_one) и print(err_two)).

Затем при попытке выполнить команду a=1/0 снова генерируется ошибка, связанная с деление на ноль (класс ZeroDivisionError). Но на этот раз экземпляр исключения создается автоматически и это уже совсем иной экземпляр, который к переменным err_one, err_two и err_three не имеет никакого отношения. Поэтому при выполнении команды print(err_four) в except-блоке (переменная err_four - ссылка на экземпляр исключения) получаем, по сравнению с предыдущими случаями, совсем иной результат (текст, описывающий исключение).

Для генерирования исключений не обязательно прибегать к помощи инструкции raise. Можно воспользоваться несколько иным подходом, в основе которого использование инструкции assert. Если после инструкции assert указать выражение с логическим значением False, будет сгенерировано исключение класса AssertionError. Небольшой пример, в котором исключение генерируется с помощью инструкции assert, приведен в листинге 8.12.

Листинг 8.12. Использование инструкции assert

```
# Функция с логическим аргументом
def show(arg):
    # Контролируемый код
    try:
        # Если аргумент arg равен False -
        # генерируется исключение
```

```
        assert arg
        # Если ошибка не сгенерирована
        print("Штатный режим.")
    # Обработка исключения
    except AssertionError as err:
        print("Исключение:",err.__class__)
# Вызываем функцию
show(True)
show(False)
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 8.12)

```
Штатный режим.
Исключение: <class 'AssertionError'>
```

В данном случае мы описываем функцию `show()`, у которой, как предполагается, один логический аргумент. Этот аргумент в теле функции указывается после инструкции `assert`. Поэтому при значении аргумента `False` генерируется исключение класса `AssertionError`. При обработке исключения в `except`-блоке отображается значение поля `__class__` экземпляра исключения (значение поля - это класс исключения). Если же аргумент функции равен `True`, исключение не генерируется, а выполняется команда `print("Штатный режим.")` после `assert`-инструкции. Вызывая функцию `show()` с разными аргументами, получаем разные результаты (точнее, в окне вывода отображается разный текст).

На заметку

При генерировании исключения с помощью инструкции `assert` можно добавить описание (текст) для экземпляра исключения. Для этого после логического значения в `assert`-инструкции через запятую указывается текст - тот текст, который будет отображаться при попытке "напечатать" экземпляр исключения. Например, в результате выполнения инструкции `assert False "Неожиданная ошибка"` будет создан экземпляр класса `AssertionError`. При приведении экземпляра к текстовому формату будет возвращаться текстовая строка `"Неожиданная ошибка"`.

Еще один вопрос, который мы здесь рассмотрим, связан с созданием классов пользовательских исключений. Общий рецепт в данном случае прост: необходимо создать новый класс путем наследования класса из иерархии классов встроенных исключений. Обычно в качестве базового класса рекомендуют использовать класс `Exception`.

Процесс создания классов пользовательских исключений рассмотрим на примере, представленном в листинге 8.13. В этом примере мы решаем ква-

дратные уравнения. Поиск решения в числовом виде подразумевает генерирование исключений пользовательских типов.

На заметку

Напомним, что квадратное уравнение имеет вид $ax^2 + bx + c = 0$. В зависимости от значений параметров a, b и c возможны следующие варианты. Формально у квадратного уравнения два решения: $x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ и $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$, но это в том случае, если параметр $a \neq 0$ и выражение $D = b^2 - 4ac$ (называется дискриминантом квадратного уравнения) неотрицательно, то есть если $D \geq 0$. Причем если дискриминант нулевой (то есть $D = 0$), то оба корня уравнения совпадают: $x_1 = x_2 = -\frac{b}{2a}$. Если дискриминант отрицательный (имеет место $D < 0$), то на множестве действительных чисел квадратное уравнение решений не имеет. Зато имеет два решения на множестве комплексных чисел: $x_1 = \frac{-b + i\sqrt{-D}}{2a}$ и $x_1 = \frac{-b - i\sqrt{-D}}{2a}$, где через i обозначена мнимая единица (такая, что по определению $i^2 = -1$). Все это имеет место, если параметр a не равен нулю. Если же $a = 0$, то фактически речь идет не о квадратном, а о линейном уравнении вида $bx + c = 0$. У этого уравнения одно решение $x = -\frac{c}{b}$, но это при условии, что $b \neq 0$. Если $b = 0$, то все зависит от того, равен ли нулю параметр c. При $c = 0$ решением уравнения $bx + c = 0$ будет любое число. При $c \neq 0$ у уравнения $bx + c = 0$ решений нет.

Понятно, что при написании программного кода для решения квадратных уравнений с учетом всех перечисленных выше факторов можно использовать структуру из вложенных условных операторов. Но мы пойдем другим путем. В приведенном программном коде описывается два класса пользовательских исключений. Одно исключение генерируется в случае, если уравнение не является квадратным, а другое - если у уравнения комплексные решения (корни).

Листинг 8.13. Пользовательские классы исключений

```
# Импорт математической функции
from math import sqrt
# Класс пользовательской ошибки
class LinearEquationWarning(Warning):
    # Конструктор
    def __init__(self,b,c):
        # Контролируемый код
        try:
            # Присваивание значения полю экземпляра.
            # Возможна ошибка деления на ноль
            self.x=-c/b
        # Если возникла ошибка деления на ноль
        except ZeroDivisionError:
            # Если параметр нулевой
```

```
            if c==0:
                # Решение - любое число
                self.x="любое число"
            # Если параметр не равен нулю
            else:
                # Решений нет
                self.x="решений нет"
    # Метод для приведения экземпляра исключения
    # к текстовому формату
    def __str__(self):
        # Формируется текстовое значение
        # для результата метода
        txt="Решение уравнения: "
        txt+=str(self.x)
        # Результат метода
        return txt
# Класс пользовательского исключения
class ComplexRootsError(Exception):
    # Конструктор
    def __init__(self,a,b,D):
        # Значение поля экземпляра
        self.x1=complex(-b/2/a,sqrt(-D)/2/a)
        # Значение поля экземпляра
        self.x2=complex(-b/2/a,-sqrt(-D)/2/a)
    # Метод для приведения экземпляра исключения
    # к текстовому формату
    def __str__(self):
        # Формирование текстовой строки для
        # результата метода
        txt="x1 ="+str(self.x1)+"\n"
        txt+="x2 ="+str(self.x2)
        # Результат метода
        return txt
# Функция для отображения параметров уравнения
def show_params(a,b,c):
    # Текст с форматированием
    print("Параметры: a={0}, b={1}, c={2}:".format(a,b,c))
# Функция для решения квадратного уравнения
def find_roots(a,b,c):
    # Отображение параметров решаемого уравнения
    show_params(a,b,c)
    # Контролируемый код
    try:
        # Если параметр нулевой
        if a==0:
            # Генерируется исключение
```

```
            # пользовательского типа
            raise LinearEquationWarning(b,c)
        # Дискриминант уравнения
        D=b*b-4*a*c
        # Если дискриминант меньше нуля
        if D<0:
            # Генерируется исключение
            # пользовательского типа
            raise ComplexRootsError(a,b,D)
        # Первый корень (решение) уравнения
        x1=(-b+sqrt(D))/2/a
        # Второй корень (решение) уравнения
        x2=(-b-sqrt(D))/2/a
        # Отображается корень уравнения
        print("x1 =",x1)
        # Отображается корень уравнения
        print("x2 =",x2)
    # Перехватывается исключение пользовательского типа
    except LinearEquationWarning as err:
        print("Это линейное уравнение!")
        # Отображается информация об экземпляре исключения
        print(err)
    # Перехватывается исключение пользовательского типа
    except ComplexRootsError as err:
        print("Внимание! Решения комплексные!")
        # Отображается информация об экземпляре исключения
        print(err)
# Решаем квадратные уравнения
print("Решение уравнения a*x**2+b*x+c=0.")
# Два действительных корня
find_roots(2,-3,1)
# Совпадающие корни
find_roots(1,2,1)
# Два комплексных корня
find_roots(2,1,3)
# Линейное уравнение с одни решением
find_roots(0,-6,5)
# Решений нет
find_roots(0,0,1)
# Решение - любое число
find_roots(0,0,0)
```

Результат выполнения данного программного кода представлен ниже:

Результат выполнения программы (из листинга 8.13)

```
Решение уравнения a*x**2+b*x+c=0.
Параметры: a=2, b=-3, c=1:
x1 = 1.0
x2 = 0.5
Параметры: a=1, b=2, c=1:
x1 = -1.0
x2 = -1.0
Параметры: a=2, b=1, c=3:
Внимание! Решения комплексные!
x1 =(-0.25+1.1989578808281798j)
x2 =(-0.25-1.1989578808281798j)
Параметры: a=0, b=-6, c=5:
Это линейное уравнение!
Решение уравнения: 0.8333333333333334
Параметры: a=0, b=0, c=1:
Это линейное уравнение!
Решение уравнения: решений нет
Параметры: a=0, b=0, c=0:
Это линейное уравнение!
Решение уравнения: любое число
```

Поскольку мы собираемся в процессе вычислений извлекать квадратный корень, командой `from math import sqrt` из модуля `math` импортируем функцию извлечения квадратного корня `sqrt()`.

Класс с названием `LinearEquationWarning` создается путем наследования класса исключения `Warning`. В теле класса описан конструктор, и среди аргументов этого конструктора, кроме ссылки `self` на экземпляр класса, есть еще параметры `b` и `c`. Названия этих аргументов совпадают с названиями параметров решаемого уравнения.

На заметку

Параметр `a` в данном случае не нужен, поскольку ошибка класса `LinearEquationWarning` будет генерироваться в случае, если параметр `a` равен нулю. То есть если уж дело дошло до создания экземпляра исключения `LinearEquationWarning`, то это автоматически означает, что параметр `a` в уравнении нулевой.

В теле конструктора командой `self.x=-c/b` полю `x` экземпляра класса присваивается значение, равное, очевидно, корню уравнения (линейного в данном случае). Но при этом возможна ошибка деления на ноль - если параметр `b` нулевой. В этом случае на сцену выходит обработчик исключения класса `ZeroDivisionError` и с помощью условного оператора, в зависи-

мости от значения параметра c, полю x экземпляра класса присваивается значение "любое число" или "решений нет".

Также в классе LinearEquationWarning описан метод __str__(), что позволяет выполнять автоматическое приведение экземпляров класса к текстовому формату. В теле метода формируется текстовая строка, содержащая, кроме прочего, значение поля x экземпляра класса. Этот текст возвращается как результат метода.

Еще один класс пользовательского исключения называется ComplexRootsError и создается он наследованием класса Exception. Конструктору, описанному в теле класса, передаются параметры уравнения a и b, а также дискриминант уравнения D. Неявно предполагается, что дискриминант отрицательный (потому что только в этом случае генерируется ошибка класса ComplexRootsError). В теле конструктора полям x1 и x2 экземпляра класса присваиваются значения. Это комплексные числа. Комплексные числа создаются с помощью встроенной функции complex() (напомним, что ее аргументы - это действительная и мнимая части комплексного числа). При вычислениях использованы формулы для комплексных корней квадратного уравнения (см. вставку выше).

При приведении экземпляра класса к текстовому формату (программный код метода __str__()) в текстовую строку, возвращаемую как результат, включаются значения полей x1 и x2 экземпляра класса.

Также в программном коде, кроме описания классов пользовательских исключений, имеется функция show_params(), предназначенная для отображения значений параметров для решаемого квадратного уравнения.

На заметку

В теле функции show_params() выполняется команда print("Параметры: a={0}, b={1}, c={2}:".format(a,b,c)). Здесь мы прибегли к формированию текстовой строки для отображения в окне вывода посредством вызова функции format() из строки формата. В тексте "Параметры: a={0}, b={1}, c={2}:", из которого вызывается функция format(), инструкции {0}, {1} и {2} определяют место вставки значений соответственно первого, второго и третьего аргументов функции format().

Функция show_params() вызывается в теле другой функции, которая называется find_roots() и предназначена для решения уравнения. Кроме этого, в теле функции find_roots() в условном операторе проверяется условие a==0, и если оно истинно, командой raise LinearEquationWarning(b,c) генерируется исключение пользовательского класса LinearEquationWarning.

На заметку

Командой `LinearEquationWarning(b,c)` создается экземпляр класса `LinearEquationWarning`, но ссылка на этот экземпляр явно ни в какую переменную не записывается. Это так называемый анонимный экземпляр класса.

Если ошибка не генерировалась (условие `a==0` ложно), то командой `D=b*b-4*a*c` вычисляется дискриминант уравнения. В случае если он отрицательный (условие `D<0` в еще одном условном операторе), командой `raise ComplexRootsError(a,b,D)` генерируется ошибка пользовательского класса `ComplexRootsError`.

На заметку

Командой `ComplexRootsError(a,b,D)` создается анонимный экземпляр класса `ComplexRootsError`.

Если условие `D<0` не выполняется, командами `x1=(-b+sqrt(D))/2/a` и `x2=(-b-sqrt(D))/2/a` вычисляются корни уравнения (при этом, если дискриминант нулевой, то значения корней одинаковы), а затем командами `print("x1 =",x1)` и `print("x2 =",x2)` вычисленные значения отображаются в окне вывода.

При перехвате и обработке исключения пользовательского типа `LinearEquationWarning` командой `print("Это линейное уравнение!")` выводится текстовое предупреждение, а затем командой `print(err)` отображается и описание экземпляра исключения `err`. Аналогично происходит обработка исключения пользовательского класса `ComplexRootsError`.

На заметку

В принципе, можно было бы перехватывать и обрабатывать исключения классов `LinearEquationWarning` и `ComplexRootsError` в одном `except`-блоке. При этом "специфические" для каждого из использованных нами `except`-блоков текстовые сообщения можно было бы "спрятать" в конструкторы экземпляров соответствующих классов.

То, что классы `LinearEquationWarning` и `ComplexRootsError` создавались наследованием различных базовых классов (`Warning` и `Exception` соответственно) в контексте их использования не является принципиальным. Но различие все же есть. Например, в `except`-блоке для перехвата исключений класса `Exception` будут перехватываться и ошибки классов `LinearEquationWarning` и `ComplexRootsError`, а в `except`-блоке для перехвата исключений класса `Warning` - ошибки только класса `LinearEquationWarning`.

Для иллюстрации того, как работает вся эта схема, в программном коде приведено несколько примеров вызова функции `find_roots()` с разными наборами параметров для решаемого уравнения.

Итераторы и функции-генераторы

Клевать будет так, что клиент позабудет обо всем на свете.
из к/ф "Бриллиантовая рука"

В Python есть особая категория объектов, которые называют *итераторами*. Итераторы создаются на основе *итерационных объектов*. К итерационным объектам относятся списки, кортежи и текст. Но нас интересует создание пользовательских итерационных объектов. Такие объекты создаются на основе классов. 1

Классы, на основе которых создаются итерационные объекты, будем называть *итерационными классами* или *классами-итераторами* (хотя последнее название, возможно, и не очень корректное). Таким образом, все начинается с класса-итератора. Если подойти к вопросу формально, то класс-итератор - это класс, в котором поддерживаются (то есть, определены) специальные методы `__iter__()` и `__next__()`.

Но чтобы понять, зачем все это нужно и как используется, лучше "начать издалека". И в первую очередь рассмотрим программный код, представленный в листинге 8.14.

Листинг 8.14. Функции next() и iter()

```
s=iter([1,2,3])
print(next(s))
print(next(s))
print(next(s))
try:
    print(next(s))
except Exception as err:
    print(err.__class__)
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 8.14)

```
1
2
3
<class 'StopIteration'>
```

Мы выполняем простые операции, в которых используем функции `iter()` и `next()`. А именно, переменной `s` присваивается результат выражения `iter([1,2,3])`, в котором аргументом функции `iter()` передается список `[1,2,3]`. Сразу заметим, что результатом выражения `iter([1,2,3])` является *итератор* - этот объект можно передавать функции `next()`. Причем каждый раз при вызове функции `next()` (с аргументом-итератором) получаем новое значение - правда, до определенного момента.

Свидетельством тому служит результат последовательного выполнения команды `print(next(s))`: при первых трех вызовах возвращаются соответственно числовые значения 1, 2 и 3. Несложно догадаться, что это элементы списка `[1,2,3]`, который передавался аргументом функции `iter()`. А вот при выполнении команды `print(next(s))` в четвертый раз, генерируется ошибка класса `StopIteration`. Это в принципе тоже понятно - в списке `[1,2,3]` закончились элементы. Что из этого следует? А следует буквально вот что: итератор - это такой объект, который можно передавать аргументом функции `next()` и получать в результате некоторое значение (на самом деле не обязательно числовое). Продолжается это не до бесконечности. В какой-то момент при вызове функции `next()` с итератором в качестве аргумента генерируется исключение класса `StopIteration`. Возникает вопрос: а какова во всем этом роль функции `iter()`? Ответ простой: с помощью функции `iter()` на основе итерационного объекта создается итератор. В рассмотренном примере у нас был список `[1,2,3]`. Список - это итерационный объект, но еще не итератор. Итератор на основе списка нужно создать. Чтобы создать итератор на основе списка, список передается аргументом функции `iter()`.

На заметку

Мы достаточно часто использовали и используем инструкции с ключевым словом `for` вида `for элемент in что-то`. Обрабатываются они на самом деле так. Сначала создается итератор `iter(что-то)`. Затем этот итератор последовательно при каждой итерации передается функции `next()`. Итерации продолжаются до тех пор, пока при вызове функции `next()` не будет сгенерировано исключение `StopIteration`.

Но функции `next()` и `iter()` - это, так сказать, внешняя часть айсберга. Рассмотрим программный код, представленный в листинге 8.15.

Листинг 8.15. Методы `__next__()` и `__iter__()`

```
s=[1,2,3].__iter__()
print(s.__next__())
print(s.__next__())
print(s.__next__())
```

```
try:
    print(s.__next__())
except Exception as err:
    print(err.__class__)
```

Результат выполнения программного кода будет таким же, как в предыдущем случае:

Результат выполнения программы (из листинга 8.15)

```
1
2
3
<class 'StopIteration'>
```

Ситуация очень напоминает рассмотренный ранее пример. Только теперь вместо функции `iter()` с аргументом-списком из списка вызывается метод `__iter__()`, а вместо функции `next()` с аргументом-итератором из итератора вызывается метод `__next__()`. Несложно догадаться, что вызов функций `iter()` и `next()` приводит к вызову соответственно методов `__iter__()` и `__next__()`. Далее логика такая: нам нужно было бы описать класс с методом `__iter__()`.

Экземпляры такого класса - итерационные объекты. При вызове из итерационного объекта метода `__iter__()` в качестве результата должен возвращаться итератор - объект, у которого есть метод `__next__()`. Такой объект тоже нужно создавать на основе какого-то класса (точнее такого, в котором описан метод `__next__()`).

Естественным образом на ум приходит решение: описать один класс с методами `__iter__()` и `__next__()`, причем метод `__iter__()` определить так, чтобы в качестве результат он возвращал ссылку на экземпляр, из которого вызывается. В итоге класс нужен всего один, а экземпляр, который создается на основе этого класса, поддерживает оба метода `__iter__()` и `__next__()`. То есть получаем как бы итерационный объект и итератор "в одном лице".

Чтобы заработать такой двойной приз, нам нужен всего один класс. Об этом как раз и шла речь в начале раздела, когда мы говорили о классах-итераторах. Далее для простоты, и если это не будет приводить к недоразумениям, экземпляры классов-итераторов будем называть итераторами.

Итак, схема наших действий такая:

- Создаем класс с методами `__iter__()` и `__next__()`.
- Метод `__iter__()` описываем таким образом, чтобы результатом он возвращал ссылку на экземпляр, из которого вызывается.

- "Правильный" метод `__next__()` должен быть таким: какое-то количество вызовов метод возвращает значения, а затем генерирует исключение класса `StopIteration`.

В листинге 8.16 приведен пример создания итератора, который позволяет вычислять числа из последовательности Фибоначчи.

На заметку

Напомним, что в последовательности Фибоначчи первые два числа равны 1, а каждое следующее число равно сумме двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, и так далее.

Листинг 8.16. Итератор для чисел Фибоначчи

```
# Класс - итератор
class Fibs:
    # Конструктор
    def __init__(self,n):
        # Начальные значения для полей
        # count (номер генерируемого числа),
        # a (генерируемое число)
        # и b (следующее число)
        self.start()
        # Значение поля n (количество генерируемых
        # чисел)
        self.n=n
    # Метод для перевода в "начальное состояние"
    # полей count, a и b
    def start(self):
        self.count=1 # Номер генерируемого числа
        self.a=1     # Генерируемое число
        self.b=1     # Следующее число
    # Метод, который возвращает итератор
    def __iter__(self):
        # Результат метода - ссылка
        # на экземпляр класса
        return self
    # Метод для вычисления числа Фибоначчи
    def __next__(self):
        # Если превышен лимит генерируемых чисел
        if self.count>self.n:
            # Поля count, a и b получают начальные
            # единичные значения
            self.start()
            # Генерируется исключение
```

```
            # класса StopIteration
            raise StopIteration
        # Если лимит генерирования
        # чисел не превышен
        res=self.a # Возвращаемое методом значение
        # Новое число в последовательности Фибоначчи
        self.a,self.b=self.b,self.a+self.b
        # Новое значение поля count
        self.count+=1
        # Результат метода
        return res
# Создается экземпляр класса
obj=Fibs(10)
# Экземпляр класса используется в операторе цикла
for s in obj:
    print(s,end=" ")
print()
# Изменяем значения полей экземпляра класса
obj.a=3     # Четвертое число в последовательности
obj.b=5     # Пятое число в последовательности
obj.count=4 # Номер числа в последовательности
# Новый оператор цикла с экземпляром класса
for s in obj:
    print(s,end=" ")
print()
# Количество генерируемых чисел
obj.n=15
# Еще один оператор цикла с экземпляром класса
for s in obj:
    print(s,end=" ")
```

Результат выполнения этого программного кода такой:

Результат выполнения программы (из листинга 8.16)

```
1 1 2 3 5 8 13 21 34 55
3 5 8 13 21 34 55
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Класс, который мы создаем в программном коде и который предназначен для создания итераторов, называется `Fibs`. У экземпляра класса есть такие поля:

- Числовое поле `n` определяет количество чисел в последовательности Фибоначчи, которые будут генерироваться итератором (он же экземпляр класса).

- Числовое поле `count` определяет номер того элемента последовательности, который будет генерироваться при вызове функции `next()`. Очевидно, что генерирование чисел может осуществляться, если значение поля `count` не превышает значение поля `n`. В противном случае будет генерироваться исключение класса `StopIteration`.
- Поля `a` и `b` "помнят" два числа из последовательности Фибоначчи: в поле `a` записывается значение числа, которое будет сгенерировано при вызове функции `next()`, а в поле `b` записывается следующее число в последовательности.

Метод `start()` экземпляра класса нужен для установки начальных значений полей `a`, `b` и `count`. В теле метода все эти поля получают единичные значения.

В конструкторе вызывается метод `start()`, а также присваивается значение полю `n`. Значение для этого поля передается аргументом конструктору.

Метод `__iter__()` должен, как уже отмечалось, возвращать ссылку на итератор. Также мы условились, что итератором будет экземпляр класса `Fibs`. Поэтому в теле метода `__iter__()` с аргументом `self` всего одна инструкция `return self`, которой в качестве результата метода возвращается ссылка на экземпляр класса `Fibs`, из которого вызывается метод (и этот экземпляр, соответственно, передается аргументом функции `iter()`).

Наиболее интересный код, пожалуй, у метода `__next__()`. Именно в этом методе определяется, какое число будет генерироваться итератором (экземпляром класса) на каждой итерации. В теле метода в первую очередь в условном операторе проверяется условие `self.count>self.n` (через `self` обозначена ссылка на экземпляр класса - аргумент метода `__next__()`). Истинность данного условия означает, что числа больше генерироваться не должны. В этом случае командой `self.start()` полям `a`, `b` и `count` экземпляра `self` присваиваются единичные значения (как в начале итерационного процесса), а затем командой `raise StopIteration` генерируется исключение класса `StopIteration`.

На заметку

Обратите внимание, что исключение генерируется, но не перехватывается и не обрабатывается. Оно будет обработано оператором `for` и именно для этого оператора предназначено. Появление исключения служит признаком того, что итерационный процесс необходимо завершить.

Поскольку исключение не перехватывается, то `try`-блок мы не используем. Также после ключевого слова `raise` мы указали только имя класса исключения (а не инструкцию создания экземпляра, пускай даже анонимного). Так можно делать.

В этом случае создается анонимный экземпляр исключения соответствующего класса.

Если условие в условном операторе не выполнено, метод должен вернуть в качестве результата очередное число из последовательности Фибоначчи. В этом случае исключение не генерируется, зато выполняется последовательность таких команд:

- Командой `res=self.a` в локальную переменную `res` записывается значение поля `a`. Именно это число будет возвращаться как результат метода.
- Командой `self.a,self.b=self.b,self.a+self.b` вычисляется новое число в последовательности Фибоначчи, и одновременно присваиваются новые значения полям `a` и `b`. Здесь используется *множественное присваивание*: сначала по старым значениям полей `a` и `b` вычисляются выражения в правой части, а затем эти выражения присваиваются соответственно полю `a` и `b`.
- Командой `self.count+=1` значение поля `count` увеличивается на единицу (поскольку генерируется новое число).
- Наконец, командой `return res` значение переменной `res` возвращается как результат метода.

На этом описание класса `Fibs` завершается. Далее идут команды, которые иллюстрируют использование экземпляров этого класса в качестве итераторов. Так, командой `obj=Fibs(10)` создается экземпляр `obj` класса `Fibs`. Аргумент `10`, переданный конструктору, означает, что экземпляр `obj` позволит генерировать `10` первых чисел в последовательности Фибоначчи. Чтобы проверить это, запускается оператор цикла, в котором переменная `s` перебирает значения "из экземпляра" `obj`. За каждый цикл выполняется команда `print(s,end=" ")`, вследствие чего числа (значение переменной `s`) выводятся через пробел в одной строке. В результате получаем последовательность Фибоначчи из 10 чисел. Если после этого запустить еще раз оператор цикла, получим такой же результат. То есть экземпляр `obj` является "многоразовым" итератором - в том смысле, что его можно использовать много раз.

🕮 На заметку

Здесь кроется важный "идеологический" момент. Дело в том, что каждый раз при вызове метода `__next__()` меняется "внутреннее состояние" экземпляра, из которого вызывается метод: изменяются значения полей экземпляра. Теоретически после того, как при очередном вызове метода `__next__()` будет сгенерировано исключение класса `StopIteration` и итерационный процесс завершится, значе-

ния полей экземпляра будут совсем не такими, как в начале итерационного процесса. Поэтому чтобы экземпляр можно было использовать снова, необходимо вернуть его "внутренние настройки" в "начальное состояние". Именно с этой целью в теле метода `__next__()` перед генерированием исключения вызывался метод `start()`. Если этого не сделать, то получим "одноразовый" итератор: такой экземпляр можно будет использовать в операторе цикла единожды. При попытке использовать его в следующий раз никаких итераций не будет - исключение `StopIteration` сгенерируется при первом же вызове метода `__next__()`.

Можно было поступить и по-другому: поместить команду вызова метода `start()` не в теле метода `__next__()`, а в теле метода `__iter__()`. Желающие могут подумать, что бы изменилось в этом случае?

Чтобы подробнее раскрыть механизм генерирования чисел с помощью экземпляра `obj`, далее проводим небольшой эксперимент. Командами `obj.a=3`, `obj.b=5` и `obj.count=4` изменяем значения полей экземпляра `obj`. В данном случае поле `a` равно четвертому (по порядку) числу в последовательности Фибоначчи, следующее число в последовательности - это значение поля `b`, и значение поля `count` соответствует порядковому номеру числа, записанного в поле `a`. Если теперь запустить оператор цикла, то генерирование чисел начнется с числа с порядковым номером `4` в последовательности. Последнее сгенерированное число - десятое (в соответствии со значением поля `n` экземпляра `obj`, а оно равно `10`).

На заметку

После завершения оператора цикла поля `a`, `b` и `count` снова будут иметь единичные значения.

Если изменить значение поля `n` экземпляра `obj`, например, командой `obj.n=15`, то при очередном запуске оператора цикла будет генерироваться не `10`, а уже `15` чисел из последовательности Фибоначчи.

В известном смысле *функция-генератор* может рассматриваться как "упрощенная" и "облегченная" форма для создания итератора или итерационного объекта (тоже "в одном лице"). Как и итератор, результат вызова функции-генератора при каждом очередном обращении к нему дает новое значение из определенного набора значений. Для простоты будем называть результат вызова функции-генератора *объектом генератора.* Объект генератора используется по той же схеме, что и итератор.

При создании функции-генератора не нужно явно описывать методы `__next__()` и `__iter__()` (они создаются автоматически). Нет необходимости явно создавать объект генератора (результат функции). Все, что нужно сделать - "сформировать" последовательность значений, которые будут возвращаться объектом генератора. Главное формальное отли-

чие функции-генератора от обычной функции состоит в том, что результат функции-генератора формируется с помощью ключевого слова `yield` (а не возвращается инструкцией `return`, как у обычных функций). Пример функции-генератора, предназначенной для генерирования чисел из последовательности Фибоначчи, приведен в листинге 8.17.

Листинг 8.17. Функция-генератор для чисел Фибоначчи

```
# Функция-генератор для чисел Фибоначчи
def fibs_gen(n):
    # Первое число последовательности
    a=1
    # Второе число последовательности
    b=1
    # Оператор цикла для вычисления
    # чисел Фибоначчи
    for i in range(n):
        # Значение для результата
        # функции (для данной итерации)
        res=a
        # Вычисление нового значения в
        # последовательности и присваивание
        # значений переменным a и b
        a,b=b,a+b
        # Результат функции-генератора
        yield res
print("Попытка №1")
# Оператор цикла для вывода 10 чисел
# из последовательности Фибоначчи
for s in fibs_gen(10):
    print(s,end=" ")
print("\nПопытка №2")
# Еще один оператор цикла для вывода
# 10 чисел из последовательности Фибоначчи
for s in fibs_gen(10):
    print(s,end=" ")
print("\nПопытка №3")
# Ссылка на объект генератора записана
# в переменную
f=fibs_gen(15)
# Оператор цикла для вывода 15 чисел
# из последовательности Фибоначчи.
# Ссылка на объект генератора выполнена
# через переменную
for s in f:
    print(s,end=" ")
```

```
print("\nПопытка №4")
# Еще один оператор цикла для вывода
# 15 чисел из последовательности Фибоначчи.
# Ссылка на объект генератора выполнена
# через переменную
for s in f:
    print(s,end=" ")
print("Завершение работы.")
```

Результат выполнения программного кода такой:

Результат выполнения программы (из листинга 8.17)

```
Попытка №1
1 1 2 3 5 8 13 21 34 55
Попытка №2
1 1 2 3 5 8 13 21 34 55
Попытка №3
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Попытка №4
Завершение работы.
```

В этом примере, как отмечалось выше, мы создаем функцию-генератор для чисел Фибоначчи. Функция называется `fibs_gen()` и у нее один аргумент (обозначен как `n`) - количество генерируемых функцией чисел. В теле функции переменным `a` и `b` присваиваются единичные значения. Это первые два числа последовательности. Затем запускается оператор цикла, в котором переменная `i` пробегает значение от `0` до `n-1`. Конкретные границы диапазона изменения переменной `i` не важны, поскольку в теле цикла эта переменная явно не используется. Главное, что цикл состоит из `n` итераций. За каждую итерацию командой `res=a` в переменную `res` записывается текущее значение переменной `a`. Это значение будет "возвращено" (на данной итерации) как результат функции. Затем командой `a,b=b,a+b` с использованием множественного присваивания вычисляется новое число в последовательности Фибоначчи, и переменные `a` и `b` получают новые значения.

На заметку

Выражение `a,b=b,a+b` вычисляется так. Для текущих значений переменных `a` и `b` вычисляются значения выражений `b` (обозначим как `значение_1`) и `a+b` (обозначим как `значение_2`) в правой части от оператора присваивания. Затем переменной `a` присваивается вычисленное на предыдущем этапе `значение_1`, а переменной `b` присваивается вычисленное на предыдущем этапе `значение_2`.

После этого командой `yield res` формируется результат (для данной итерации).

На заметку

Что касается результата функции-генератора, то разумнее на все это смотреть как на последовательность значений, которые возвращаются в процессе выполнения программного кода функции. Команда с инструкцией `yield` дает, или добавляет новый элемент в эту последовательность. Нередко для формирования такой последовательности результатов прибегают к помощи оператора цикла. Однако это не обязательно. Например, можем определить такую функцию-генератор:

```
def colors():
    yield "Красный"
    yield "Желтый"
    yield "Зеленый"
```

Тогда при выполнении оператора цикла

```
for clr in colors():
    print(clr)
```

получим следующий результат:

```
Красный
Желтый
Зеленый
```

Для проверки работы функции запускаем оператор цикла, в котором переменная `s` пробегает значения из объекта генератора, который вычисляется инструкцией `fibs_gen(10)` (вызываем функцию `fibs_gen()` с аргументом `10`). В теле оператора цикла выполняется команда `print(s,end=" ")`. Как следствие, в окне вывода в строчку через пробел выводится `10` чисел из последовательности Фибоначчи.

Что происходит при вызове функции-генератора `fibs_gen()`? При вызове функции-генератора создается объект генератора. Для объекта генератора можно вызвать функции `iter()` и `next()` (что неявно и происходит при использовании функции-генератора в операторе цикла). Однако здесь есть важное обстоятельство: созданный в результате вызова функции-генератора объект генератора "одноразовый". Его можно использовать в операторе цикла только один раз.

На первый взгляд это может показаться странным. Особенно если учесть, что повторно выполненный оператор цикла в нашем примере дает точно такой же результат, как и первый оператор цикла. Но все становится на свои места, если учесть, что каждый раз при вызове функции-генератора (даже если с теми же аргументами) создается новый объект. Он и используется. Чтобы убедиться в справедливости данного утверждения, выполним простую провеерку.

Сначала командой f=fibs_gen(15) в переменную f записывается ссылка на объект генератора, созданный вызовом функции-генератора fibs_gen(15). Затем в операторе цикла указываем переменную f. Сначала все происходит вполне ожидаемо: в окне вывода отображается 15 чисел из последовательности Фибоначчи. Но если мы попытаемся выполнить точно такой же оператор с той же самой переменной f, ни одно число выведено не будет.

Причина в том, что переменная f ссылается на тот же самый объект, что использовался в первом (из двух последних) операторе цикла. И этот объект свою задачу выполнил. Больше от него пользы нет.

На заметку

Классы-итераторы и функции-генераторы можно использовать не только для создания объектов, служащих "контейнерами" перебираемых значений в операторах цикла. На основе итераторов можно создавать, например, списки. Для этого достаточно итератор передать аргументом функции list(). Скажем, результатом инструкции list(fibs_gen(15)) будет список из 15 чисел из последовательности Фибоначчи (функция fibs_gen() описана в листинге 8.17). К аналогичному результату приведет выполнение инструкции list(Fibs(15)) (класс Fibs описан в листинге 8.16).

Кроме того, имеются некоторые встроенные функции, которые позволяют просто и быстро создавать объекты итерационного типа. В качестве примера можно привести функцию map(). Если первым аргументом функции map() передать имя некоторой функции, а вторым аргументом - список значений (команда формата map(функция, список)), то в результате получим объект итерационного типа. Каждое значение, возвращаемое при обращении к этому объекту с помощью функции next(), получается вызовом функции, указанной первым аргументом в map(), с аргументом, который является элементом списка, переданного вторым аргументом функции map(). Если использовать инструкцию вида list(map(функция, список)), в результате получим список, элементы которого получаются вызовом функции с аргументом, который пробегает значения элементов списка. Например, результатом выражения list(map(lambda n: 2**n,[1, 2,3])) будет список [2,4,8].

По сходному принципу действует функция filter(). Результатом выражения вида list(filter(функция, список)) является список, который получается из списка, переданного аргументом функции filter() - но только тех элементов, для которых вызов функции (первый аргумент функции filter()) возвращает значение True. То есть фактически в данном случае можно выполнять "фильтрацию" списков. Имя функции, играющей роль критерия включения элемента в список-результат, указывается первым аргументом функции filter(). Например, результатом выражения list(filter(lambda x: bool(x%2),[1,5,6,2,7,8])) будет список [1,5,7] (только нечетные элементы).

Функция zip() позволяет из нескольких списков создать список кортежей, где каждый кортеж получается включением соответствующих элементов из исходных списков. Например, если x=[1,2,3], y=[4,5,6] и z=[7,8,9], то результатом выражения list(zip(x,y,z)) будет список [(1,4,7),(2,5,8),(3,6,9)].

Следует также иметь в виду, что создаваемые в результате вызова функций `map()`, `filter()` и `zip()` объекты итерационного типа являются "одноразовыми" в том контексте, как это описывалось выше. Так, если выполнить команду `obj=map(lambda n: 2**n,[1,2,3])`, а затем инструкцию `list(obj)`, то вполне ожидаемо получим в результате список `[2,4,8]`. Но если на следующем этапе использовать объект `obj` в операторе цикла, итераций не будет! Хотя если перед оператором цикла инструкцию `list(obj)` не выполнять, то оператор цикла выполняется. Причина в том, что объект `obj` может использоваться только один раз. Мы с такой ситуацией уже сталкивались.

Резюме

Он начинает новую жизнь, дайте ему возможность вспомнить все лучшее.
из к/ф "Покровские ворота"

1. В Python можно создавать функции с переменным (нефиксированным) количеством аргументом. Весь набор таких аргументов обрабатывается как список (кортеж), а при описании функции обозначается как один аргумент, но со звездочкой * перед названием аргумента.

2. Аргумент, помеченный двойной звездочкой **, обозначает словарь с именованными аргументами (переданными функции по ключу).

3. Декораторы позволяют изменять поведение и свойства функций и классов. Декоратор для функций создается на основе функции, которой аргументом передается функция и которая возвращает в качестве результата функцию. Декоратор класса создается на основе функции, аргументом которой является класс, и результатом тоже является класс. Декоратор (символ @ и имя функции для создания декоратора) указывается перед именем функции или класса. В результате функция декоратора применяется к "декорируемой" функции или классу. Результат применения декоратора к функции или классу записывается в переменную, которая служит названием соответственно функции или класса.

4. Первая текстовая строка в описании функции служит в качестве текста справки по функции и возвращается как значение поля `__doc__` объекта функции.

5. Для аргументов и результата функции можно создавать аннотации: специальные текстовые "пояснения". Аннотация для аргументов указывается через двоеточие после имени аргументов в описании функции. Аннотация для результата указывается после закрывающих круглых скобок (в которых описываются аргументы функции) через инструкцию ->. Наличие аннотаций не накладывает ограничений на тип аргументов или результата функции.

6. Классы встроенных исключений образуют иерархию, в вершине которой находится класс `BaseException`. Другой важный класс - класс `Exception`. Большинство классов исключений являются производными классами от класса `Exception`.

7. В `except`-блоке перехватываются не только те исключения, которые указаны явно, но еще и исключения производных классов.

8. При обработке исключения в `except`-блоке может использоваться ссылка на экземпляр исключения.

9. Исключение может быть сгенерировано вручную с помощью инструкций `raise` и `assert`.

10. Пользовательские классы исключений создаются наследованием классов встроенных исключений (обычно базовым является класс `Exception`).

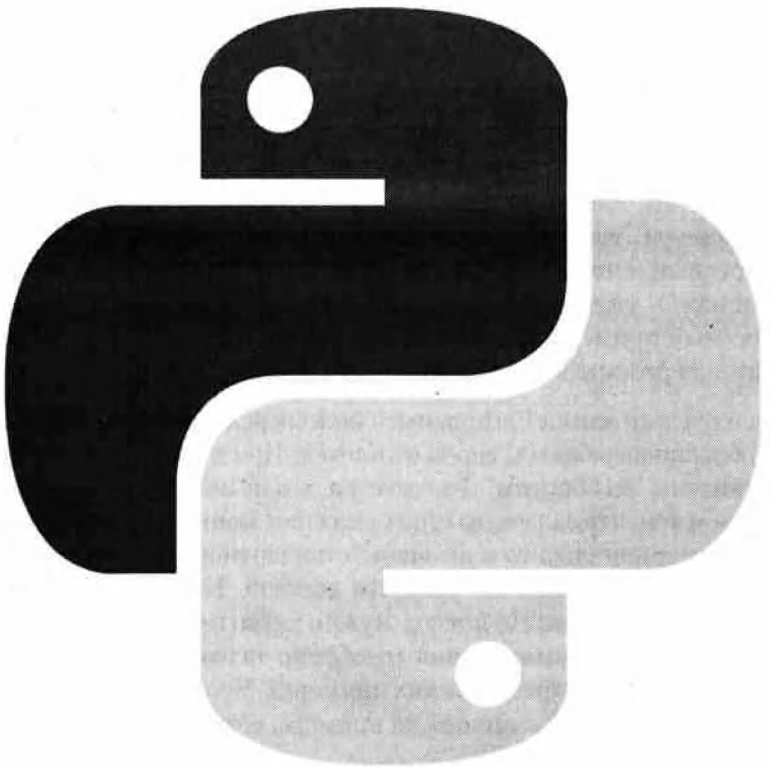
11. Итераторы создаются на основе классов, содержащих описание методов `__iter__()` и `__next__()`. Метод `__iter__()` возвращает в качестве результата ссылку на экземпляр, из которого был вызван. При вызове метода `__next__()` каждый раз возвращается некоторое значение, пока при очередном вызове не будет сгенерировано исключение класса `StopIteration`.

12. Итератор можно использовать в операторе цикла (как контейнер для перебора значений в цикле), или, например, для формирования списков.

13. Функция-генератор возвращает объект, который может использоваться как итератор. В теле функции-генератора результат формируется с помощью инструкции `yield`.

Заключение

О чем мы не поговорили



Дорогу осилит идущий. Вам необходим творческий непокой.
из к/ф "Покровские ворота"

Обычно о языках программирования говорят и пишут сухим техническим языком. В принципе это правильно. Но вот что касается языка Python, то "лирические" эпитеты приходят сами, непроизвольно, так сказать. Их много. Но если попытаться охарактеризовать язык одним словом, то, пожалуй, "многоликий" будет удачным вариантом.

На заметку

С другой стороны, "гибкий" - тоже звучит неплохо.

Как бы там ни было, а изучение языка Python - процесс творческий и, как правило, долгий. Азы, основы языка можно освоить достаточно легко. Но вот что касается "тонкостей", то это процесс небыстрый - правда, во многом интересный и неожиданный. В какой-то момент может показаться, что все точки над "i" уже расставлены. Но не тут то было! Обязательно найдется интересный пример, понять который сразу может быть сложно. Иногда приходится переосмыслить все, что до этого узнал.

Мы, при изучении языка Python, пытались сосредоточиться на главном, отделить, образно выражаясь, зерна от плевел. При этом некоторые моменты были оставлены "за боротом". Разумеется, это не очень хорошо. Но мы себя оправдываем тем, что задача-то стоит сложная: понять и освоить принципы достаточно нетривиального и не очень "стандартного" языка программирования. Тут, как говорится, все средства хороши. Насколько хороши наши средства, покажет время. Но вот что нужно четко понимать - так это то, что учиться языку программирования (особенно такому, как Python) следует постоянно, и лучше на практических примерах. Чтобы не быть голословными, проиллюстрируем сказанное - на примере, естественно.

Мы знаем, что у аргументов функций могут быть значения по умолчанию. Более того, мы этой особенностью функций не раз пользовались. Рассмотрим функцию со следующим программным кодом:

```
def f(A=[0]):
    A[0]+=1
    return A[0]
```

Здесь у функции `f()` объявлен один аргумент `A`, у которого есть значение по умолчанию - это список `[0]`, состоящий всего из одного элемента (число `0`). В общем случае предполагается, что аргумент `A` функции `f()` - список. В теле функции командой `A[0]+=1` на единицу увеличивается значение первого элемента в списке, переданном аргументом функции. Это новое значение элемента возвращается как результат функции.

Чего можно ожидать от такой функции? Можно ожидать, что список, переданный аргументом функции, будет изменен. Это действительно так. Например, если список `B=[10,20,30]`, то результатом выражения `f(B)` будет число `11`, а список `B` после выполнения команды будет равен `[11,20,30]` (точнее, переменная `B` будет ссылаться на такой список). Также можно ожидать, что если мы вызовем функцию `f()` без аргументов, то результатом будет число `1`. Это так. Но только первый раз. Если мы еще раз вычислим выражение `f()`, в результате получим значение `2`, затем - значение `3`, и так далее. Желающие могут проделать эксперимент: после описания функции `f()` выполнить оператор цикла

```
for s in range(10):
    print(f(),end=" ")
```

В результате в области вывода появится строка из десяти натуральных чисел

```
1 2 3 4 5 6 7 8 9 10
```

Таким образом, будучи вызванная без аргументов, функция `f()` каждый раз возвращает новое значение - на единицу больше предыдущего. В чем причина? А причина в *значении по умолчанию* для аргумента функции. Если более конкретно, то происходит следующее. При создании объекта для функции `f()` в памяти выделяется место под список `[0]`. Это как бы значение аргумента по умолчанию. "Как бы" - потому что на самом деле значением по умолчанию является ссылка на список. Эта ссылка остается неизменной от запуска к запуску функции. Но сам список меняется каждый раз, когда вызывается функция. Отсюда и такой немного неожиданный результат (хотя вполне логичный, если во всем разобраться).

Следующая иллюстрация будет скорее о том, как при составлении программных кодов на языке Python вносить в процесс "творческое начало". Итак, допустим, имеется описание двух классов. Класс A:

```
class A:
    def show(self):
        print("Это класс A!")
```

В этом классе всего один метод `show()`, которым отображается сообщение. Что касается класса B, то он практически такой же, как и класса A:

```
class B:
    def show(self):
        print("Это класс B!")
```

Имя класса - это на самом деле переменная, которая ссылается на объект класса. На объект класса может ссылаться больше, чем одна переменная. Например, законной является команда `MyClass=A`. В результате переменная `MyClass` будет ссылаться на тот же объект класса, что и переменная A. Другими словами, переменная `MyClass` является "синонимом" названия класса A. Теперь путем наследования создаем класс `Stranger`:

```
class Stranger(MyClass):
    def show(self):
        MyClass.show(self)
        print("Класс Stranger!")
```

Здесь две особенности. Во-первых, в качестве имени базового класса мы указали переменную `MyClass`, а не A. Во-вторых, при переопределении метода `show()` в классе `Stranger` вызывается версия этого метода из базового класса, причем в ссылке на этот метод явно указано имя класса `MyClass`.

Далее командой `obj=Stranger()` создаем экземпляр `obj` класса `Stranger`. Если после этого выполнить команду `obj.show()`, результатом будут такие сообщения:

```
Это класс A!
Класс Stranger!
```

А затем мы выполняем команду `MyClass=B`, которой ссылка в переменной `MyClass` с объекта класса A перебрасывается на объект класса B. После проделанных манипуляций результатом команды `obj.show()` будут сообщения

```
Это класс B!
Класс Stranger!
```

Желающие могут подумать, почему так происходит. Возможно, в процессе поисков ответа на этот вопрос читатель найдет для себя что-то новое и интересное. Ведь поиск ответов на вопросы - лучший путь для саморазвития.

Для записей

Васильев А. Н.

Python на примерах

Практический курс по программированию

В этой книге речь будет идти о том, как писать программы на языке программирования, который называется Python (правильно читается как пайтон, но обычно название языка читают как питон, что тоже вполне приемлемо). Таким образом, решать будем две задачи, одна из которых приоритетная, а вторая, хотя и вспомогательная, но достаточно важная. Наша основная задача, конечно же, изучение синтаксиса языка программирования Python. Параллельно мы будем осваивать программирование как таковое, явно или неявно принимая во внимание, что соответствующие алгоритмы предполагается реализовывать на языке Python.

Большинство авторов книг в своих трудах рассматривают теоретические основы языка и уделяют основное внимание базовому синтаксису языка, не рассматривая при этом практическую сторону его применения. Эта же книга старается восполнить недостаток практического материала, содержит множество примеров с комментариями, которые вы сможете использовать в качестве основы своих программных решений, изучения Python.

Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений.

www.nit.com.ru

ISBN 978-5-94387-995-1

9 785943 879951

НиТ издательство

Россия: Санкт-Петербург,
пр. Обуховской обороны, 107
для писем: 192029, Санкт-Петербург, а/я 44
(812) 412 7025, 412 7026
e-mail: nit@mail.wplus.net

Украина: 02166, Киев, ул. Курчатова, 9/21
(044) 516 3866
e-mail: nits@voliacable.com