

10.1

Процедурное и объектно-ориентированное
программирование**Ключевые положения**

Процедурное программирование представляет собой метод написания программного обеспечения. Это практика программирования, в центре внимания которой находятся процедуры или действия, происходящие в программе. Основой объектно-ориентированного программирования служат объекты, которые создаются из абстрактных типов данных, объединяющих данные и функции.

В настоящее время применяются главным образом два метода программирования: процедурный и объектно-ориентированный. Первые языки программирования были процедурными, т. е. программа состояла из одной или нескольких процедур. *Процедура* может рассматриваться просто как функция, которая выполняет определенную задачу, такую как сбор вводимых пользователем данных, выполнение вычислений, чтение или запись файлов, вывод результатов и т. д. Программы, которые вы писали до сих пор, были по своей природе процедурными.

Как правило, процедуры оперируют элементами данных, которые существуют отдельно от процедур. В процедурной программе элементы данных обычно передаются из одной процедуры в другую. Как можно предположить, в центре процедурного программирования находится создание процедур, которые оперируют данными программы. По мере увеличения и усложнения программы разделение данных и программного кода, который оперирует данными, может привести к проблемам.

Например, предположим, что вы являетесь участником команды программистов, которая написала масштабную программу обработки базы данных клиентов. Эта программа первоначально разрабатывалась с использованием трех переменных, которые ссылаются на имя, адрес и телефонный номер клиента. Ваша работа состояла в том, чтобы разработать несколько функций, которые принимают эти три переменные в качестве аргументов и выполняют с ними операции. Созданный программный продукт успешно работал в течение некоторого времени, однако вашу команду попросили его обновить, внедрив несколько новых возможностей. Во время пересмотра версии ведущий программист вам сообщает, что имя, адрес и телефонный номер клиента больше не будут храниться в переменных. Вместо этого они будут храниться в списке. Это означает, что вам необходимо изменить все разработанные вами функции таким образом, чтобы они принимали список и работали с ним вместо этих трех переменных. Внесение таких масштабных модификаций не только предполагает большой объем работы, но и открывает возможность для внесения ошибок в программный код.

В отличие от процедурного программирования, в центре внимания которого находится создание процедур (функций), *объектно-ориентированное программирование (ООП)* сосредоточено на создании объектов. *Объект* — это программная сущность, которая содержит данные и процедуры. Находящиеся внутри объекта данные называются *атрибутами данных*. Это просто переменные, которые ссылаются на данные. Выполняемые объектом процедуры называются *методами*. Методы объекта — это функции, которые выполняют операции с атрибутами данных. В концептуальном плане объект представляет собой автономную единицу, которая состоит из атрибутов данных и методов, которые оперируют атрибутами данных (рис. 10.1).

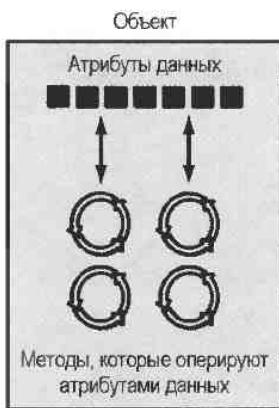


РИС. 10.1. Объект содержит атрибуты данных и методы

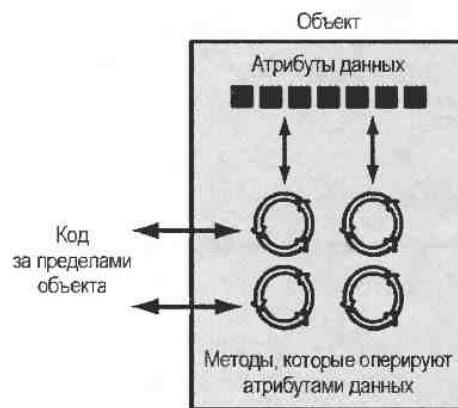


РИС. 10.2. Программный код за пределами объекта взаимодействует с методами объекта

ООП решает проблему разделения программного кода и данных посредством инкапсуляции и сокрытия данных. *Инкапсуляция* обозначает объединение данных и программного кода в одном объекте. *Скрытие данных* связано со способностью объекта скрывать свои атрибуты данных от программного кода, который находится за пределами объекта. Только методы объекта могут непосредственно получать доступ и вносить изменения в атрибуты данных объекта.

Объект, как правило, скрывает свои данные, но позволяет внешнему коду получать доступ к своим методам. Как показано на рис. 10.2, методы объекта предоставляют программным инструкциям за пределами объекта косвенный доступ к атрибутам данных объекта.

Когда атрибуты данных объекта скрыты от внешнего кода, и доступ к атрибутам данных ограничен методами объекта, атрибуты данных защищены от случайного повреждения. Кроме того, программному коду за пределами объекта не нужно знать о формате или внутренней структуре данных объекта. Программный код взаимодействует только с методами объекта. Когда программист меняет структуру внутренних атрибутов данных, он также меняет методы объекта, чтобы они могли должным образом оперировать данными. Однако приемы взаимодействия внешнего кода с методами не меняются.

Возможность многократного использования объекта

В дополнение к решению проблем разделения программного кода и данных, применение ООП также всецело поддерживалось трендом на многократное использование объектов. Объект не является автономной программой. Напротив, он используется программами,

которым нужны его услуги. Например, Шэрон является программистом, и она разработала ряд объектов для визуализации 3D-изображений. Она эрудит в математике и очень много знает о компьютерной графике, поэтому ее объекты запрограммированы на выполнение всех необходимых математических операций с 3D-графикой и взаимодействие с компьютерным видеооборудованием. Для Тома, который пишет программу по заказу архитектурной фирмы, требуется, чтобы его приложение выводило 3D-изображения зданий. Поскольку он работает в рамках жестких сроков и не обладает большим объемом знаний в области компьютерной графики, то может применить объекты Шэрон, чтобы выполнить 3D-визуализацию (за небольшую плату, разумеется!).

Пример объекта из повседневной жизни

Предположим, что ваш будильник — это на самом деле программный объект. Будь это так, то он имел бы приведенные ниже атрибуты данных:

- ◆ `current_second` (текущая секунда, значение в диапазоне 0–59);
- ◆ `current_minute` (текущая минута, значение в диапазоне 0–59);
- ◆ `current_hour` (текущий час, значение в диапазоне 1–12);
- ◆ `alarm_time` (время сигнала, допустимые час и минута);
- ◆ `alarm_is_set` (будильник включен, истина или ложь).

Этот пример четко показывает, что атрибуты данных — это всего-навсего значения, которые определяют *состояние*, в котором будильник находится в настоящее время. Вы, пользователь объекта "Будильник", не можете непосредственно манипулировать этими атрибутами данных, потому что они являются *приватными*, или частными. Для того чтобы изменить значение атрибута данных, необходимо применить один из методов объекта. Ниже приведено несколько методов объекта "Будильник":

- ◆ `set_time` (задать время);
- ◆ `set_alarm_time` (задать время сигнала);
- ◆ `set_alarm_on` (включить будильник);
- ◆ `set_alarm_off` (выключить будильник).

Каждый метод манипулирует одним или несколькими атрибутами данных. Например, метод `set_time()` позволяет устанавливать время будильника и активируется нажатием кнопки вверху часов. При помощи другой кнопки можно активировать метод `set_alarm_time()`.

Еще одна кнопка позволяет выполнять методы `set_alarm_on()` и `set_alarm_off()`. Обратите внимание, что все эти методы могут быть активированы вами, т. е. тем, кто находится за пределами будильника. Методы, к которым могут получать доступ объекты, находящиеся за пределами объекта, называются *открытыми*, или *публичными методами*.

Будильник также имеет закрытые, или *приватные методы*, которые являются составной частью приватного, внутреннего устройства объекта. Внешние сущности (такие как вы, пользователь будильника) не имеют прямого доступа к приватным методам будильника. Объект предназначен выполнять эти методы автоматически и скрывать детали от вас. Ниже приведены приватные методы объекта "Будильник":

- ◆ `increment_current_second()` (прирастить текущую секунду);
- ◆ `increment_current_minute()` (прирастить текущую минуту);

- ◆ `increment_current_hour()` (прирастить текущий час);
- ◆ `sound_alarm()` (подать звуковой сигнал).

Метод `increment_current_second()` выполняется каждую секунду. Он изменяет значение атрибута данных `current_second`. Если при исполнении этого метода атрибут данных `current_second` равняется 59, то этот метод запрограммирован сбросить `current_second` в значение 0 и затем вызвать метод `increment_current_minute()`, который добавляет 1 к атрибуту данных `current_minute`, если он не равен 59. В противном случае он сбрасывает `current_minute` в значение 0 и вызывает метод `increment_current_hour()`. Метод `increment_current_minute()` сравнивает новое время с `alarm_time`. Если оба времени совпадают и при этом звонок включен, исполняется метод `sound_alarm()`.



Контрольная точка

- 10.1. Что такое объект?
- 10.2. Что такое инкапсуляция?
- 10.3. Почему внутренние данные объекта обычно скрыты от внешнего программного кода?
- 10.4. Что такое публичные методы? Что такое приватные методы?

10.2 Классы

Ключевые положения

Класс — это программный код, который задает атрибуты данных и методы для объекта определенного типа.



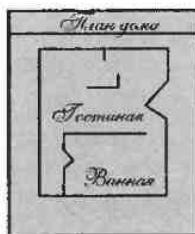
Видеозапись "Классы и объекты" (*Classes and Objects*)

Теперь давайте обсудим, каким образом объекты создаются в программном коде. Прежде чем объект будет создан, он должен быть разработан программистом. Программист определяет атрибуты данных, необходимые методы и затем создает класс. Класс — это программный код, который задает атрибуты данных и методы объекта определенного типа. Представьте класс как "строительный проект", на основе которого будут возводиться объекты. Класс выполняет аналогичные задачи, что и проект дома. Сам проект не является домом, он выступает подробным описанием дома. Когда для возведения реального дома используется строительный проект, обычно говорят, что типовой дом (экземпляр дома) возводится согласно проекту. По желанию заказчика может быть построено несколько идентичных зданий на основе одного и того же проекта. Каждый возведенный дом является отдельным экземпляром дома, описанного в проекте. Эта идея проиллюстрирована на рис. 10.3.

Разницу между классом и объектом можно представить по-другому, если задуматься о различии между формой для печенья и печеньем. Форма для печенья не является печеньем, вместе с тем она дает описание печенья. Форма для печенья может использоваться для изготовления одной или нескольких штук печенья. Представьте класс как форму для печенья, а также объекты, созданные на основе класса, как отдельные печеньшки.

Итак, класс — это описание свойств объекта. Когда программа работает, она может использовать класс для создания в оперативной памяти такого количества объектов определенного типа, какое понадобится. Каждый объект, который создается на основе класса, называется **экземпляром класса**.

Проект, который описывает дом



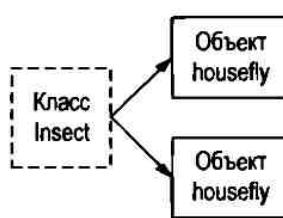
Экземпляры домов, возведенных по проекту



РИС. 10.3. Проект и дома, возведенные по проекту

Например, Джессика является энтомологом (специалистом, изучающим насекомых), и она также любит писать компьютерные программы. Она разрабатывает программу для каталогизации различных типов насекомых. В рамках программы она создает класс *Insect* (Насекомое), который задает свойства, характерные для всех типов насекомых. Класс *Insect* представляет собой спецификацию, согласно которой создаются объекты. Далее она пишет программные инструкции, создающие объект под названием *housefly* (комнатная муха), который является экземпляром класса *Insect*. Объект *housefly* — это сущность, которая занимает место в оперативной памяти компьютера и там хранит данные о комнатной мухе. Этот объект имеет атрибуты данных и методы, заданные классом *Insect*. Затем Джессика пишет программные инструкции, которые создают объект под названием *mosquito* (комар). Объект *mosquito* тоже является экземпляром класса *Insect*. Он занимает собственную область в оперативной памяти и там хранит данные о комаре. И хотя объекты *housefly* и *mosquito* в оперативной памяти компьютера являются отдельными объектами, они оба были созданы на основе класса *Insect*. Другими словами, каждый из этих объектов имеет атрибуты данных и методы, описанные классом *Insect*. Это проиллюстрировано на рис. 10.4.

Класс *Insect* описывает атрибуты данных и методы, которые может иметь объект определенного типа



Объект *housefly* — это экземпляр класса *Insect*. Он имеет атрибуты данных и методы, описанные классом *Insect*

Объект *mosquito* — это экземпляр класса *Insect*. Он имеет атрибуты данных и методы, описанные классом *Insect*

РИС. 10.4. Объекты *housefly* и *mosquito* являются экземплярами класса *Insect*

Определения классов

Для того чтобы создать класс, пишут *определение класса* — набор инструкций, которые задают *методы класса* и *атрибуты данных*. Давайте взглянем на простой пример. Предположим, что мы пишем программу для имитации бросания монеты. В программе мы должны неоднократно имитировать подбрасывание монеты и всякий раз определять, приземлилась ли она орлом либо решкой. Принимая объектно-ориентированный подход, мы напишем класс *Coin* (Монета), который может выполнять действия монеты.

В программе 10.1 представлено определение класса, пояснения которого будут даны далее, а сейчас обратите внимание, что эта программа не полная. Мы будем добавлять в нее код по мере продвижения.

Программа 10.1 (класс *Coin*, незаконченная программа)

```
1 import random
2
3 # Класс Coin имитирует монету, которую
4 # можно подбрасывать.
5
6 class Coin:
7
8     # Метод __init__ инициализирует
9     # атрибут данных sideup значением 'Орел'.
10
11    def __init__(self):
12        self.sideup = 'Орел'
13
14    # Метод toss генерирует случайное число
15    # в диапазоне от 0 до 1. Если это число
16    # равно 0, то sideup получает значение 'Орел'.
17    # В противном случае sideup получает значение 'Решка'.
18
19    def toss(self):
20        if random.randint(0, 1) == 0:
21            self.sideup = 'Орел'
22        else:
23            self.sideup = 'Решка'
24
25    # Метод get_sideup возвращает значение,
26    # на которое ссылается sideup.
27
28    def get_sideup(self):
29        return self.sideup
```

В строке 1 мы импортируем модуль *random*, т. к. для генерации случайного числа мы применяем функцию *randint*. Стока 6 является началом определения класса. Оно начинается с ключевого слова *class*, за которым идет имя класса, т. е. *Coin*, и потом двоеточие.

К именам классов применимы те же самые правила, которые применимы к именам переменных. Однако следует учесть, что мы начинаем имя класса, `Coin`, с заглавной буквой. Такое написание необязательно. Оно является широко распространенным среди программистов соглашением о наименовании классов. При чтении программного кода такое написание помогает легко отличать имена классов от имен переменных.

Класс `Coin` имеет три метода:

- ◆ метод `__init__()` появляется в строках 11–12;
- ◆ метод `toss()` (подбрасывать) — в строках 19–23;
- ◆ метод `get_sideup()` (получить обращенную вверх сторону монеты) — в строках 28–29.

Обратите внимание, что за исключением того, что эти определения методов появляются в классе, они похожи на любое другое определение функции в Python. Они начинаются со строки заголовка, после которой идет выделенный отступом блок инструкций.

Взглядите на заголовок каждого определения метода (строки 11, 19 и 28) и обратите внимание, что каждый метод имеет параметрическую переменную с именем `self`:

- ◆ строка 11: `def __init__(self):`
- ◆ строка 19: `def toss(self):`
- ◆ строка 28: `def get_sideup(self):`

Параметр `self`¹ требуется в каждом методе класса. Вспомните из предшествующего обсуждения ООП, что метод оперирует атрибутами данных конкретного объекта. Во время исполнения метод должен иметь возможность знать, атрибутами данных какого объекта он призван оперировать. Именно здесь на первый план выходит параметр `self`. Когда метод вызывается, Python делает так, что параметр `self` ссылается на конкретный объект, которым этот метод призван оперировать.

Давайте рассмотрим каждый из этих методов. Первый метод с именем `__init__()` определен в строках 11–12:

```
def __init__(self):
    self.sideup = 'Орел'
```

Большинство классов Python имеет специальный метод `__init__()`, который автоматически исполняется, когда экземпляр класса создается в оперативной памяти. Метод `__init__()` обычно называется *методом инициализации*, потому что он инициализирует атрибуты данных объекта. (Название метода состоит из двух символов подчеркивания, слова `init` и еще двух символов подчеркивания.)

Сразу после создания объекта в оперативной памяти исполняется метод `__init__()`, и параметру `self` автоматически присваивается объект, который был только что создан. Внутри этого метода исполняется инструкция в строке 12:

```
self.sideup = 'Орел'
```

Эта инструкция присваивает строковый литерал 'Орел' атрибуту данных `sideup`, принадлежащему только что созданному объекту. В результате работы метода `__init__()` каждый объект, который мы создаем на основе класса `Coin`, будет первоначально иметь атрибут `sideup` с заданным значением 'Орел'.

¹ Присутствие этого параметра в методе обязательно. Вы не обязаны называть его `self`, однако строго рекомендуется действовать согласно общепринятой практике.



ПРИМЕЧАНИЕ

Метод `__init__()` обычно является первым методом в определении класса.

Метод `toss()` расположен в строках 19–23:

```
def toss(self):
    if random.randint(0, 1) == 0:
        self.sideup = 'Орел'
    else:
        self.sideup = 'Решка'
```

Этот метод тоже имеет необходимую параметрическую переменную `self`. При вызове метода `toss()` параметрическая переменная `self` автоматически будет ссылаться на объект, которым этот метод должен оперировать.

Метод `toss()` имитирует подбрасывание монеты. Когда он вызывается, инструкция `if` в строке 20 вызывает функцию `random.randint` для получения случайного целого числа в диапазоне от 0 до 1. Если число равняется 0, то инструкция в строке 21 присваивает атрибуту `self.sideup` значение 'Орел'. В противном случае инструкция в строке 23 присваивает атрибуту `self.sideup` значение 'Решка'.

Метод `get_sideup()` появляется в строках 28–29:

```
def get_sideup(self):
    return self.sideup
```

Как и ранее, этот метод имеет необходимую параметрическую переменную `self` и просто возвращает значение атрибута `self.sideup`. Данный метод вызывается в любое время, когда возникает необходимость узнать, какой стороной монета обращена вверх.

Для того чтобы продемонстрировать класс `Coin`, необходимо написать законченную программу, которая его использует для создания объекта. В программе 10.2 приведен такой пример. Определение класса `Coin` находится в строках 6–29, а главная функция расположена в строках 32–44.

Программа 10.2 (coin_demo1.py)

```
1 import random
2
3 # Класс Coin имитирует монету, которую
4 # можно подбрасывать.
5
6 class Coin:
7
8     # Метод __init__ инициализирует
9     # атрибут данных sideup значением 'Орел'.
10
11    def __init__(self):
12        self.sideup = 'Орел'
13
```

```
14  # Метод toss генерирует случайное число
15  # в диапазоне от 0 до 1. Если это число
16  # равно 0, то sideup получает значение 'Орел'.
17  # В противном случае sideup получает значение 'Решка'.
18
19  def toss(self):
20      if random.randint(0, 1) == 0:
21          self.sideup = 'Орел'
22      else:
23          self.sideup = 'Решка'
24
25  # Метод get_sideup возвращает значение,
26  # на которое ссылается sideup.
27
28  def get_sideup(self):
29      return self.sideup
30
31 # Главная функция.
32 def main():
33     # Создать объект на основе класса Coin.
34     my_coin = Coin()
35
36     # Показать обращенную вверх сторону монеты.
37     print('Эта сторона обращена вверх:', my_coin.get_sideup())
38
39     # Подбросить монету.
40     print('Подбрасываю монету...')
41     my_coin.toss()
42
43     # Показать обращенную вверх сторону монеты.
44     print('Эта сторона обращена вверх:', my_coin.get_sideup())
45
46 # Вызвать главную функцию.
47 if __name__ == '__main__':
48     main()
```

Вывод 1 программы

```
Эта сторона обращена вверх: Орел
Подбрасываю монету...
Эта сторона обращена вверх: Решка
```

Вывод 2 программы

```
Эта сторона обращена вверх: Орел
Подбрасываю монету...
Эта сторона обращена вверх: Орел
```

Вывод 3 программы

```
Эта сторона обращена вверх: Орел
Подбрасываю монету...
Эта сторона обращена вверх: Решка
```

Приглядитесь к инструкции в строке 34:

```
my_coin = Coin()
```

Выражение `Coin()`, которое расположено справа от оператора `=`, приводит к тому, что:

- ◆ в оперативной памяти создается объект на основе класса `Coin`;
- ◆ исполняется метод `__init__()` класса `Coin`, и параметру `self` автоматически назначается объект, который был только что создан. В результате атрибуту `sideup` этого объекта присваивается строковый литерал 'Орел'.

На рис. 10.5 проиллюстрированы эти шаги.

После этого оператор `=` присваивает только что созданный объект `Coin` переменной `my_coin`. На рис. 10.6 показано, что после исполнения инструкции в строке 12 переменная `my_coin` будет ссылаться на объект `Coin`, а атрибуту `sideup` этого объекта присвоен строковый литерал 'Орел'.

Следующая исполняемая инструкция находится в строке 37:

```
print('Эта сторона обращена вверх:', my_coin.get_sideup())
```

Эта инструкция печатает сообщение об обращенной вверх стороне монеты. В данной инструкции расположено приведенное ниже выражение:

```
my_coin.get_sideup()
```

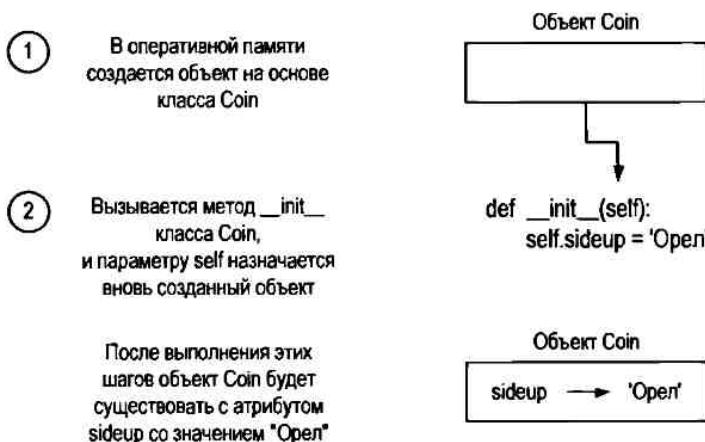


РИС. 10.5. Действия, вызванные выражением `Coin()`

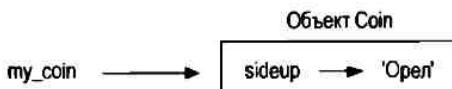


РИС. 10.6. Переменная `my_coin` ссылается на объект `Coin`

Это выражение для вызова метода `get_sideup()` использует объект, на который ссылается `my_coin`. После исполнения метода параметр `self` будет ссылаться на объект `my_coin`. В результате метод возвращает строковое значение 'Орел'.

Несмотря на то что метод `sideup()` имеет параметрическую переменную `self`, нам не нужно передавать в него аргумент. При вызове этого метода Python автоматически передает в первый параметр метода ссылку на вызывающий объект. В результате параметр `self` автоматически будет ссылаться на объект, которым этот метод должен оперировать.

Следующими исполняемыми строками являются строки 40 и 41:

```
print('Подбрасываю монету...')
my_coin.toss()
```

Инструкция в строке 41 использует объект, на который ссылается `my_coin`, для вызова метода `toss()`. После исполнения этого метода параметр `self` будет ссылаться на объект `my_coin`. Метод генерирует случайное число и использует это число для изменения значения атрибута `sideup` данного объекта.

Далее исполняется строка 44. Эта инструкция вызывает метод `my_coin.get_sideup()` для отображения обращенной вверх стороны монеты.

Скрытие атрибутов

Ранее в этой главе мы упомянули, что атрибуты данных объекта должны быть приватными для того, чтобы только методы объекта имели к ним непосредственный доступ. Такой подход защищает атрибуты данных от случайного повреждения. Однако в классе `Coin`, который был показан в предыдущем примере, атрибут `sideup` не является приватным. К нему могут получать непосредственный доступ инструкции, отсутствующие в методе класса `Coin`. Программа 10.3 демонстрирует соответствующий пример. Отметим, что строки 1–30 пропущены для экономии места. Эти строки содержат класс `Coin` и совпадают со строками 1–30 в программе 10.2.

Программа 10.3 (coin_demo2.py)

```
31 # Главная функция.
32 def main():
33     # Создать объект на основе класса Coin.
34     my_coin = Coin()
35
36     # Показать обращенную вверх сторону монеты.
37     print('Эта сторона обращена вверх:', my_coin.get_sideup())
38
39     # Подбросить монету.
40     print('Подбрасываю монету...')
41     my_coin.toss()
42
43     # Но теперь я обману! В этом объекте
44     # я собираюсь напрямую поменять
45     # значение атрибута sideup на 'Орел'.
46     my_coin.sideup = 'Орел'
47
```

```

48     # Показать обращенную вверх сторону монеты.
49     print('Эта сторона обращена вверх:', my_coin.get_sideup())
50
51 # Вызвать главную функцию.
52 if __name__ == '__main__':
53     main()

```

Вывод 1 программы

```

Эта сторона обращена вверх: Орел
Подбрасываю монету...
Эта сторона обращена вверх: Орел

```

Вывод 2 программы

```

Эта сторона обращена вверх: Орел
Подбрасываю монету...
Эта сторона обращена вверх: Орел

```

Вывод 3 программы

```

Эта сторона обращена вверх: Орел
Подбрасываю монету...
Эта сторона обращена вверх: Орел

```

Строка 34 создает в оперативной памяти объект `Coin` и присваивает его переменной `my_coin`. Инструкция в строке 37 показывает обращенную вверх сторону монеты, и затем строка 41 вызывает метод `toss()` объекта. Далее инструкция в строке 46 напрямую присваивает атрибуту `sideup` данного объекта строковый литерал 'Орел':

```
my_coin.sideup = 'Орел'
```

Независимо от исхода метода `toss()`, эта инструкция будет изменять значение атрибута `sideup` объекта `my_coin` на значение 'Орел'. Как видно из трех демонстрационных выполнений программы, монета всегда приземляется орлом вверх!

Если мы хотим сымитировать подбрасывание монеты по-настоящему, надо сделать так, чтобы код за пределами класса не имел возможности менять результат метода `toss()`. А для этого следует сделать атрибут `sideup` приватным. В Python атрибут можно скрыть, если предварить его имя двумя символами подчеркивания. Если изменить имя атрибута `sideup` на `__sideup`, то программный код за пределами класса `Coin` не сможет получать к нему доступ. В программе 10.4 приведена новая версия класса `Coin` с этим внесенным изменением.

Программа 10.4 (coin_demo3.py)

```

1 import random
2
3 # Класс Coin имитирует монету, которую
4 # можно подбрасывать.
5
6 class Coin:
7
8     # Метод __init__ инициализирует
9     # атрибут данных __sideup значением 'Орел'.
10

```

```
11     def __init__(self):
12         self.__sideup = 'Орел'
13
14     # Метод toss генерирует случайное число
15     # в диапазоне от 0 до 1. Если это число
16     # равно 0, то __sideup получает значение 'Орел'.
17     # В противном случае sideup получает значение 'Решка'.
18
19     def toss(self):
20         if random.randint(0, 1) == 0:
21             self.__sideup = 'Орел'
22         else:
23             self.__sideup = 'Решка'
24
25     # Метод get_sideup возвращает значение,
26     # на которое ссылается sideup.
27
28     def get_sideup(self):
29         return self.__sideup
30
31 # Главная функция.
32 def main():
33     # Создать объект на основе класса Coin.
34     my_coin = Coin()
35
36     # Показать обращенную вверх сторону монеты.
37     print('Эта сторона обращена вверх:', my_coin.get_sideup())
38
39     # Подбросить монету.
40     print('Собираюсь подбросить монету десять раз:')
41     for count in range(10):
42         my_coin.toss()
43         print(my_coin.get_sideup())
44
45 # Вызвать главную функцию.
46 if __name__ == '__main__':
47     main()
```

Вывод программы

```
Эта сторона обращена вверх: Орел
Собираюсь подбросить монету десять раз:
Орел
Орел
Решка
Орел
Орел
Орел
Орел
```

Решка
Решка
Решка
Решка

Хранение классов в модулях

Программы, которые вы до сих пор встречали в этой главе, содержат определение класса Coin в том же файле, что и программные инструкции, которые используют класс Coin. Этот подход хорошо работает с небольшими программами, содержащими всего один или два класса. Однако по мере привлечения программами все большего количества классов возрас-тает потребность в упорядочении этих классов.

Программисты обычно упорядочивают свои определения классов, размещая их в модулях. Затем модули можно импортировать в любые программы, которым требуется использовать содержащиеся в них классы. Например, предположим, что мы решаем сохранить класс Coin в модуле coin. В программе 10.5 приведено содержимое файла coin.py. Затем, когда нам нужно применить класс Coin в программе, мы импортируем модуль coin. Это продемонстрировано в программе 10.6.

Программа 10.5 (coin.py)

```
1 import random
2
3 # Класс Coin имитирует монету, которую
4 # можно подбрасывать (теперь это модуль, который хранится в файле).
5
6 class Coin:
7
8     # Метод __init__ инициализирует
9     # атрибут данных __sideup значением 'Орел'.
10
11    def __init__(self):
12        self.__sideup = 'Орел'
13
14    # Метод toss генерирует случайное число
15    # в диапазоне от 0 до 1. Если это число
16    # равно 0, то __sideup получает значение 'Орел'.
17    # В противном случае __sideup получает значение 'Решка'.
18
19    def toss(self):
20        if random.randint(0, 1) == 0:
21            self.__sideup = 'Орел'
22        else:
23            self.__sideup = 'Решка'
24
25    # Метод get_sideup возвращает значение,
26    # на которое ссылается __sideup.
```

```
28     def get_sideup(self):
29         return self._sideup
```

Программа 10.6 (coin_demo4.py)

```
1 # Эта программа импортирует модуль coin
2 # и создает экземпляр класса Coin.
3
4 import coin
5
6 def main():
7     # Создать объект на основе класса Coin.
8     my_coin = coin.Coin()
9
10    # Показать обращенную вверх сторону монеты.
11    print('Эта сторона обращена вверх:', my_coin.get_sideup())
12
13    # Подбросить монету.
14    print('Собираюсь подбросить монету десять раз:')
15    for count in range(10):
16        my_coin.toss()
17        print(my_coin.get_sideup())
18
19 # Вызвать главную функцию.
20 if __name__ == '__main__':
21     main()
```

Вывод программы

```
Эта сторона обращена вверх: Орел
Собираюсь подбросить монету десять раз:
Решка
Решка
Орел
Орел
Решка
Орел
Решка
Решка
Орел
Орел
```

Строка 4 импортирует модуль coin. Обратите внимание, что в строке 8 нам пришлось квалифицировать имя класса Coin, добавив в качестве префикса имя его модуля с точкой:

```
my_coin = coin.Coin()
```

Класс *BankAccount*

Давайте рассмотрим еще один пример. В программе 10.7 представлен класс *BankAccount* (Банковский счет), сохраненный в модуле *bankaccount*. Объекты, которые создаются на основе этого класса, имитируют банковские счета, позволяя иметь начальный остаток, вносить вклады, снимать суммы со счета и получать текущий остаток на счете.

Программа 10.7 (bankaccount.py)

```
1 # Класс BankAccount имитирует банковский счет.
2
3 class BankAccount:
4
5     # Метод __init__ принимает аргумент
6     # с остатком на счете.
7     # Он присваивается атрибуту __balance.
8
9     def __init__(self, bal):
10         self.__balance = bal
11
12     # Метод deposit вносит
13     # на счет вклад.
14
15     def deposit(self, amount):
16         self.__balance += amount
17
18     # Метод withdraw снимает сумму
19     # со счета.
20
21     def withdraw(self, amount):
22         if self.__balance >= amount:
23             self.__balance -= amount
24         else:
25             print('Ошибка: недостаточно средств')
26
27     # Метод get_balance возвращает
28     # остаток средств на счете.
29
30     def get_balance(self):
31         return self.__balance
```

Обратите внимание, что метод *__init__()* имеет две параметрические переменные: *self* и *bal*. Параметр *bal* в качестве аргумента принимает начальный остаток на расчетном счете. В строке 10 сумма параметра *bal* присваивается атрибуту *__balance* объекта.

Метод *deposit()* расположен в строках 15–16. Он имеет две параметрические переменные: *self* и *amount*. При вызове этого метода вносимая на счет сумма передается в параметр *amount*, значение которого затем прибавляется к атрибуту *__balance* в строке 16.

Метод `withdraw()` расположен в строках 21–25. Он имеет две параметрические переменные: `self` и `amount`. При вызове этого метода снимаемая с банковского счета сумма передается в параметр `amount`. Инструкция `if`, которая начинается в строке 22, определяет, достаточна ли величина остатка для снятия средств с банковского счета. Если да, то сумма `amount` вычитается из остатка в строке 23. В противном случае строка 25 выводит сообщение 'Ошибка: недостаточно средств'.

Метод `get_balance()` расположен в строках 30–31. Он возвращает значение атрибута `_balance`.

Программа 10.8 демонстрирует применение этого класса.

Программа 10.8 (account_test.py)

```

1 # Эта программа демонстрирует класс BankAccount.
2
3 import bankaccount
4
5 def main():
6     # Получить начальный остаток.
7     start_bal = float(input('Введите свой начальный остаток: '))
8
9     # Создать объект BankAccount.
10    savings = bankaccount.BankAccount(start_bal)
11
12    # Внести на счет зарплату пользователя.
13    pay = float(input('Сколько Вы получили на этой неделе? '))
14    print('Вношу эту сумму на Ваш счет.')
15    savings.deposit(pay)
16
17    # Показать остаток.
18    print(f'Ваш остаток на счете составляет ${savings.get_balance():,.2f}.')
19
20    # Получить сумму для снятия с банковского счета.
21    cash = float(input('Какую сумму Вы желаете снять со счета? '))
22    print('Снимаю эту сумму с Вашего банковского счета.')
23    savings.withdraw(cash)
24
25    # Показать остаток.
26    print(f'Ваш остаток на счете составляет ${savings.get_balance():,.2f}.')
27
28 # Вызвать главную функцию.
29 if __name__ == '__main__':
30     main()

```

Вывод 1 программы (вводимые данные выделены жирным шрифтом)

Введите свой начальный остаток: 1000

Сколько Вы получили на этой неделе? 500

Вношу эту сумму на Ваш счет.

Ваш остаток на счете составляет \$1500.00

Какую сумму Вы желаете снять со счета? **1200**

Снимаю эту сумму с Вашего банковского счета.

Ваш остаток на счете составляет \$300.00

Вывод 2 программы (вводимые данные выделены жирным шрифтом)

Введите свой начальный остаток: **1000**

Сколько Вы получили на этой неделе? **500**

Вношу эту сумму на Ваш счет.

Ваш остаток на счете составляет \$1500.00

Какую сумму Вы желаете снять со счета? **2000**

Снимаю эту сумму с Вашего банковского счета.

Ошибка: недостаточно средств

Ваш остаток на счете составляет \$1500.00

Строка 7 получает от пользователя начальный остаток счета и присваивает его переменной `start_bal`. Стока 10 создает экземпляр класса `BankAccount` и присваивает его переменной `savings` (сбережения):

```
savings = bankaccount.BankAccount(start_bal)
```

Обратите внимание, что переменная `start_bal` помещена в круглые скобки. В результате переменная `start_bal` передается в качестве аргумента в метод `__init__()`. В методе `__init__()` она будет передана в параметр `bal`.

Строка 13 получает сумму заработной платы пользователя и присваивает ее переменной `pay`. В строке 15 вызывается метод `savings.deposit()` с переменной `pay`, передаваемой ему в качестве аргумента. В методе `deposit()` она будет передана в параметр `amount`.

Инструкция в строке 18 показывает остаток на банковском счете. Обратите внимание, что мы используем f-строку для вызова метода `savings.get_balance()`. Значение, возвращаемое указанным методом, форматируется в виде суммы в долларах.

Строка 21 получает сумму, которую пользователь хочет снять, и закрепляет ее за переменной `cash`. В строке 23 вызывается метод `savings.withdraw()`, получающий переменную `cash` в качестве аргумента. В методе `withdraw()` она будет передана в параметр `amount`. Инструкция в строке 26 выводит на экран окончательный остаток на счете.

Метод `__str__`

Довольно часто возникает необходимость вывести сообщение, которое показывает состояние объекта. *Состояние* объекта — это просто значения атрибутов объекта в тот или иной конкретный момент. Например, вспомните, что класс `BankAccount` имеет один атрибут данных: `_balance`. В любой конкретный момент атрибут `_balance` объекта `BankAccount` будет ссылаться на какое-то значение. Значение атрибута `_balance` представляет состояние объекта в этот момент. Приведенный ниже фрагмент кода служит примером вывода состояния объекта `BankAccount`:

```
account = bankaccount.BankAccount(1500.0)
print(f'Остаток составляет ${savings.get_balance():,.2f}')
```

Первая инструкция создает объект `BankAccount`, передавая в метод `__init__()` значение `1500.0`. После исполнения этой инструкции переменная `account` будет ссылаться на объект `BankAccount`. Вторая строка выводит форматированное строковое значение, показывающее значение атрибута `_balance` данного объекта. Результат работы этой инструкции будет выглядеть так:

Остаток составляет \$1500.00

Вывод на экран состояния объекта — широко распространенная задача. Причем настолько, что многие программисты оснащают свои классы методом, который возвращает строковое значение, содержащее состояние объекта. В Python этому методу присвоено специальное имя `__str__`. В программе 10.9 представлен класс `BankAccount` с добавленным в него методом `__str__()`, который расположен в строках 36–37. Он возвращает строковое значение, сообщающее остаток на банковском счете.

Программа 10.9 (bankaccount2.py)

```
1 # Класс BankAccount имитирует банковский счет.
2
3 class BankAccount:
4
5     # Метод __init__ принимает аргумент
6     # с остатком на счете.
7     # Он присваивается атрибуту __balance.
8
9     def __init__(self, bal):
10         self.__balance = bal
11
12     # Метод deposit вносит
13     # на счет вклад.
14
15     def deposit(self, amount):
16         self.__balance += amount
17
18     # Метод withdraw снимает сумму
19     # со счета.
20
21     def withdraw(self, amount):
22         if self.__balance >= amount:
23             self.__balance -= amount
24         else:
25             print('Ошибка: недостаточно средств')
26
27     # Метод get_balance возвращает
28     # остаток средств на счете.
29
30     def get_balance(self):
31         return self.__balance
32
```

```

33     # Метод __str__ возвращает строковое
34     # значение, сообщающее о состоянии объекта.
35
36     def __str__(self):
37         return f'Остаток составляет ${self.__balance:.2f}'

```

Метод `__str__()` вызывается не напрямую, а автоматически во время передачи объекта в качестве аргумента в функцию `print`. В программе 10.10 приведен соответствующий пример.

Программа 10.10 (account_test2.py)

```

1 # Эта программа демонстрирует класс BankAccount
2 # с добавленным в него методом __str__.
3
4 import bankaccount2
5
6 def main():
7     # Получить начальный остаток.
8     start_bal = float(input('Введите свой начальный остаток: '))
9
10    # Создать объект BankAccount.
11    savings = bankaccount2.BankAccount(start_bal)
12
13    # Внести на счет зарплату пользователя.
14    pay = float(input('Сколько Вы получили на этой неделе? '))
15    print('Вношу эту сумму на Ваш счет.')
16    savings.deposit(pay)
17
18    # Показать остаток.
19    print(savings)
20
21    # Получить сумму для снятия с банковского счета.
22    cash = float(input('Какую сумму Вы желаете снять со счета? '))
23    print('Снимаю эту сумму с Вашего банковского счета.')
24    savings.withdraw(cash)
25
26    # Показать остаток.
27    print(savings)
28
29 # Вызвать главную функцию.
30 if __name__ == '__main__':
31     main()

```

Вывод программы (вводимые данные выделены жирным шрифтом)

Введите свой начальный остаток: 1000

Сколько Вы получили на этой неделе? 500

```

Вношу эту сумму на Ваш счет.
Остаток составляет $1,500.00
Какую сумму Вы желаете снять со счета? 1200 
Снимаю эту сумму с Вашего банковского счета.
Остаток составляет $300.00

```

Имя объекта, `savings`, передается в функцию `print` в строках 19 и 27. В результате вызывается метод `__str__()` класса `BankAccount`. Затем выводится строковое значение, которое возвращается из метода `__str__()`.

Метод `__str__()` также вызывается автоматически, когда объект передается в качестве аргумента во встроенную функцию `str`. Вот пример:

```

account = bankaccount2.BankAccount(1500.0)
message = str(account)
print(message)

```

Во второй инструкции объект `account` передается в качестве аргумента в функцию `str`. В результате вызывается метод `__str__()` класса `BankAccount`. Возвращаемое строковое значение присваивается переменной `message` и затем в третьей строке выводится функцией `print`.



Контрольная точка

- 10.5. Вы слышите, что кто-то высказывает следующий комментарий: "Проект — это дизайн дома. Плотник использует проект для возведения дома. Если плотник пожелает, он может построить несколько идентичных домов на основе одного и того же проекта". Представьте это как метафору для классов и объектов. Этот проект представляет класс или же он представляет объект?
- 10.6. В данной главе с целью описания классов и объектов мы используем метафору с формой и печеньем, которые изготавливаются при помощи формы. В этой метафоре объекты представлены формой или же печеньем?
- 10.7. Какова задача метода `__init__()`? И когда он исполняется?
- 10.8. Какова задача параметра `self` в методе?
- 10.9. Каким образом в классе Python атрибут скрывается от программного кода, находящегося за пределами класса?
- 10.10. Какова задача метода `__str__()`?
- 10.11. Каким образом происходит вызов метода `__str__()`?

10.3

Работа с экземплярами

Ключевые положения

Каждый экземпляр класса имеет собственный набор атрибутов данных.

При использовании методом параметра `self` для создания атрибута этот атрибут принадлежит конкретному объекту, на который ссылается параметр `self`. Мы называем такие атрибуты *атрибутами экземпляра*, потому что они принадлежат конкретному экземпляру класса.

В программе можно создавать многочисленные экземпляры одного и того же класса. И каждый экземпляр будет иметь собственный набор атрибутов. Например, взгляните на программу 10.11. В ней создаются три экземпляра класса Coin. Каждый экземпляр имеет собственный атрибут `_sideup`.

Программа 10.11 (coin_demo5.py)

```
1 # Эта программа импортирует имитационный модуль
2 # и создает три экземпляра класса Coin.
3
4 import coin
5
6 def main():
7     # Создать три объекта класса Coin.
8     coin1 = coin.Coin()
9     coin2 = coin.Coin()
10    coin3 = coin.Coin()
11
12    # Показать повернутую вверх сторону каждой монеты.
13    print('Вот три монеты, у которых эти стороны обращены вверх:')
14    print(coin1.get_sideup())
15    print(coin2.get_sideup())
16    print(coin3.get_sideup())
17    print()
18
19    # Подбросить монету.
20    print('Подбрасываю все три монеты...')
21    print()
22    coin1.toss()
23    coin2.toss()
24    coin3.toss()
25
26    # Показать повернутую вверх сторону каждой монеты.
27    print('Теперь обращены вверх вот эти стороны:')
28    print(coin1.get_sideup())
29    print(coin2.get_sideup())
30    print(coin3.get_sideup())
31    print()
32
33 # Вызвать главную функцию.
34 if __name__ == '__main__':
35     main()
```

Вывод программы

Вот три монеты, у которых эти стороны обращены вверх:

Орел

Орел

Орел

Подбрасываю все три монеты...

Теперь обращены вверх вот эти стороны:

Решка

Решка

Орел

В строках 8–10 приведенные ниже инструкции создают три объекта, каждый из которых является экземпляром класса Coin:

```
coin1 = coin.Coin()
coin2 = coin.Coin()
coin3 = coin.Coin()
```

На рис. 10.7 представлено, каким образом переменные coin1, coin2 и coin3 ссылаются на три объекта после исполнения этих инструкций. Обратите внимание, что каждый объект имеет свой атрибут `_sideup`. Строки 14–16 показывают значения, возвращаемые из метода `get_sideup()` каждого объекта.

Затем инструкции в строках 22–24 вызывают метод `toss()` каждого объекта:

```
coin1.toss()
coin2.toss()
coin3.toss()
```

На рис. 10.8 показано, каким образом эти инструкции изменили атрибут `_sideup` каждого объекта в демонстрационном выполнении программы.

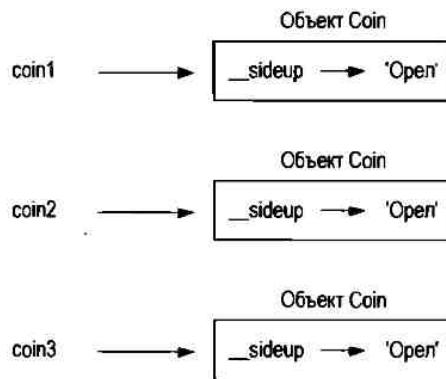


РИС. 10.7. Переменные coin1, coin2 и coin3 ссылаются на три объекта

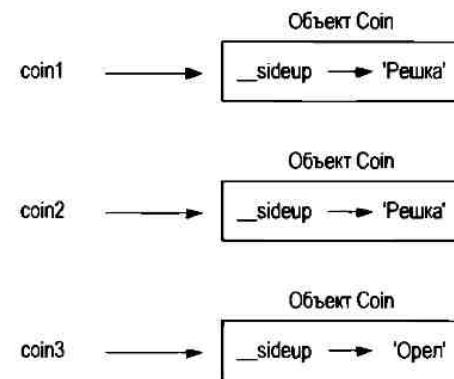


РИС. 10.8. Объекты после метода `toss()`

В ЦЕНТРЕ ВНИМАНИЯ

Создание класса `CellPhone`

Компания "Беспроводные решения" продает сотовые телефоны и услуги беспроводной связи. Вы работаете программистом в IT-отделе компании, и ваша команда разрабатывает программу для учета сведений обо всех сотовых телефонах, которые находятся на складе.



Вам поручили разработать класс, который представляет сотовый телефон. Вот данные, которые должны храниться в качестве атрибутов класса:

- ◆ имя производителя телефона будет присвоено атрибуту `_manufact`;
- ◆ номер модели телефона будет присвоен атрибуту `_model`;
- ◆ розничная цена телефона будет присвоена атрибуту `_retail_price`.

Кроме того, этот класс будет иметь приведенные ниже методы:

- ◆ метод `__init__()` принимает аргументы для производителя, номера модели и розничной цены;
- ◆ метод `set_manufact()` принимает аргумент для производителя и в случае необходимости позволит изменять значение атрибута `_manufact` после создания объекта;
- ◆ метод `set_model()` принимает аргумент для модели и в случае необходимости позволит изменять значение атрибута `_model` после создания объекта;
- ◆ метод `set_retail_price()` принимает аргумент для розничной цены и в случае необходимости позволяет изменять значение атрибута `retail_price` после создания объекта;
- ◆ метод `get_manufact()` возвращает название производителя телефона;
- ◆ метод `get_model()` возвращает номер модели телефона;
- ◆ метод `get_retail_price()` возвращает розничную цену телефона.

В программе 10.12 показано определение класса, который сохранен в модуле `cellphone`.

Программа 10.12 (cellphone.py)

```
1 # Класс CellPhone содержит данные о сотовом телефоне.
2
3 class CellPhone:
4
5     # Метод __init__ инициализирует атрибуты.
6
7     def __init__(self, manufact, model, price):
8         self.__manufact = manufact
9         self.__model = model
10        self.__retail_price = price
11
12    # Метод set_manufact принимает аргумент для
13    # производителя телефона.
14
15    def set_manufact(self, manufact):
16        self.__manufact = manufact
17
18    # Метод set_model принимает аргумент для
19    # номера модели телефона.
20
21    def set_model(self, model):
22        self.__model = model
23
```

```
24     # Метод set_retail_price принимает аргумент для
25     # розничной цены телефона.
26
27     def set_retail_price(self, price):
28         self.__retail_price = price
29
30     # Метод get_manufact возвращает
31     # производителя телефона.
32
33     def get_manufact(self):
34         return self.__manufact
35
36     # Метод get_model возвращает
37     # номер модели телефона.
38
39     def get_model(self):
40         return self.__model
41
42     # Метод get_retail_price возвращает
43     # розничную цену телефона.
44
45     def get_retail_price(self):
46         return self.__retail_price
```

Класс CellPhone будет импортирован в несколько программ, которые разрабатывает ваша команда. Для того чтобы протестировать класс, вы пишете программный код, приведенный в программе 10.13. Это простая программа предлагает пользователю ввести название производителя, номер модели и розничную цену телефона, создает экземпляр класса CellPhone и присваивает эти данные его атрибутам.

Программа 10.13 (cell_phone_test.py)

```
1 # Эта программа тестирует класс CellPhone.
2
3 import cellphone
4
5 def main():
6     # Получить данные о телефоне.
7     man = input('Введите производителя: ')
8     mod = input('Введите номер модели: ')
9     retail = float(input('Введите розничную цену: '))
10
11     # Создать экземпляр класса CellPhone.
12     phone = cellphone.CellPhone(man, mod, retail)
13
```

```

14 # Показать введенные данные.
15 print('Вот введенные Вами данные:')
16 print(f'Производитель: {phone.get_manufact()}')
17 print(f'Номер модели: {phone.get_model()}')
18 print(f'Розничная цена: ${phone.get_retail_price():,.2f}')
19
20 # Вызвать главную функцию.
21 if __name__ == '__main__':
22     main()

```

Вывод программы (вводимые данные выделены жирным шрифтом)

```

Ведите производителя: Acme Electronics 
Ведите номер модели: M1000 
Ведите розничную цену: 199.99 
Вот введенные Вами данные:
Производитель: Acme Electronics
Номер модели: M1000
Розничная цена: $199.99

```

Методы-получатели и методы-мутаторы

Как отмечалось ранее, на практике широко принято делать в классе все атрибуты данных приватными и предоставлять публичные методы для доступа к этим атрибутам и их изменения. Так гарантируется, что объект, владеющий этими атрибутами, будет держать под контролем все вносимые в них изменения.

Метод, который возвращает значение из атрибута класса и при этом его не изменяет, называется *методом-получателем*. Методы-получатели дают возможность программному коду, находящемуся за пределами класса, извлекать значения атрибутов безопасным образом, не подвергая эти атрибуты изменению программным кодом, находящимся вне метода. В классе *CellPhone*, который вы увидели в программе 10.12, методы *get_manufact()*, *get_model()* и *get_retail_price()* являются методами-получателями.

Метод, который сохраняет значение в атрибуте данных либо каким-нибудь иным образом изменяет значение атрибута данных, называется *методом-мутатором*. Методы-мутаторы могут управлять тем, как атрибуты данных класса изменяются. Когда программный код, находящийся вне класса, должен изменить в объекте значение атрибута данных, он, как правило, вызывает мутатор и передает новое значение в качестве аргумента. Если это необходимо, то мутатор, прежде чем он присвоит значение атрибуту данных, может выполнить проверку этого значения. В программе 10.12 методы *set_manufact()*, *set_model()* и *set_retail_price()* являются методами-мутаторами.

ПРИМЕЧАНИЕ

Методы-мутаторы иногда называют *сеттерами* (setter), или методами-установщиками, а методы-получатели — *геттерами* (getter).

В ЦЕНТРЕ ВНИМАНИЯ

Хранение объектов в списке



Класс CellPhone, который вы создали в предыдущей рубрике *"В центре внимания"*, будет использоваться во множестве программ. Многие из этих программ будут хранить объекты CellPhone в списках. Для того чтобы протестировать способность хранить объекты CellPhone в списке, вы пишете программный код в программе 10.14. Она получает от пользователя данные о пяти телефонах, создает пять объектов CellPhone, содержащих эти данные, и сохраняет эти объекты в списке. Затем она выполняет последовательный обход списка, показывая атрибуты каждого объекта.

Программа 10.14 (cell_phone_list.py)

```
1 # Эта программа создает пять объектов CellPhone
2 # и сохраняет их в списке.
3
4 import cellphone
5
6 def main():
7     # Получить список объектов CellPhone.
8     phones = make_list()
9
10    # Показать данные в списке.
11    print('Вот введенные Вами данные:')
12    display_list(phones)
13
14 # Функция make_list получает от пользователя данные
15 # о пяти телефонах, а затем возвращает список
16 # объектов CellPhone, содержащих эти данные.
17
18 def make_list():
19     # Создать пустой список.
20     phone_list = []
21
22     # Добавить пять объектов CellPhone в список.
23     print('Введите данные о пяти телефонах.')
24     for count in range(1, 6):
25         # Получить данные о телефоне.
26         print('Номер телефона ' + str(count) + ':')
27         man = input('Введите производителя: ')
28         mod = input('Введите номер модели: ')
29         retail = float(input('Введите розничную цену: '))
30         print()
31
32     # Создать новый объект CellPhone в памяти
33     # и присвоить его переменной phone.
34     phone = cellphone.CellPhone(man, mod, retail)
```

```
35
36     # Добавить объект в список.
37     phone_list.append(phone)
38
39     # Вернуть список.
40     return phone_list
41
42 # Функция display_list принимает список с объектами
43 # CellPhone в качестве аргумента и показывает
44 # хранящиеся в каждом объекте данные.
45
46 def display_list(phone_list):
47     for item in phone_list:
48         print(item.get_manufact())
49         print(item.get_model())
50         print(item.get_retail_price())
51         print()
52
53 # Вызвать главную функцию.
54 if __name__ == '__main__':
55     main()
```

Вывод программы (вводимые данные выделены жирным шрифтом)

Введите данные о пяти телефонах.

Номер телефона 1:

Введите производителя: **Actme Electronics**

Введите номер модели: **M1000**

Введите розничную цену: **199.99**

Номер телефона 2:

Введите производителя: **Atlantic Communications**

Введите номер модели: **S2**

Введите розничную цену: **149.99**

Номер телефона 3:

Введите производителя: **Wavelength Electronics**

Введите номер модели: **N477**

Введите розничную цену: **249.99**

Номер телефона 4:

Введите производителя: **Edison Wireless**

Введите номер модели: **SLX88**

Введите розничную цену: **169.99**

Номер телефона 5:

Введите производителя: **Sonic Systems**

```
Введите номер модели: X99 [Enter]
Введите розничную цену: 299.99 [Enter]
```

Вот введенные Вами данные:

Acme Electronics

M1000

199.99

Atlantic Communications

S2

149.99

Wavelength Electronics

N477

249.99

Edison Wireless

SLX88

169.99

Sonic Systems

X99

299.99

Функция `make_list` расположена в строках 18–40. В строке 20 создается пустой список `phone_list`. Цикл `for`, который начинается в строке 24, выполняет пять итераций. Во время каждой итерации цикл получает от пользователя данные о сотовом телефоне (строки 27–29), создает экземпляр класса `CellPhone`, который инициализируется данными (строка 34), и добавляет этот объект в список `phone_list` (строка 37). Стока 40 возвращает список.

Функция `display_list` в строках 46–51 принимает список объектов `CellPhone` в качестве аргумента. Цикл `for`, который начинается в строке 47, выполняет перебор объектов в списке и показывает значения атрибутов каждого объекта.

Передача объектов в качестве аргументов

Когда вы разрабатываете приложения, которые работают с объектами, часто возникает необходимость написать функции и методы, принимающие объекты в качестве аргументов. Например, приведенный ниже фрагмент кода демонстрирует функцию `show_coin_status` (показать состояние монеты), которая принимает объект `Coin` в качестве аргумента:

```
def show_coin_status(coin_obj):
    print('Эта сторона обращена вверх:', coin_obj.get_sideup())
```

Приведенный ниже пример программного кода показывает, каким образом можно создать объект `Coin` и затем передать его в качестве аргумента в функцию `show_coin_status`:

```
my_coin = coin.Coin()
show_coin_status(my_coin)
```

Во время передачи объекта в качестве аргумента в параметрическую переменную передается ссылка на объект. В результате функция или метод, который получает объект в качестве аргумента, имеют доступ к фактическому объекту. Например, взгляните на приведенный ниже метод `flip()` (подбросить):

```
def flip(coin_obj):  
    coin_obj.toss()
```

Этот метод принимает объект `Coin` в качестве аргумента и вызывает метод `toss()` этого объекта. Программа 10.15 демонстрирует этот метод.

Программа 10.15 (coin_argument.py)

```
1 # Эта программа передает объект Coin  
2 # в качестве аргумента в функцию.  
3 import coin  
4  
5 # Главная функция  
6 def main():  
7     # Создать объект Coin.  
8     my_coin = coin.Coin()  
9  
10    # Эта инструкция покажет 'Орел'.  
11    print(my_coin.get_sideup())  
12  
13    # Передать объект в функцию flip.  
14    flip(my_coin)  
15  
16    # Эта инструкция может показать 'Орел'  
17    # либо 'Решка'.  
18    print(my_coin.get_sideup())  
19  
20 # Функция flip подбрасывает монету.  
21 def flip(coin_obj):  
22     coin_obj.toss()  
23  
24 # Вызвать главную функцию.  
25 if __name__ == '__main__':  
26     main()
```

Вывод 1 программы

```
Орел  
Решка
```

Вывод 2 программы

```
Орел  
Орел
```

Вывод 3 программы

```
Орел  
Решка
```

Инструкция в строке 8 создает объект `Coin`, на который ссылается переменная `my_coin`. Стока 11 показывает значение атрибута `_sideup` объекта `my_coin`. Поскольку метод `__init__()` объекта назначил атрибуту `_sideup` значение 'Орел', мы знаем, что строка 11 покажет строковое значение 'Орел'. Стока 14 вызывает функцию `flip`, передавая объект `my_coin` в качестве аргумента. Внутри функции `flip` вызывается метод `toss()` объекта `my_coin`. Затем строка 18 снова показывает значение атрибута `_sideup` объекта `my_coin`. На этот раз мы не можем предсказать, будет ли показано значение 'Орел' или 'Решка', потому что был вызван метод `toss()` объекта `my_coin`.

В ЦЕНТРЕ ВНИМАНИЯ



Консервация собственных объектов

Из главы 9 известно, что модуль `pickle` предоставляет функции для сериализации объектов. Сериализация объекта означает его преобразование в поток байтов, которые могут быть сохранены в файле для последующего извлечения. Функция `dump` модуля `pickle` сериализует (консервирует) объект и записывает его в файл, а функция `load` извлекает объект из файла и его десериализует (расконсервирует).

В главе 9 вы встречали примеры, в которых консервировались и расконсервировались объекты-словари. Консервировать и расконсервировать можно также объекты собственных классов. В программе 10.16 представлен пример, который консервирует три объекта `CellPhone` и сохраняет их в файле. Программа 10.17 извлекает эти объекты из файла и расконсервирует их.

Программа 10.16 (pickle_cellphone.py)

```

1 # Эта программа консервирует объекты CellPhone.
2 import pickle
3 import cellphone
4
5 # Константа для имени файла.
6 FILENAME = 'cellphones.dat'
7
8 def main():
9     # Инициализировать переменную для управления циклом.
10    again = 'д'
11
12    # Открыть файл.
13    output_file = open(FILENAME, 'wb')
14
15    # Получить данные от пользователя.
16    while again.lower() == 'д':
17        # Получить данные о сотовом телефоне.
18        man = input('Введите производителя: ')
19        mod = input('Введите номер модели: ')
20        retail = float(input('Введите розничную цену: '))
21

```

```

22     # Создать объект CellPhone.
23     phone = cellphone.CellPhone(man, mod, retail)
24
25     # Законсервировать объект и записать его в файл.
26     pickle.dump(phone, output_file)
27
28     # Получить еще один элемент данных?
29     again = input('Введите еще один элемент данных? (д/н) : ')
30
31     # Закрыть файл.
32     output_file.close()
33     print(f'Данные записаны в {FILENAME}')
34
35 # Вызвать главную функцию.
36 if __name__ == '__main__':
37     main()

```

Вывод программы (вводимые данные выделены жирным шрифтом)

```

Ведите производителя: ACME Electronics 
Ведите номер модели: M1000 
Ведите розничную цену: 199.99 
Ведете еще один элемент данных? (д/н): д 
Ведите производителя: Sonic Systems 
Ведите номер модели: X99 
Ведите розничную цену: 299.99 
Ведете еще один элемент данных? (д/н): и 
Данные записаны в cellphones.dat

```

Программа 10.17 (unpickle_cellphone.py)

```

1 # Эта программа расконсервирует объекты CellPhone.
2 import pickle
3 import cellphone
4
5 # Константа для имени файла.
6 FILENAME = 'cellphones.dat'
7
8 def main():
9     end_of_file = False # Для обозначения конца файла
10
11    # Открыть файл.
12    input_file = open(FILENAME, 'rb')
13
14    # Прочитать до конца файла.
15    while not end_of_file:
16        try:
17            # Расконсервировать следующий объект.
18            phone = pickle.load(input_file)

```

```

19
20         # Показать данные о сотовом телефоне.
21         display_data(phone)
22     except EOFError:
23         # Установить флаг, чтобы обозначить, что
24         # был достигнут конец файла.
25         end_of_file = True
26
27     # Закрыть файл.
28     input_file.close()
29
30 # Функция display_data показывает данные
31 # из объекта CellPhone, переданного в качестве аргумента.
32 def display_data(phone):
33     print(f'Производитель: {phone.get_manufact()}')
34     print(f'Номер модели: {phone.get_model()}')
35     print(f'Розничная цена: ${phone.get_retail_price():,.2f}')
36     print()
37
38 # Вызвать главную функцию.
39 if __name__ == '__main__':
40     main()

```

Вывод программы

```

Производитель: ACME Electronics
Номер модели: M1000
Розничная цена: $199.99
Производитель: Sonic Systems
Номер модели: X99
Розничная цена: $299.99

```

В ЦЕНТРЕ ВНИМАНИЯ

Хранение объектов в словаре

Из главы 9 известно, что словари — это объекты, которые хранят элементы в качестве пар "ключ : значение". Каждый элемент в словаре имеет ключ и значение. Если нужно получить из словаря конкретное значение, то это можно сделать, указав ключ. В главе 9 вы встречали примеры, которые сохраняли значения в словарях, в частности строковые значения, целые числа, числа с плавающей точкой, списки и кортежи. Словари также широко используются для хранения объектов, которые вы создаете на основе собственных классов.

Давайте рассмотрим пример. Предположим, что вы хотите создать программу, которая поддерживает контактную информацию, скажем, имена, телефонные номера и электронные адреса. Начать эту работу можно с написания класса, в частности класса `Contact`, приведенного в программе 10.18. Экземпляр класса `Contact` хранит следующие ниже данные:



- ◆ имя человека — в атрибуте `_name`;
- ◆ телефонный номер — в атрибуте `_phone`;
- ◆ электронный адрес — в атрибуте `_email`.

Этот класс имеет следующие методы:

- ◆ метод `__init__()` принимает аргументы для имени, телефонного номера и электронного адреса человека;
- ◆ метод `set_name()` устанавливает атрибут `_name`;
- ◆ метод `set_phone()` устанавливает атрибут `_phone`;
- ◆ метод `set_email()` устанавливает атрибут `_email`;
- ◆ метод `get_name()` возвращает атрибут `_name`;
- ◆ метод `get_phone()` возвращает атрибут `_phone`;
- ◆ метод `get_email()` возвращает атрибут `_email`;
- ◆ метод `__str__()` возвращает состояние объекта в виде строкового значения.

Программа 10.18 (contact.py)

```
1 # Класс Contact содержит контактную информацию.
2
3 class Contact:
4     # Метод __init__ инициализирует атрибуты.
5     def __init__(self, name, phone, email):
6         self.__name = name
7         self.__phone = phone
8         self.__email = email
9
10    # Метод set_name устанавливает атрибут name.
11    def set_name(self, name):
12        self.__name = name
13
14    # Метод set_phone устанавливает атрибут phone.
15    def set_phone(self, phone):
16        self.__phone = phone
17
18    # Метод set_email устанавливает атрибут email.
19    def set_email(self, email):
20        self.__email = email
21
22    # Метод get_name возвращает атрибут name.
23    def get_name(self):
24        return self.__name
25
26    # Метод get_phone возвращает атрибут phone.
27    def get_phone(self):
28        return self.__phone
29
```

```

30     # Метод get_email возвращает атрибут email.
31     def get_email(self):
32         return self._email
33
34     # Метод __str__ возвращает состояние объекта
35     # в виде строкового значения.
36     def __str__(self):
37         return f'Имя: {self._name}\n' + \
38             f'Телефон: {self._phone}\n' + \
39             f'Электронная почта: {self._email}'

```

Далее вы можете написать программу, которая сохраняет объекты Contact в словаре. Всякий раз, когда программа создает объект Contact с данными о конкретном человеке, этот объект будет сохраняться в качестве значения в словаре, используя имя этого человека как ключ. Затем в любое время, когда вам потребуется получить данные конкретного человека, вы воспользуетесь его именем в качестве ключа для извлечения объекта Contact из словаря.

Программа 10.19 демонстрирует такой пример. Она выводит меню, которое позволяет пользователю выполнять любую из приведенных ниже операций:

- ◆ найти контактное лицо в словаре;
- ◆ добавить новое контактное лицо в словарь;
- ◆ изменить существующее контактное лицо в словаре;
- ◆ удалить контактное лицо из словаря;
- ◆ выйти из программы.

Кроме того, когда пользователь выходит из программы, она автоматически консервирует словарь и сохраняет его в файле. Во время запуска программы она автоматически извлекает и расконсервирует словарь из файла. (Из главы 10 известно, что в процессе консервации объекта он сохраняется в файле, а в процессе расконсервации объекта он извлекается из файла.) Если файл не существует, то программа начинает работу с пустого словаря.

Программа разделена на восемь функций: `main` (главную), `load_contacts` (загрузить контакты), `get_menu_choice` (получить пункт меню), `look_up` (найти), `add` (добавить), `change` (изменить), `delete` (удалить) и `save_contacts` (сохранить контакты). Вместо того чтобы приводить всю программу целиком, сперва исследуем начальную часть, которая включает инструкции `import`, глобальные константы и главную функцию `main`.

Программа 10.19 (contact_manager.py). Главная функция

```

1 # Эта программа управляет контактами.
2 import contact
3 import pickle
4
5 # Глобальные константы для пунктов меню.
6 LOOK_UP = 1
7 ADD = 2
8 CHANGE = 3

```

```
9 DELETE = 4
10 QUIT = 5
11
12 # Глобальная константа для имени файла.
13 FILENAME = 'contacts.dat'
14
15 # Главная функция.
16 def main():
17     # Загрузить существующий словарь контактов
18     # и присвоить его переменной mycontacts.
19     mycontacts = load_contacts()
20
21     # Инициализировать переменную для выбора пользователя.
22     choice = 0
23
24     # Обрабатывать варианты выбора пунктов меню до тех пор,
25     # пока пользователь не пожелает выйти из программы.
26     while choice != QUIT:
27         # Получить выбранный пользователем пункт меню.
28         choice = get_menu_choice()
29
30         # Обработать выбранный вариант действий.
31         if choice == LOOK_UP:
32             look_up(mycontacts)
33         elif choice == ADD:
34             add(mycontacts)
35         elif choice == CHANGE:
36             change(mycontacts)
37         elif choice == DELETE:
38             delete(mycontacts)
39
40     # Сохранить словарь mycontacts в файле.
41     save_contacts(mycontacts)
42
```

Строка 2 импортирует модуль `contact`, который содержит класс `Contact`. Стока 3 импортирует модуль `pickle`. Глобальные константы, которые инициализированы в строках 6–10, используются для проверки выбранного пользователем пункта меню. Константа `FILENAME`, инициализированная в строке 13, содержит имя файла, который будет содержать законсервированную копию словаря, т. е. `contacts.dat`.

В функции `main` строка 19 вызывает функцию `load_contacts`. Следует иметь в виду, что если программа уже выполнялась ранее и имена были добавлены в словарь, то эти имена были сохранены в файле `contacts.dat`. Функция `load_contacts` открывает файл, извлекает из него словарь и возвращает ссылку на словарь. Если программа еще не выполнялась, то файл `contacts.dat` не существует. В этом случае функция `load_contacts` создает пустой словарь и возвращает на него ссылку. Так, после исполнения инструкции в строке 19 переменная `mycontacts` ссылается на словарь. Если программа прежде уже выполнялась, то `mycontacts`

ссылается на словарь, содержащий объекты `Contact`. Если программа выполняется впервые, то `mycontacts` ссылается на пустой словарь.

Строка 22 инициализирует переменную `choice` значением 0. Эта переменная будет содержать выбранный пользователем пункт меню.

Цикл `while`, который начинается в строке 26, повторяется до тех пор, пока пользователь не примет решение выйти из программы. Внутри цикла строка 28 вызывает функцию `get_menu_choice`. Эта функция выводит приведенное ниже меню:

1. Найти контактное лицо
2. Добавить новое контактное лицо
3. Изменить существующее контактное лицо
4. Удалить контактное лицо
5. Выйти из программы

Выбранный пользователем вариант возвращается из функции `get_menu_choice` и присваивается переменной `choice`.

Инструкция `if-elif` в строках 31–38 обрабатывает выбранный пользователем пункт меню. Если пользователь выбирает пункт 1, то строка 32 вызывает функцию `look_up`. Если пользователь выбирает пункт 2, то строка 34 вызывает функцию `add`. Если пользователь выбирает пункт 3, то строка 36 вызывает функцию `change`. Если пользователь выбирает пункт 4, то строка 38 вызывает функцию `delete`.

Когда пользователь выбирает из меню пункт 5, цикл `while` прекращает повторяться, и исполняется инструкция в строке 41. Эта инструкция вызывает функцию `save_contacts`, передавая в качестве аргумента словарь `mycontacts`. Функция `save_contacts` сохраняет словарь `mycontacts` в файле `contacts.dat`.

Далее идет функция `load_contacts`.

Программа 10.19 (продолжение). Функция `load_contacts`

```

43 def load_contacts():
44     try:
45         # Открыть файл contacts.dat.
46         input_file = open(FILENAME, 'rb')
47
48         # Расконсервировать словарь.
49         contact_dct = pickle.load(input_file)
50
51         # Закрыть файл phone_inventory.dat.
52         input_file.close()
53     except IOError:
54         # Не получилось открыть файл, поэтому
55         # создать пустой словарь.
56         contact_dct = {}
57
58     # Вернуть словарь.
59     return contact_dct
60

```

Внутри группы `try` строка 46 пытается открыть файл `contacts.dat`. Если файл успешно открыт, то строка 49 загружает из него объект-словарь, расконсервирует его и присваивает его переменной `contact_dct`. Стока 52 закрывает файл.

Если файл `contacts.dat` не существует (это будет в случае, если программа выполняется впервые), то инструкция в строке 46 вызывает исключение `IOError`. Это приводит к тому, что программа перескакивает к выражению `except` в строке 53. Затем оператор в строке 56 создает пустой словарь и присваивает его переменной `contact_dct`.

Оператор в строке 59 возвращает переменную `contact_dct`.

Далее идет функция `get_menu_choice`.

Программа 10.19 (продолжение). Функция `get_menu_choice`

```
61 # Функция get_menu_choice выводит меню и получает
62 # проверенный на допустимость выбранный пользователем пункт.
63 def get_menu_choice():
64     print()
65     print('Меню')
66     print('-----')
67     print('1. Найти контактное лицо')
68     print('2. Добавить новое контактное лицо')
69     print('3. Изменить существующее контактное лицо')
70     print('4. Удалить контактное лицо')
71     print('5. Выйти из программы')
72     print()
73
74     # Получить выбранный пользователем пункт меню.
75     choice = int(input('Введите выбранный пункт: '))
76
77     # Проверить выбранный пункт на допустимость.
78     while choice < LOOK_UP or choice > QUIT:
79         choice = int(input('Введите выбранный пункт: '))
80
81     # Вернуть выбранный пользователем пункт.
82     return choice
83
```

Инструкции в строках 64–72 выводят на экран меню. Стока 75 предлагает пользователю ввести выбранный пункт. Введенное значение приводится к типу `int` и присваивается переменной `choice`. Цикл `while` в строках 78–79 проверяет введенное пользователем значение на допустимость и при необходимости предлагает пользователю ввести выбранный пункт повторно. Как только вводится допустимый пункт меню, этот пункт возвращается из функции в строке 82.

Далее идет функция `look_up`.

Программа 10.19 (продолжение). Функция `look_up`

```

84 # Функция look_up отыскивает элемент
85 # в заданном словаре.
86 def look_up(mycontacts):
87     # Получить искомое имя.
88     name = input('Введите имя: ')
89
90     # Отыскать его в словаре.
91     print(mycontacts.get(name, 'Это имя не найдено.'))
92

```

Задача функции `look_up` — позволить пользователю найти заданное контактное лицо. В качестве аргумента она принимает словарь `mycontacts`. Стока 88 предлагает пользователю ввести имя, а строка 91 передает это имя в словарную функцию `get` в качестве аргумента. В результате исполнения строки 91 произойдет одно из приведенных ниже действий.

- ◆ Если указанное имя в словаре найдено, то метод `get()` возвращает ссылку на объект `Contact`, который связан с этим именем. Затем объект `Contact` передается в качестве аргумента в функцию `print`. Функция `print` показывает строковое значение, которое возвращается из метода `__str__()` объекта `Contact`.
- ◆ Если указанное в качестве ключа имя в словаре не найдено, метод `get()` возвращает строковый литерал 'Это имя не найдено.', который выводится функцией `print`.

Далее идет функция `add`.

Программа 10.19 (продолжение). Функция `add`

```

93 # Функция add добавляет новую запись
94 # в указанный словарь.
95 def add(mycontacts):
96     # Получить контактную информацию.
97     name = input('Имя: ')
98     phone = input('Телефон: ')
99     email = input('Электронный адрес: ')
100
101    # Создать именованную запись с объектом Contact.
102    entry = contact.Contact(name, phone, email)
103
104    # Если имя в словаре не существует, то
105    # добавить его в качестве ключа со связанным с ним
106    # значением в виде объекта.
107    if name not in mycontacts:
108        mycontacts[name] = entry
109        print('Запись добавлена.')
110    else:
111        print('Это имя уже существует.')
112

```

Задача функции `add` состоит в том, чтобы позволить пользователю добавить в словарь новое контактное лицо. В качестве аргумента она принимает словарь `mycontacts`. Строки 97–99 предлагают пользователю ввести имя, телефонный номер и электронный адрес. Стока 102 создает новый объект `Contact`, инициализированный введенными пользователем данными.

Инструкция `if` в строке 107 определяет, есть ли это имя в словаре. Если его нет, то строка 108 добавляет вновь созданный объект `Contact` в словарь, а строка 109 печатает сообщение о том, что новые данные добавлены. В противном случае в строке 111 печатается сообщение о том, что запись уже существует.

Далее идет функция `change`.

Программа 10.19 (продолжение). Функция `change`

```
113 # Функция change изменяет существующую
114 # запись в указанном словаре.
115 def change(mycontacts):
116     # Получить искомое имя.
117     name = input('Введите имя: ')
118
119     if name in mycontacts:
120         # Получить новый телефонный номер.
121         phone = input('Введите новый телефонный номер: ')
122
123         # Получить новый электронный адрес.
124         email = input('Введите новый электронный адрес: ')
125
126         # Создать именованную запись с объектом Contact.
127         entry = contact.Contact(name, phone, email)
128
129         # Обновить запись.
130         mycontacts[name] = entry
131         print('Информация обновлена.')
132     else:
133         print('Это имя не найдено.')
134
```

Задача функции `change` — позволить пользователю изменить существующее контактное лицо в словаре. В качестве аргумента она принимает словарь `mycontacts`. Стока 117 получает от пользователя имя. Инструкция `if` в строке 119 определяет, есть ли имя в словаре. Если да, то строка 121 получает новый телефонный номер, а строка 124 — новый электронный адрес. Стока 127 создает новый объект `Contact`, инициализированный существующим именем, новым телефонным номером и электронным адресом. Стока 130 сохраняет новый объект `Contact` в словаре, используя существующее имя в качестве ключа.

Если указанного имени в словаре нет, то строка 133 печатает соответствующее сообщение.

Далее идет функция `delete`.

Программа 10.19 (продолжение). Функция `delete`

```

135 # Функция delete удаляет запись
136 # из указанного словаря.
137 def delete(mycontacts):
138     # Получить искомое имя.
139     name = input('Введите имя: ')
140
141     # Если имя найдено, то удалить запись.
142     if name in mycontacts:
143         del mycontacts[name]
144         print('Запись удалена.')
145     else:
146         print('Это имя не найдено.')
147

```

Задача функции `delete` — позволить пользователю удалить существующее контактное лицо из словаря. В качестве аргумента она принимает словарь `mycontacts`. Стока 139 получает от пользователя имя. Инструкция `if` в строке 142 определяет, есть ли имя в словаре. Если да, то строка 143 его удаляет, а строка 144 печатает сообщение о том, что запись была удалена. Если имени в словаре нет, то строка 146 печатает соответствующее сообщение.

Далее идет функция `save_contacts`.

Программа 10.19 (окончание). Функция `save_contacts`

```

148 # Функция save_contacts консервирует указанный
149 # объект и сохраняет его в файле контактов.
150 def save_contacts(mycontacts):
151     # Открыть файл для записи.
152     output_file = open(FILENAME, 'wb')
153
154     # Законсервировать словарь и сохранить его.
155     pickle.dump(mycontacts, output_file)
156
157     # Закрыть файл.
158     output_file.close()
159
160 # Вызвать главную функцию.
161 if __name__ == '__main__':
162     main()

```

Функция `save_contacts` вызывается непосредственно перед тем, как программа закончит выполняться. В качестве аргумента она принимает словарь `mycontacts`. Стока 152 открывает файл `contacts.dat` для записи. Стока 155 консервирует словарь `mycontacts` и сохраняет его в файле. Стока 158 закрывает файл.

Приведенный ниже результат показывает два сеанса работы с программой. Демонстрационный вывод показывает не все, что программа может делать, и тем не менее видно, как контактная информация сохраняется, когда программа заканчивает работу, и как она затем загружается, когда программа выполняется снова.

Вывод 1 программы (вводимые данные выделены жирным шрифтом)

Меню

-
- 1. Найти контактное лицо
 - 2. Добавить новое контактное лицо
 - 3. Изменить существующее контактное лицо
 - 4. Удалить контактное лицо
 - 5. Выйти из программы

Введите выбранный пункт: 2

Имя: **Мэт Гольдштейн**

Телефон: 617-555-1234

Электронный адрес: matt@fakecompany.com

Запись добавлена.

Меню

-
- 1. Найти контактное лицо
 - 2. Добавить новое контактное лицо
 - 3. Изменить существующее контактное лицо
 - 4. Удалить контактное лицо
 - 5. Выйти из программы

Введите выбранный пункт: 2

Имя: **Хорхе Руис**

Телефон: 919-555-1212

Электронный адрес: **jorge@myschool.edu**

Запись добавлена.

Меню

-
- 1. Найти контактное лицо
 - 2. Добавить новое контактное лицо
 - 3. Изменить существующее контактное лицо
 - 4. Удалить контактное лицо
 - 5. Выйти из программы

Введите выбранный пункт: 5

Вывод 2 программы (вводимые данные выделены жирным шрифтом)

Меню

-
- 1. Найти контактное лицо
 - 2. Добавить новое контактное лицо

3. Изменить существующее контактное лицо
4. Удалить контактное лицо
5. Выйти из программы

Введите выбранный пункт: 1

Введите имя: Мэт Гольдштейн

Имя: Мэт Гольдштейн

Телефон: 617-555-1234

Электронная почта: matt@fakecompany.com

Меню

1. Найти контактное лицо
2. Добавить новое контактное лицо
3. Изменить существующее контактное лицо
4. Удалить контактное лицо
5. Выйти из программы

Введите выбранный пункт: 1

Введите имя: Хорхе Руис

Имя: Хорхе Руис

Телефон: 919-555-1212

Электронная почта: jorge@myschool.edu

Меню

1. Найти контактное лицо
2. Добавить новое контактное лицо
3. Изменить существующее контактное лицо
4. Удалить контактное лицо
5. Выйти из программы

Введите выбранный пункт: 5



Контрольная точка

10.12. Что такое атрибут экземпляра?

10.13. Программа создает 10 экземпляров класса Coin. Сколько атрибутов `_sideup` существует в оперативной памяти?

10.14. Что такое метод-получатель? Что такое метод-мутатор?

10.4 Приемы конструирования классов

Унифицированный язык моделирования

Во время разработки класса часто полезно нарисовать диаграмму UML (Unified Modeling Language — унифицированный язык моделирования). Это набор стандартных диаграмм для графического изображения объектно-ориентированных систем. На рис. 10.9 показан общий макет диаграммы UML для класса. Обратите внимание, что диаграмма представляет собой прямоугольник, который разделен на три секции. Верхняя секция — это то место, где пишется имя класса. Средняя секция содержит список атрибутов данных класса. Нижняя секция содержит список методов класса.



РИС. 10.9. Общий макет диаграммы UML для класса

Согласно общему макету на рис. 10.10 и 10.11 представлены диаграммы UML для классов Coin и CellPhone, которые вы встречали ранее в этой главе. Обратите внимание, что ни в одном из методов мы не показали параметр `self`, поскольку предполагается, что параметр `self` является обязательным.

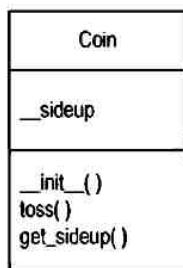


РИС. 10.10. Диаграмма UML для класса Coin



РИС. 10.11. Диаграмма UML для класса CellPhone

Идентификация классов в задаче

Во время разработки объектно-ориентированной программы одной из первых задач является идентификация классов, которые вам требуется создать. Как правило, ваша цель состоит в том, чтобы идентифицировать различные типы реальных объектов задачи и затем создать классы для этих типов объектов в своем приложении.

За прошедшие годы специалисты в области программного обеспечения разработали многочисленные методы идентификации классов в поставленной задаче. Один из простых и популярных методов связан с приведенными ниже шагами:

1. Получить письменное описание предметной области задачи.
2. Идентифицировать в описании все именные группы (включая существительные, местоимения и именные словосочетания). Каждая из них является потенциальным классом.
3. Уточнить полученный список, чтобы он включал только те классы, которые относятся к задаче.

Давайте взглянем на каждый из этих шагов поближе.

Составление описания предметной области задачи

Предметная область задачи — это набор объектов реального мира, участников и крупных событий, связанных с задачей. Если вы адекватно понимаете природу задачи, которую пытаетесь решить, то можете составить описание проблемной области задачи самостоятельно. Если же природа задачи до конца вам не ясна, то за вас это может сделать эксперт.

Предположим, что мы пишем программу, которую менеджер авторемонтной фирмы "Автоцех" будет использовать для подготовки справочных цен на услуги для своих клиентов. Вот описание, которое, возможно, составил эксперт или же лично сам владелец фирмы.

Авторемонтная фирма "Автоцех" проводит техническое обслуживание зарубежных легковых автомобилей и специализируется на обслуживании легковых автомобилей марок Mercedes, Porsche и BMW. Когда клиент доставляет легковой автомобиль в авторемонтный цех, менеджер получает имя, адрес и телефонный номер клиента. Затем менеджер определяет изготовителя, модель и год производства легкового автомобиля и выдает клиенту расценки на услуги. Расценки на услуги показывают предполагаемую стоимость запасных частей, предполагаемую стоимость трудозатрат, налог с продаж и общую стоимость предполагаемых расходов.

Описание предметной области задачи должно включать любой из приведенных ниже пунктов:

- ◆ физические объекты, такие как автомобили, механизмы или изделия;
- ◆ любую роль, выполняемую человеком, такую как менеджер, сотрудник, клиент, учитель, студент и т. д.;
- ◆ результаты делового процесса, такие как заказ клиента или же в данном случае расценки на услуги;
- ◆ элементы, связанные с ведением учета, такие как история обслуживания клиента и зарплатные ведомости.

Идентификация всех именных групп

Следующий шаг состоит в идентификации всех именных групп. (Если описание содержит местоимения, то включить их тоже.) Вот еще один взгляд на предыдущее описание проблемной области задачи.

На этот раз именные группы выделены жирным шрифтом.

Авторемонтная фирма "Автоцех" проводит техническое обслуживание зарубежных легковых автомобилей и специализируется на обслуживании легковых автомобилей

марок **Mercedes**, **Porsche** и **BMW**. Когда клиент доставляет легковой автомобиль в авторемонтный цех, менеджер получает имя, адрес и телефонный номер клиента. Затем менеджер определяет изготовителя, модель и год производства легкового автомобиля и выдает клиенту расценки на услуги. Расценки на услуги показывают оценочную стоимость запчастей, оценочную стоимость трудозатрат, налог с продаж и общую оценочную стоимость расходов.

Обратите внимание, что некоторые именные группы повторяются. Приведенный ниже список показывает все именные группы без повторов:

BMW
Mercedes
Porsche
авторемонтная фирма "Автоцех"
адрес
год
зарубежные автомобили
изготовитель
имя
клиент
легковой автомобиль
легковые автомобили
менеджер
модель
налог с продаж
общая оценочная стоимость расходов
оценочная стоимость запчастей
оценочная стоимость трудозатрат
расценки на услуги
телефонный номер
цех

Уточнение списка именных групп

Именные группы, которые появляются в описании задачи, — это всего лишь кандидаты на роль классов. Может оказаться, что конструировать классы для них всех не понадобится. Следующий шаг состоит в совершенствовании списка, чтобы оставить только те классы, которые необходимы для решения рассматриваемой задачи. Мы обратимся к общим причинам, почему именная группа может быть устранена из списка потенциальных классов.

1. Некоторые именные группы по существу означают одно и то же.

В этом примере приведенные ниже наборы именных групп относятся к одному и тому же:

- легковой автомобиль, легковые автомобили и зарубежные легковые автомобили.
Все они относятся к общему понятию "легковой автомобиль".
- авторемонтная фирма "Автоцех" и цех.
Оба обозначают авторемонтную фирму "Автоцех".

Можно остановиться на единственном классе для каждого из них. В этом примере мы по своему усмотрению удалим из списка **легковые автомобили** и **зарубежные легковые автомобили** и будем использовать понятие "**легковой автомобиль**". Аналогично мы вычеркнем из списка **авторемонтную фирму "Автоцех"** и будем использовать слово "**цех**". Получаем обновленный список потенциальных классов.

BMW	
Mercedes	
Porsche	
авторемонтная фирма "Автоцех"	
адрес	
год	
зарубежные легковые автомобили	
изготовитель	
имя	
клиент	
легковой автомобиль	
легковые автомобили	
менеджер	
модель	
налог с продаж	
общая оценочная стоимость расходов	
оценочная стоимость запчастей	
оценочная стоимость трудозатрат	
расценки на услуги	
телефонный номер	
цех	

Поскольку понятия "**легковой автомобиль**", "**легковые автомобили**" и "**зарубежные легковые автомобили**" в этой задаче означают одно и то же, мы вычеркнули "**легковые автомобили**" и "**зарубежные легковые автомобили**". Кроме того, поскольку понятие **„авторемонтная фирма "Автоцех"“** и **“цех”** подразумевают одно и то же, мы вычеркнули понятие **„авторемонтная фирма "Автоцех"“**.

2. Некоторые именные группы могут представлять элементы, которые не требуются для решения задачи.

Оперативный анализ описания задачи напоминает нам о том, что именно должно делать наше приложение: распечатывать расценки на услуги. В этом примере можно удалить из списка два ненужных класса.

- Можно вычеркнуть из списка "**цех**", потому что наше приложение должно касаться только расценок на отдельные услуги. Оно не должно работать с общефирменной информацией или определять таковую. Если бы описание задачи требовало от нас, чтобы мы поддерживали суммарный объем всех расценок на услуги, то было бы целесообразно иметь класс для цеха.
- Нам не потребуется класс для "**менеджера**", потому что постановка задачи не указывает на обработку какой-либо информации о менеджере. Если бы имелось несколько менеджеров цеха и описание задачи требовало, чтобы мы вели учет, какой именно менеджер какую расценку на услуги составил, то было бы целесообразно иметь класс для менеджера.

Вот обновленный список потенциальных классов в данной точке.

BMW	
Mercedes	
Porsche	
автотехника "Автоцех"	
адрес	
год	
зарубежные легковые автомобили	
изготовитель	
имя	Описание задачи не указывает на необходимость обработки какой-либо информации о цехе либо какой-либо информации о менеджере , поэтому мы их вычеркнули из списка.
клиент	
легковой автомобиль	
легковые автомобили	
менеджер	
модель	
налог с продаж	
общая оценочная стоимость расходов	
оценочная стоимость запчастей	
оценочная стоимость трудозатрат	
расценки на услуги	
телефонный номер	
цех	

3. Некоторые именные группы могут представлять не классы, а объекты.

Мы можем удалить **Mercedes**, **Porsche** и **BMW** как классы, потому что в этом примере все они представляют конкретные автомобили и могут считаться экземплярами класса "**легковой автомобиль**". В этой точке обновленный список потенциальных классов выглядит так.

BMW	
Mercedes	
Porsche	
автотехника "Автоцех"	
адрес	
год	
зарубежные легковые автомобили	
изготовитель	
имя	Мы вычеркнули Mercedes , Porsche и BMW , потому что все они являются экземплярами класса " легковой автомобиль ". Это означает, что эти именные группы идентифицируют не классы, а объекты.
клиент	
легковой автомобиль	
легковые автомобили	
менеджер	
модель	
налог с продаж	

общая оценочная стоимость расходов
 оценочная стоимость запчастей
 оценочная стоимость трудозатрат
 расценки на услуги
 телефонный номер
 цех



ПРИМЕЧАНИЕ

Некоторые объектно-ориентированные разработчики учитывают также, находится ли именная группа во множественном или единственном числе. Иногда именная группа во множественном числе будет указывать на класс, а в единственном числе — на объект.

4. Некоторые именные группы могут представлять простые значения, которые могут быть присвоены переменной, и не требуют класса.

Напомним, что класс содержит атрибуты данных и методы. Атрибуты данных — это связанные между собой элементы, которые хранятся в объекте класса и определяют состояние объекта. Методы — это действия или виды поведения, которые могут выполняться объектом класса. Если именная группа представляет вид элемента, который не обладает какими-то идентифицируемыми атрибутами данных или методами, то его, по-видимому, можно из списка вычеркнуть. Для того чтобы определить, представляет ли именная группа элемент, который имеет атрибуты данных и методы, необходимо задать о ней следующие вопросы:

- Будет ли использоваться группа значений для представления состояния этого элемента?
- Существуют ли какие-либо очевидные действия, которые этот элемент должен выполнять?

Если ответы на оба этих вопроса отрицательные, то именная группа, скорее всего, представляет значение, которое может быть сохранено в простой переменной. Если применить этот тест к каждой именной группе, которая осталась в списке, то придем к заключению, что следующие из них, вероятно, не являются классами: **адрес, оценочная стоимость трудозатрат, оценочная стоимость запчастей, изготовитель, модель, имя, налог с продаж, телефонный номер, общая оценочная стоимость расходов и год**. Все они представляют простые символьные или числовые значения, которые могут быть сохранены в переменных. Вот обновленный список потенциальных классов:

BMW

Mercedes

Porsche

автотехника фирма "Автоцех"

адрес

Фед

зарубежные легковые автомобили

изготовитель

имя

клиент

легковой автомобиль

легковые автомобили

Мы устранили **адрес, оценочная стоимость трудозатрат, оценочная стоимость запчастей, изготовитель, модель, имя, налог с продаж, телефонный номер, общая оценочная стоимость расходов и год** как классы, потому что они представляют простые значения, которые могут храниться в переменных.

~~модель~~
~~налог с продаж~~
~~общая оценочная стоимость расходов~~
~~оценочная стоимость запчастей~~
~~оценочная стоимость трудозатрат~~
~~расценки на услуги~~
~~телефонный номер~~
~~управляющий~~
~~цех~~

Как видно из полученного списка, мы удалили все, кроме **легкового автомобиля, клиента и расценок на услуги**. Это означает, что в приложении будут нужны классы для представления легковых автомобилей, клиентов и расценок на услуги. Мы напишем класс *Car* (Легковой автомобиль), класс *Customer* (Клиент) и класс *ServiceQuote* (Расценки на услуги).

Идентификация обязанностей класса

После того как были идентифицированы классы, следующая задача состоит в том, чтобы идентифицировать обязанности каждого класса. *Обязанности класса* это:

- ◆ то, что класс обязан знать;
- ◆ действия, которые класс обязан выполнять.

Выяснив, что именно класс обязан знать, получаем атрибуты данных класса. Аналогичным образом, вычислив действия, которые класс обязан выполнять, получаем его методы.

Часто целесообразно задать следующие вопросы: "Что именно класс должен знать и что именно класс должен делать в контексте поставленной задачи?" В первую очередь за ответом следует обратиться к описанию предметной области задачи. Многое из того, что класс должен знать и делать, будет там упомянуто. Однако некоторые обязанности класса непосредственно могут быть не указаны в предметной области задачи, поэтому нередко требуется дальнейшее рассмотрение. Давайте применим эту методологию к классам, которые мы ранее идентифицировали в нашей предметной области задачи.

Класс *Customer*

Что именно класс *Customer* должен знать в контексте предметной области нашей задачи? В описании непосредственно упоминаются приведенные ниже элементы, все они являются атрибутами данных клиента:

- ◆ имя клиента;
- ◆ адрес клиента;
- ◆ телефонный номер клиента.

Все эти значения могут быть представлены как строковые и храниться в качестве атрибутов данных. Класс *Customer* потенциально может знать многие другие вещи. Одна из ошибок, которая может быть сделана в этот момент, состоит в идентификации слишком большого количества характеристик, которые объект обязан знать. В некоторых приложениях класс *Customer* может знать электронный адрес клиента. Данная конкретная предметная область

не упоминает, что электронный адрес клиента используется для какой-либо цели, поэтому нам не следует его включать в обязанности класса.

Теперь давайте идентифицируем методы класса. Что именно класс `Customer` должен делать в контексте нашей предметной области задачи? Единственными очевидными действиями являются:

- ◆ инициализировать объект класса `Customer`;
- ◆ задать и вернуть имя клиента;
- ◆ задать и вернуть адрес клиента;
- ◆ задать и вернуть телефонный номер клиента.

Из этого списка мы видим, что класс `Customer` будет иметь метод `__init__()`, а также методы-мутаторы и методы-получатели атрибутов данных. На рис. 10.12 показана UML-диаграмма для класса `Customer`. Код Python для этого класса приведен в программе 10.20.

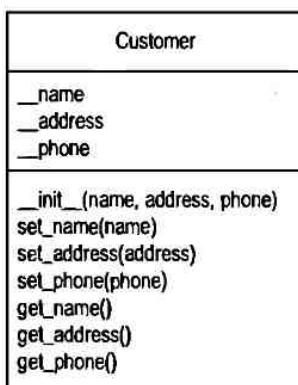


РИС. 10.12. Диаграмма UML для класса `Customer`

Программа 10.20 (customer.py)

```

1 # Класс Customer.
2 class Customer:
3     def __init__(self, name, address, phone):
4         self.__name = name
5         self.__address = address
6         self.__phone = phone
7
8     def set_name(self, name):
9         self.__name = name
10
11    def set_address(self, address):
12        self.__address = address
13
14    def set_phone(self, phone):
15        self.__phone = phone
16
  
```

```

17     def get_name(self):
18         return self._name
19
20     def get_address(self):
21         return self._address
22
23     def get_phone(self):
24         return self._phone

```

Класс *Car*

Что именно класс *Car* должен знать в контексте предметной области нашей задачи? Приведенные ниже элементы являются атрибутами данных легкового автомобиля и упомянуты в предметной области задачи:

- ◆ изготовитель легкового автомобиля;
- ◆ модель легкового автомобиля;
- ◆ год изготовления легкового автомобиля.

Теперь идентифицируем методы класса. Что именно класс *Car* должен делать в контексте предметной области нашей задачи? И снова единственные очевидные действия представлены стандартным набором методов, которые мы найдем в большинстве классов (метод `__init__()`, методы-мутаторы и методы-получатели). В частности, следующие ниже действия:

- ◆ инициализировать объект класса *Car*;
- ◆ задать и получить изготовителя легкового автомобиля;
- ◆ задать и получить модель легкового автомобиля;
- ◆ задать и получить год изготовления легкового автомобиля.

На рис. 10.13 показана UML-диаграмма класса *Car* по состоянию на текущий момент. Исходный код Python для этого класса приведен в программе 10.21.

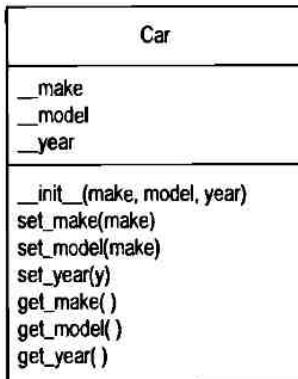


РИС. 10.13. Диаграмма UML для класса *Car*

Программа 10.21 (car.py)

```

1 # Класс Car.
2 class Car:
3     def __init__(self, make, model, year):
4         self.__make = make
5         self.__model = model
6         self.__year = year
7
8     def set_make(self, make):
9         self.__make = make
10
11    def set_model(self, model):
12        self.__model = model
13
14    def set_year(self, year):
15        self.__year = year
16
17    def get_make(self):
18        return self.__make
19
20    def get_model(self):
21        return self.__model
22
23    def get_year(self):
24        return self.__year

```

Класс ServiceQuote

Что именно класс *ServiceQuote* должен знать в контексте предметной области нашей задачи? Упоминаются следующие элементы:

- ◆ оценочная стоимость запчастей;
- ◆ оценочная стоимость трудозатрат;
- ◆ налог с продаж;
- ◆ общая оценочная стоимость расходов.

Для этого класса нам, в частности, потребуются следующие методы: метод *__init__()*, методы-мутаторы и методы-получатели атрибутов "Оценочная стоимость запчастей" и "Оценочная стоимость трудозатрат". Кроме того, для класса будут нужны методы, которые вычисляют и возвращают налог с продаж и общую оценочную стоимость расходов. На рис. 10.14 показана UML-диаграмма для класса *ServiceQuote*. В программе 10.22 приведен пример класса в коде Python.

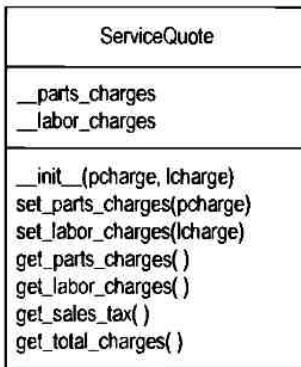


РИС. 10.14. Диаграмма UML для класса ServiceQuote

Программа 10.22 (servicequote.py)

```

1 # Константа для ставки налога с продаж.
2 TAX_RATE = 0.05
3
4 # Класс ServiceQuote.
5 class ServiceQuote:
6     def __init__(self, pcharge, lcharge):
7         self.__parts_charges = pcharge
8         self.__labor_charges = lcharge
9
10    def set_parts_charges(self, pcharge):
11        self.__parts_charges = pcharge
12
13    def set_labor_charges(self, lcharge):
14        self.__labor_charges = lcharge
15
16    def get_parts_charges(self):
17        return self.__parts_charges
18
19    def get_labor_charges(self):
20        return self.__labor_charges
21
22    def get_sales_tax(self):
23        return __parts_charges * TAX_RATE
24
25    def get_total_charges(self):
26        return __parts_charges + __labor_charges + \
27            (__parts_charges * TAX_RATE)

```

Это только начало

Процесс, который мы обсудили в этом разделе, следует рассматривать всего лишь как отправную точку. Важно понимать, что разработка объектно-ориентированного приложения представляет собой итеративный процесс. Для того чтобы идентифицировать все классы, которые вам потребуются, и определять все их обязанности, может потребоваться сделать несколько попыток. По ходу развития процесса разработки вы более глубоко станете понимать задачу и, следовательно, увидите пути совершенствования проекта.



Контрольная точка

- 10.15. Типичная диаграмма UML для класса имеет три секции. Что размещается в них?
- 10.16. Что такое предметная область задачи?
- 10.17. Кто должен составлять описание предметной области задачи во время разработки объектно-ориентированного приложения?
- 10.18. Каким образом идентифицируются потенциальные классы в описании предметной области задачи?
- 10.19. Что такое обязанности класса?
- 10.20. Какие два вопроса следует задать, чтобы определить обязанности класса?
- 10.21. Всегда ли все действия класса будут упомянуты в описании предметной области задачи?

Вопросы для повторения

Множественный выбор

1. Практика _____ программирования сконцентрирована на создании функций, отделенных от данных, с которыми они работают.
 - а) модульного;
 - б) процедурного;
 - в) функционального;
 - г) объектно-ориентированного.
2. Практика _____ программирования сконцентрирована на создании объектов.
 - а) объектно-центрированного;
 - б) объектного;
 - в) процедурного;
 - г) объектно-ориентированного.
3. _____ — это компонент класса, который ссылается на данные.
 - а) метод;
 - б) экземпляр;
 - в) атрибут данных;
 - г) модуль.

4. Объект — это _____.
- проект;
 - форма для печенья;
 - переменная;
 - экземпляр.
5. Поступая так, вы прячете атрибуты класса от программного кода, находящегося за пределами класса.
- избегая использования параметра `self` при создании атрибута;
 - начиная имя атрибута двумя символами подчеркивания;
 - начиная имя атрибута с `private_`;
 - начиная имя атрибута символом @.
6. Метод-_____ получает значение атрибута данных и при этом его не изменяет.
- извлечатель;
 - конструктор;
 - мутатор;
 - получатель.
7. Метод-_____ сохраняет значение в атрибуте данных либо каким-либо образом изменяет его значение.
- виdeoизменитель;
 - конструктор;
 - мутатор;
 - получатель.
8. Метод _____ автоматически вызывается при создании объекта.
- `__init__()`;
 - `init()`;
 - `__str__()`;
 - `__object__()`.
9. Если класс имеет метод `__str__()`, какой из нижеперечисленных способов вызывает этот метод?
- он вызывается подобно любому другому методу: `объект.__str__()`;
 - путем передачи экземпляра класса во встроенную функцию `str`;
 - этот метод автоматически вызывается при создании объекта;
 - путем передачи экземпляра класса во встроенную функцию `state`.
10. Набор стандартных диаграмм для графического изображения объектно-ориентированных систем обеспечивается за счет _____.
- унифицированного языка моделирования;
 - блок-схем;

- в) псевдокода;
г) системы иерархии объектов.
11. В одном из подходов к идентификации классов в задаче программист идентифицирует _____ в описании предметной области задачи.
- а) глаголы;
б) прилагательные;
в) существительные;
г) именные группы.
12. В одном из подходов к идентификации атрибутов и методов данных класса программист идентифицирует _____ класса.
- а) обязанности;
б) имя;
в) синонимы;
г) существительные.

Истина или ложь

1. Практика процедурного программирования сконцентрирована на создании объектов.
2. Возможность многократного использования объектов — важный фактор востребованности ООП.
3. В ООП общепринятой практикой является обеспечение доступности всех атрибутов данных класса для инструкций, находящихся за пределами класса.
4. Метод класса не должен иметь параметр `self`.
5. Имя атрибута, начинающееся двумя символами подчеркивания, помогает скрыть атрибут от программного кода, находящегося за пределами класса.
6. Метод `__str__()` невозможно вызвать напрямую.
7. Один из приемов найти классы, необходимые для объектно-ориентированной программы, состоит в том, чтобы идентифицировать все глаголы в описании предметной области задачи.

Короткий ответ

1. Что такое инкапсуляция?
2. Почему атрибуты данных объекта должны быть скрыты от программного кода, находящегося за пределами класса?
3. В чем разница между классом и экземпляром класса?
4. Приведенная ниже инструкция вызывает метод объекта. Как называется этот метод? Как называется переменная, которая ссылается на объект?
`wallet.get_dollar()`
5. На что ссылается параметр `self`, когда выполняется метод `__init__()`?

6. Каким образом в классе Python атрибут скрывается от программного кода, находящегося за пределами класса?
7. Как вызывается метод `__str__()`?

Алгоритмический тренажер

1. Предположим, что `my_car` — это имя переменной, которая ссылается на объект, и `go` — это имя метода. Напишите инструкцию, которая использует переменную `my_car` для вызова метода `go()`. (В метод `go()` аргументы не должны передаваться.)
2. Напишите определение класса с именем `Book`. Класс `Book` должен иметь атрибуты данных для заголовка книги, имени автора и имени издателя. Этот класс должен также иметь следующие методы:
 - метод `__init__()` для класса должен принимать аргумент для каждого атрибута данных;
 - методы-получатели и методы-мутаторы для каждого атрибута данных;
 - метод `__str__()`, который возвращает строковое значение, сообщающее о состоянии объекта.
3. Взгляните на приведенное ниже описание предметной области задачи.

Банк предлагает своим клиентам следующие типы счетов: сберегательные счета, текущие счета и счета с процентами по ставке денежного рынка. Клиентам разрешается вносить деньги на банковский счет (тем самым увеличивая остаток на своем счете), снимать деньги с банковского счета (тем самым уменьшая остаток на своем счете) и накапливать процентный доход на банковском счете. Каждый счет имеет процентную ставку.

Допустим, что вы пишете программу, которая вычисляет сумму процентного дохода, накопленного на банковском счете.

- Идентифицируйте потенциальные классы в данной предметной области.
- Уточните список, чтобы он включал только необходимый для этой задачи класс или классы.
- Идентифицируйте обязанности класса или классов.

Упражнения по программированию

1. Класс `Pet`.



Видеозапись "Класс `Pet`" (*The Pet class*)

Напишите класс `Pet` (Домашнее животное), который должен иметь приведенные ниже атрибуты данных:

- `__name` (для клички домашнего животного);
- `__animal_type` (для типа домашнего животного; например, это может быть 'собака', 'кот' и 'птица');
- `__age` (для возраста домашнего животного).

Класс `Pet` должен иметь метод `__init__()`, который создает эти атрибуты. Он также должен иметь приведенные ниже методы:

- метод `set_name()` присваивает значение полю `__name`;
- метод `set_animal_type()` присваивает значение полю `__animal_type`;
- метод `set_age()` присваивает значение полю `__age`;
- метод `get_name()` возвращает значение полю `__name`;
- метод `get_animal_type()` возвращает значение полю `__animal_type`;
- метод `get_age()` возвращает значение полю `__age`.

После написания данного класса напишите программу, которая создает объект класса и предлагает пользователю ввести кличку, тип и возраст своего домашнего животного. Эти данные должны храниться в качестве атрибутов объекта. Примените методы-получатели, чтобы извлечь имя, тип и возраст домашнего животного и показать эти данные на экране.

2. **Класс `Car`.** Напишите класс под названием `Car` (Легковой автомобиль), который имеет приведенные ниже атрибуты данных:

- `__year_model` (для модели указанного года выпуска);
- `__make` (для фирмы-изготовителя автомобиля);
- `__speed` (для текущей скорости автомобиля).

Класс `Car` должен иметь метод `__init__()`, который в качестве аргументов принимает модель указанного года выпуска и фирму-изготовителя. Эти значения должны быть присвоены атрибутам данных `__year_model` и `__make` объекта. Он также должен присвоить 0 атрибуту данных `__speed`.

Этот класс также должен иметь методы:

- метод `accelerate()` (ускоряться) при каждом его вызове должен прибавлять 5 в атрибут данных `__speed`;
- метод `break()` (тормозить) при каждом его вызове должен вычесть 5 из атрибута данных `__speed`;
- метод `get_speed()` (получить скорость) должен возвращать текущую скорость.

Далее разработайте программу, которая создает объект `Car` и пятикратно вызывает метод `accelerate()`. После каждого вызова метода `accelerate()` она должна получать текущую скорость автомобиля и выводить ее на экран. Затем она должна пятикратно вызвать метод `break()`. После каждого вызова метода `break()` она должна получать текущую скорость автомобиля и выводить ее на экран.

3. **Класс персональных данных `Information`.** Разработайте класс, который содержит следующие персональные данные: имя, адрес, возраст и телефонный номер. Напишите соответствующие методы-получатели и методы-мутаторы. Кроме того, напишите программу, которая создает три экземпляра класса. Один экземпляр должен содержать информацию о вас, а два других — информацию о ваших друзьях или членах семьи.

4. **Класс `Employee`.** Напишите класс под названием `Employee`, который в атрибутах содержит данные о сотруднике: имя, идентификационный номер, отдел и должность.

После написания данного класса напишите программу, которая создает три объекта `Employee` с приведенными в табл. 10.1 данными.

Таблица 10.1

Имя	Идентификационный номер	Отдел	Должность
Сьюзан Мейерс	47899	Бухгалтерия	Вице-президент
Марк Джоунс	39119	IT	Программист
Джой Роджерс	81774	Производственный	Инженер

Программа должна сохранить эти данные в трех объектах и затем вывести данные по каждому сотруднику на экран.

5. Класс `RetailItem`. Напишите класс под названием `RetailItem` (Розничная товарная единица), который содержит данные о товаре в розничном магазине. Этот класс должен хранить данные в атрибутах: описание товара, количество единиц на складе и цена. После написания этого класса напишите программу, которая создает три объекта `RetailItem` и сохраняет в них приведенные в табл. 10.2 данные.

Таблица 10.2

	Описание	Количество на складе	Цена
Товар № 1	Пиджак	12	59.95
Товар № 2	Дизайнерские джинсы	40	34.95
Товар № 3	Рубашка	20	24.95

6. Расходы на лечение. Напишите класс под названием `Patient` (Пациент), который имеет атрибуты для приведенных ниже данных:

- имя, отчество и фамилия;
- адрес, город, область и почтовый индекс;
- телефонный номер;
- имя и телефон контактного лица для экстренной связи.

Метод `__init__()` класса `Patient` должен принимать аргумент для каждого атрибута. Класс `Patient` также должен для каждого атрибута иметь методы-получатели и методы-мутаторы.

Затем напишите класс `Procedure`, который представляет пройденную пациентом медицинскую процедуру. Класс `Procedure` должен иметь атрибуты для приведенных ниже данных:

- название процедуры;
- дата процедуры;
- имя врача, который выполнял процедуру;
- стоимость процедуры.

Метод `__init__()` класса `Procedure` должен принимать аргумент для каждого атрибута.

Класс `Procedure` также должен для каждого атрибута иметь методы-получатели и методы-мутаторы. Далее напишите программу, которая создает экземпляр класса `Patient`,

инициализированного демонстрационными данными. Затем создайте три экземпляра класса `Procedure`, инициализированного приведенными в табл. 10.3 данными.

Программа должна вывести на экран информацию о пациенте, сведения обо всех трех процедурах и об общей стоимости всех трех процедур.

Таблица 10.3

Процедура № 1	Процедура № 2	Процедура № 3
Название процедуры: врачебный осмотр	Название процедуры: рентгенография	Название процедуры: анализ крови
Дата: сегодняшняя	Дата: сегодняшняя	Дата: сегодняшняя
Врач: Ирвин	Врач: Джемисон	Врач: Смит
Стоимость: 250.00	Стоимость: 500.00	Стоимость: 200.00

7. **Система управления персоналом.** Это упражнение предполагает создание класса `Employee` из упражнения 4 по программированию. Создайте программу, которая сохраняет объекты `Employee` в словаре. Используйте идентификационный номер сотрудника в качестве ключа. Программа должна вывести меню, которое позволяет пользователю:

- найти сотрудника в словаре;
- добавить нового сотрудника в словарь;
- изменить имя, отдел и должность существующего сотрудника в словаре;
- удалить сотрудника из словаря;
- выйти из программы.

По завершении работы программа должна законсервировать словарь и сохранить его в файле. При каждом запуске программы она должна попытаться загрузить законсервированный словарь из файла. Если файл не существует, то программа должна начать работу с пустого словаря.

8. **Класс `CashRegister`.** Это упражнение предполагает создание класса `RetailItem` из упражнения 5 по программированию. Создайте класс `CashRegister` (Кассовый аппарат), который может использоваться вместе с классом `RetailItem`. Класс `CashRegister` должен иметь внутренний список объектов `RetailItem`, а также приведенные ниже методы.

- Метод `purchase_item()` (приобрести товар) в качестве аргумента принимает объект `RetailItem`. При каждом вызове метода `purchase_item()` объект `RetailItem`, переданный в качестве аргумента, должен быть добавлен в список.
- Метод `get_total()` (получить сумму покупки) возвращает общую стоимость всех объектов `RetailItem`, хранящихся во внутреннем списке объекта `CashRegister`.
- Метод `show_items()` (показать товары) выводит данные об объектах `RetailItem`, хранящихся во внутреннем списке объекта `CashRegister`.
- Метод `clear()` (очистить) должен очистить внутренний список объекта `CashRegister`.

Продемонстрируйте класс `CashRegister` в программе, которая позволяет пользователю выбрать несколько товаров для покупки. Когда пользователь готов рассчитаться за по-

купку, программа должна вывести список всех товаров, которые он выбрал для покупки, а также их общую стоимость.

9. **Викторина.** В этой задаче по программированию следует создать простую викторину для двух игроков. Программа будет работать следующим образом.

- Начиная с игрока 1, каждый игрок по очереди отвечает на 5 вопросов викторины. (Должно быть в общей сложности 10 вопросов.) При выводе вопроса на экран также выводятся 4 возможных ответа. Только один из этих ответов является правильным, и если игрок выбирает правильный ответ, то он зарабатывает очко.
- После того как были выбраны ответы на все вопросы, программа показывает количество очков, заработанное каждым игроком, и объявляет игрока с наибольшим количеством очков победителем.

Для создания этой программы напишите класс `Question` (Вопрос), который будет содержать данные о вопросе викторины. Класс `Question` должен иметь атрибуты для приведенных ниже данных:

- вопрос викторины;
- возможный ответ 1;
- возможный ответ 2;
- возможный ответ 3;
- возможный ответ 4;
- номер правильного ответа (1, 2, 3 или 4).

Класс `Question` также должен иметь соответствующий метод `__init__()`, методы-получатели и методы-мутаторы.

Программа должна содержать список или словарь с 10 объектами `Question`, один для каждого вопроса викторины. Составьте для объектов собственные вопросы викторины по теме или темам по вашему выбору.

11.1

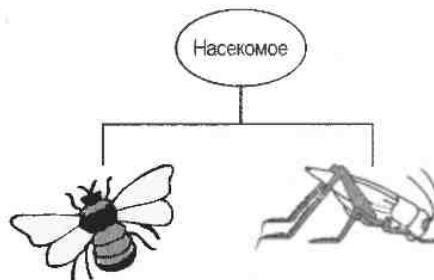
Введение в наследование

Ключевые положения

Наследование позволяет новому классу расширять существующий класс. Новый класс наследует члены расширяемого класса.

Обобщение и конкретизация

В реальном мире можно найти множество объектов, которые являются конкретизированными вариантами других более общих объектов. Например, термин "насекомое" описывает общий, родовой тип существ с различными свойствами. Поскольку кузнечики и шмели являются насекомыми, у них есть все родовые свойства насекомого. Кроме того, у них есть особые, свои свойства. Например, кузнечик умеет прыгать, а шмель жалить. Кузнечики и шмели являются конкретизированными вариантами насекомого (рис. 11.1).



В добавление к общим свойствам насекомого шмель имеет собственные уникальные свойства, такие как способность жалить

В добавление к общим свойствам насекомого кузнечик имеет собственные уникальные свойства, такие как способность прыгать

РИС. 11.1. Шмели и кузнечики являются конкретизированными вариантами насекомого

Наследование и отношение "род — вид"

Когда один объект является конкретизированным вариантом другого объекта, между ними существует отношение "род — вид", или отношение классификации. Например, кузнечик является видом насекомого. Вот несколько других примеров, отношения "род — вид":

- ◆ автомобиль — это вид автотранспортного средства;
- ◆ цветок — это вид растения;

- ◆ прямоугольник — это вид геометрической фигуры;
- ◆ футболист — это вид спортсмена.

Когда между объектами существует отношение "род — вид", это означает, что конкретизированный объект имеет все свойства родового объекта плюс дополнительные свойства, которые делают его особенным. В объектно-ориентированном программировании наследование используется для создания отношения "род — вид" среди классов. Оно позволяет расширять возможности класса путем создания другого класса, который является его конкретизированным вариантом.

Наследование связано с существованием надкласса и подкласса. *Надкласс* — это родовой класс, *подкласс* — это конкретизированный класс. Подкласс можно представить, как расширенный вариант надкласса. Подкласс наследует атрибуты и методы надкласса без необходимости переписывать его атрибуты и методы. Кроме того, в подкласс могут быть добавлены новые атрибуты и методы, и именно это делает его конкретизированным вариантом надкласса.



ПРИМЕЧАНИЕ

Надклассы также называются *базовыми классами*, а подклассы — *производными классами*. Обе версии терминов применимы. Для единобразия в этой книге используются термины "надкласс" и "подкласс".

Обратимся к примеру использования наследования. Предположим, что мы разрабатываем программу, которую автодилер может использовать для управления своими складскими запасами подержанных автомашин. Складские ресурсы автодилера включают три типа автомашин: легковые автомобили, пикапы и джипы. Независимо от типа автомобиля автодилер о каждом автомобиле поддерживает приведенные ниже данные:

- ◆ фирма-изготовитель;
- ◆ модель указанного года выпуска;
- ◆ пробег;
- ◆ цена.

Каждый вид автотранспортного средства, который содержится среди складских запасов, имеет эти общие свойства плюс свои специализированные свойства. Для легковых автомобилей автодилер поддерживает следующие дополнительные данные: число дверей (2 или 4).

Для пикапов автодилер поддерживает следующие дополнительные данные: тип привода (моноприводный, т. е. с приводом на два колеса, или полноприводный, т. е. с приводом на четыре колеса).

И для джипов автодилер поддерживает следующие дополнительные данные: пассажирская вместимость.

При разработке этой программы один из подходов состоит в написании трех классов:

- ◆ класса *Car* (Легковой автомобиль) с атрибутами данных для фирмы-изготовителя, модели указанного года выпуска, пробега, цены и количества дверей;
- ◆ класса *Truck* (Пикап) с атрибутами данных для фирмы-изготовителя, модели указанного года выпуска, пробега, цены и типа привода;
- ◆ класса *SUV* (Джип) с атрибутами данных для фирмы-изготовителя, модели указанного года выпуска, пробега, цены и пассажирской вместимостью.

Однако такой подход был бы неэффективным, потому что все три класса имеют большое количество общих атрибутов данных. В результате классы содержали бы много повторяющегося программного кода. И если позже обнаружится, что нужно добавить дополнительные общие атрибуты, то придется видоизменить все три класса.

Оптимальный подход состоит в написании надкласса `Automobile`, который будет содержать все общие данные об автомобиле, и затем в написании подклассов для каждого конкретного вида автомобиля. В программе 11.1 приведен код класса `Automobile`, который содержится в модуле `vehicles` (автотранспортные средства).

Программа 11.1 (vehicles.py, строки 1–44)

```
1 # Класс Automobile содержит общие данные
2 # об автомобиле на складе.
3
4 class Automobile:
5     # Метод __init__method принимает аргументы для
6     # фирмы-изготовителя, модели, пробега и цены.
7     # Он инициализирует атрибуты данных этими значениями.
8
9     def __init__(self, make, model, mileage, price):
10         self.__make = make
11         self.__model = model
12         self.__mileage = mileage
13         self.__price = price
14
15     # Следующие ниже методы являются методами-мутаторами
16     # атрибутов данных этого класса.
17
18     def set_make(self, make):
19         self.__make = make
20
21     def set_model(self, model):
22         self.__model = model
23
24     def set_mileage(self, mileage):
25         self.__mileage = mileage
26
27     def set_price(self, price):
28         self.__price = price
29
30     # Следующие ниже методы являются методами-получателями
31     # атрибутов данных этого класса.
32
33     def get_make(self):
34         return self.__make
35
```

```
36     def get_model(self):
37         return self.__model
38
39     def get_mileage(self):
40         return self.__mileage
41
42     def get_price(self):
43         return self.__price
44
```

Метод `__init__()` класса `Automobile` принимает аргументы для фирмы-изготовителя, модели, пробега и цены автотранспортного средства. Он использует эти значения для инициализации атрибутов данных:

- ◆ `__make` (изготовитель);
- ◆ `__model` (модель);
- ◆ `__mileage` (пробег);
- ◆ `__price` (цена).

(Из главы 10 известно, что атрибут данных становится скрытым, когда он начинается с двух символов подчеркивания.) Методы в строках 18–28 являются методами-мутаторами для каждого атрибута данных, а методы в строках 33–43 — методами-получателями.

Класс `Automobile` является законченным классом, из которого можно создавать объекты. Если есть необходимость, можно написать программу, которая импортирует модуль `vehicles` и создает экземпляры класса `Automobile`. Однако класс `Automobile` содержит только общие данные об автомобиле. В нем нет ни одной конкретной порции данных, которые автодилер желает поддерживать о легковых автомобилях, пикапах и джипах. Для того чтобы включить данные об этих конкретных видах автомобилей, мы напишем подклассы, которые наследуют от класса `Automobile`. В программе 11.2 приведен соответствующий программный код для класса `Car`, который тоже находится в модуле `vehicles`.

Программа 11.2 (vehicles.py, строки 45–72)

```
45 # Класс Car представляет легковой автомобиль.
46 # Он является подклассом класса Automobile.
47
48 class Car(Automobile):
49     # Метод __init__ принимает аргументы для фирмы-изготовителя,
50     # модели, пробега, цены и количества дверей.
51
52     def __init__(self, make, model, mileage, price, doors):
53         # Вызвать метод __init__ надкласса и передать
54         # требуемые аргументы. Обратите внимание, что мы также
55         # передаем self в качестве аргумента.
56         Automobile.__init__(self, make, model, mileage, price)
57
```

```

58     # Инициализировать атрибут __doors.
59     self.__doors = doors
60
61     # Метод set_doors является методом-мутатором
62     # атрибута __doors.
63
64     def set_doors(self, doors):
65         self.__doors = doors
66
67     # Метод get_doors является методом-получателем
68     # атрибута __doors.
69
70     def get_doors(self):
71         return self.__doors
72

```

Приглядитесь к инструкции объявления класса в строке 48:

```
class Car(Automobile):
```

Эта строка говорит о том, что мы определяем класс под названием `Car`, и он наследует от класса `Automobile`. Класс `Car` является подклассом, а класс `Automobile` — надклассом. Если нужно выразить связь между классом `Car` и классом `Automobile`, то можно сказать, что легковой автомобиль `Car` является видом родового автомобиля `Automobile`. Поскольку класс `Car` расширяет класс `Automobile`, он наследует все методы и атрибуты данных класса `Automobile`.

Взгляните на заголовок метода `__init__()` в строке 52:

```
def __init__(self, make, model, mileage, price, doors):
```

Обратите внимание, что в дополнение к требуемому параметру `self` у метода есть параметры `make`, `model`, `mileage`, `price` и `doors`. Это разумно, потому что объект `Car` будет иметь атрибуты данных для фирмы-изготовителя, модели, пробега, цены и количества дверей легкового автомобиля. Однако некоторые из этих атрибутов создаются классом `Automobile`, поэтому нам нужно вызвать метод `__init__()` класса `Automobile` и передать в него эти значения. Это происходит в строке 56:

```
Automobile.__init__(self, make, model, mileage, price)
```

Эта инструкция вызывает метод `__init__()` класса `Automobile`. Обратите внимание, что инструкция передает переменную `self`, а также переменные `make`, `model`, `mileage` и `price` в качестве аргументов. Когда этот метод исполняется, он инициализирует атрибуты данных переменными `__make`, `__model`, `__mileage` и `__price`. Затем в строке 59 атрибут `__doors` инициализируется значением, переданным в параметр `doors`:

```
self.__doors = doors
```

Метод `set_doors()` в строках 64–65 является методом-мутатором атрибута `__doors`, а метод `get_doors()` в строках 70–71 — методом-получателем атрибута `__doors`. Прежде чем пойти дальше, рассмотрим класс `Car`, который приведен в программе 11.3.

Программа 11.3 (car_demo.py)

```

1 # Эта программа демонстрирует класс Car.
2
3 import vehicles
4
5 def main():
6     # Создать объект на основе класса Car.
7     # Легковое авто: 2007 Audi с 12500 милями пробега,
8     # ценой $21500.00 и с 4 дверьми.
9     used_car = vehicles.Car('Audi', 2007, 12500, 21500.0, 4)
10
11     # Показать данные легкового авто.
12     print('Изготовитель:', used_car.get_make())
13     print('Модель:', used_car.get_model())
14     print('Пробег:', used_car.get_mileage())
15     print('Цена:', used_car.get_price())
16     print('Количество дверей:', used_car.get_doors())
17
18 # Вызвать главную функцию.
19 if __name__ == '__main__':
20     main()

```

Вывод программы

Изготовитель: Audi

Модель: 2007

Пробег: 12500

Цена: 21500.0

Количество дверей: 4

Строка 3 импортирует модуль `vehicles`, который содержит определения классов `Automobile` и `Car`. Стока 9 создает экземпляр класса `Car`, передавая 'Audi' в качестве названия фирмы-изготовителя легкового автомобиля, 2007 в качестве модели легкового автомобиля, 12500 в качестве пробега, 21500.0 в качестве цены легкового автомобиля и 4 в качестве количества дверей. Полученный объект присваивается переменной `used_car` (поддержанное авто).

Инструкции в строках 12–15 вызывают методы `get_make()`, `get_model()`, `get_mileage()` и `get_price()` этого объекта. Несмотря на то что класс `Car` не имеет ни одного из этих методов, он их наследует от класса `Automobile`. Стока 16 вызывает метод `get_doors()`, который определен в классе `Car`.

Теперь давайте рассмотрим класс `Truck`, который тоже наследует от класса `Automobile`. Код для класса `Truck`, который также находится в модуле `vehicles`, приведен в программе 11.4.

Программа 11.4 (vehicles.py, строки 73–100)

```

73 # Класс Truck представляет пикап.
74 # Он является подклассом класса Automobile.
75

```

```

76 class Truck(Automobile):
77     # Метод __init__ принимает аргументы для
78     # изготовителя, модели, пробега, цены и типа привода пикапа.
79
80     def __init__(self, make, model, mileage, price, drive_type):
81         # Вызывать метод __init__ надкласса и передать
82         # требуемые аргументы. Обратите внимание, что мы также
83         # передаем self в качестве аргумента.
84         Automobile.__init__(self, make, model, mileage, price)
85
86         # Инициализировать атрибут __drive_type.
87         self.__drive_type = drive_type
88
89     # Метод set_drive_type является методом-мутатором
90     # атрибута __drive_type.
91
92     def set_drive_type(self, drive_type):
93         self.__drive = drive_type
94
95     # Метод get_drive_type является методом-получателем
96     # атрибута __drive_type.
97
98     def get_drive_type(self):
99         return self.__drive_type
100

```

Метод `__init__()` класса `Truck` начинается в строке 80. Он принимает аргументы для изготовителя, модели, пробега, цены и типа привода пикапа. Так же как класс `Car`, класс `Truck` вызывает метод `__init__()` класса `Automobile` (в строке 84), передавая название фирмы-изготовителя, модель, пробег и цену в качестве аргументов. Стока 87 создает атрибут `__drive_type`, инициализируя его значением параметра `drive_type`.

Метод `set_drive_type()` в строках 92–93 является методом-мутатором атрибута `__drive_type`, а метод `get_drive_type()` в строках 98–99 — методом-получателем этого атрибута.

Теперь рассмотрим класс `SUV`, который тоже наследует класс `Automobile`. Код для класса `SUV`, который находится в модуле `vehicles`, представлен в программе 11.5.

Программа 11.5 (vehicles.py, строки 101–128)

```

101 # Класс SUV представляет джип.
102 # Он является подклассом класса Automobile.
103
104 class SUV(Automobile):
105     # Метод __init__ принимает аргументы для
106     # изготовителя, модели, пробега, цены
107     # и пассажирской вместимости.
108

```

```
109     def __init__(self, make, model, mileage, price, pass_cap):  
110         # Вызвать метод __init__ надкласса и передать  
111         # требуемые аргументы. Обратите внимание, что мы также  
112         # передаем self в качестве аргумента.  
113         Automobile.__init__(self, make, model, mileage, price)  
114  
115         # Инициализировать атрибут __pass_cap.  
116         self.__pass_cap = pass_cap  
117  
118     # Метод set_pass_cap является методом-мутатором  
119     # атрибута __pass_cap.  
120  
121     def set_pass_cap(self, pass_cap):  
122         self.__pass_cap = pass_cap  
123  
124     # Метод get_pass_cap является методом-получателем  
125     # атрибута __pass_cap.  
126  
127     def get_pass_cap(self):  
128         return self.__pass_cap
```

Метод `__init__()` класса `SUV` начинается в строке 109. Он принимает аргументы для изгото-
вителя, модели, пробега, цены и пассажирской вместимости. Так же как классы `Car` и `Truck`,
класс `SUV` вызывает метод `__init__()` класса `Automobile` (в строке 113), передавая
название фирмы-изготовителя, модель, пробег и цену в качестве аргументов. Стока 116
создает атрибут `__pass_cap`, инициализируя его значением параметра `pass_cap`.

Метод `set_pass_cap()` в строках 121–122 является методом-мутатором атрибута `__pass_cap`,
а метод `get_pass_cap()` в строках 127–128 — методом-получателем этого атрибута.

Программа 11.6 демонстрирует каждый из классов, которые мы обсудили до сих пор. Она
создает объекты `Car`, `Truck` и `SUV`.

Программа 11.6 (car_truck_suv_demo.py)

```
1 # Эта программа создает объекты Car, Truck  
2 # и SUV.  
3  
4 import vehicles  
5  
6 def main():  
7     # Создать объект Car для подержанного авто 2001 BMW  
8     # с 70000 милями пробега, ценой $15000,  
9     # с 4 дверьми.  
10    car = vehicles.Car('BMW', 2001, 70000, 15000.0, 4)  
11
```

```

12  # Создать объект Truck для подержанного пикапа 2002
13  # Toyota с 40000 милями пробега, ценой
14  # $12000 и с 4-колесным приводом.
15  truck = vehicles.Truck('Toyota', 2002, 40000, 12000.0, '4WD')
16
17  # Создать объект SUV для подержанного 2000
18  # Volvo с 30000 милями пробега, ценой
19  # $18500 и вместимостью 5 человек.
20  suv = vehicles.SUV('Volvo', 2000, 30000, 18500.0, 5)
21
22  print('ПОДДЕРЖАННЫЕ АВТО НА СКЛАДЕ')
23  print('=====')
24
25  # Показать данные легкового авто.
26  print('Данный легковой автомобиль имеется на складе.')
27  print('Изготовитель:', car.get_make())
28  print('Модель:', car.get_model())
29  print('Пробег:', car.get_mileage())
30  print('Цена:', car.get_price())
31  print('Количество дверей:', car.get_doors())
32  print()
33
34  # Показать данные пикапа.
35  print('Данный пикап имеется на складе.')
36  print('Изготовитель:', truck.get_make())
37  print('Модель:', truck.get_model())
38  print('Пробег:', truck.get_mileage())
39  print('Цена:', truck.get_price())
40  print('Тип привода:', truck.get_drive_type())
41  print()
42
43  # Показать данные джипа.
44  print('Данный джип имеется на складе.')
45  print('Изготовитель:', suv.get_make())
46  print('Модель:', suv.get_model())
47  print('Пробег:', suv.get_mileage())
48  print('Цена:', suv.get_price())
49  print('Пассажирская вместимость:', suv.get_pass_cap())
50
51 # Вызвать главную функцию.
52 if __name__ == '__main__':
53     main()

```

Вывод программы

ПОДДЕРЖАННЫЕ АВТО НА СКЛАДЕ

Данный легковой автомобиль имеется на складе.

Изготовитель: BMW

Модель: 2001

Пробег: 70000
 Цена: 15000.0
 Количество дверей: 4

Данный пикап имеется на складе.

Изготовитель: Toyota
 Модель: 2002
 Пробег: 40000
 Цена: 12000.0
 Тип привода: 4WD

Данный джип имеется на складе.

Изготовитель: Volvo
 Модель: 2000
 Пробег: 30000
 Цена: 18500.0
 Пассажирская вместимость: 5

Наследование в диаграммах UML

В диаграмме UML наследование обозначается путем нанесения линии с пустым указателем стрелки от подкласса к надклассу. (Указатель стрелки показывает на надкласс.) На рис. 11.2

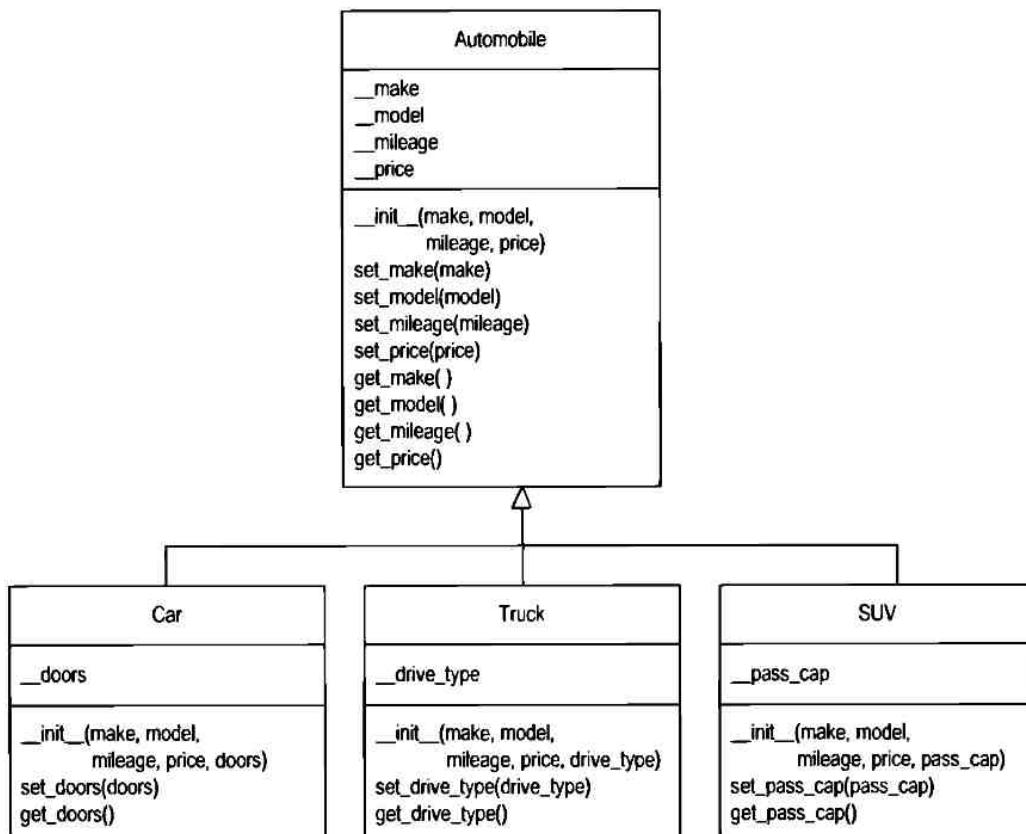


РИС. 11.2. Диаграмма UML с демонстрацией наследования

приведена диаграмма UML, указывающая на связь между классами `Automobile`, `Car`, `Truck` и `SUV`.



В ЦЕНТРЕ ВНИМАНИЯ

Использование наследования

Компания "Банковские финансовые системы" разрабатывает финансовое программное обеспечение для банков и кредитных союзов. Компания создает новую объектно-ориентированную систему, которая управляет клиентскими банковскими счетами. Одна из ваших задач состоит в том, чтобы спроектировать класс, который представляет сберегательный счет. Вот данные, которые должен содержать объект этого класса:

- ◆ номер счета;
- ◆ процентная ставка;
- ◆ остаток счета.

Вам также необходимо разработать класс, который представляет счет депозитного сертификата (CD, certificate of deposit). Вот данные, которые должен содержать объект этого класса:

- ◆ номер счета;
- ◆ процентная ставка;
- ◆ остаток счета;
- ◆ дата погашения счета.

Анализируя это техническое задание, вы понимаете, что счет депозитного сертификата (CD) является конкретизированным вариантом сберегательного счета. Класс, который представляет счет депозитного сертификата, будет содержать все те же данные, что и класс, который представляет сберегательный счет, плюс дополнительный атрибут для даты погашения. Вы решаете разработать класс `SavingsAccount` для представления сберегательного счета и подкласс `SavingsAccount` с именем `CD` для представления счета депозитного сертификата. Вы будете хранить оба этих класса в модуле `accounts` (счета). В программе 11.7 приведен код для класса `SavingsAccount`.

Программа 11.7 (accounts.py, строки 1–37)

```

1 # Класс SavingsAccount представляет
2 # сберегательный счет.
3
4 class SavingsAccount:
5
6     # Метод __init__ принимает аргументы для
7     # номера счета, процентной ставки и остатка.
8
9     def __init__(self, account_num, int_rate, bal):
10         self.__account_num = account_num
11         self.__interest_rate = int_rate
12         self.__balance = bal
13

```

```
14 # Следующие ниже методы являются методами-мутаторами
15 # атрибутов данных.
16
17 def set_account_num(self, account_num):
18     self.__account_num = account_num
19
20 def set_interest_rate(self, int_rate):
21     self.__interest_rate = int_rate
22
23 def set_balance(self, bal):
24     self.__balance = bal
25
26 # Следующие ниже методы являются методами-получателями
27 # атрибутов данных.
28
29 def get_account_num(self):
30     return self.__account_num
31
32 def get_interest_rate(self):
33     return self.__interest_rate
34
35 def get_balance(self):
36     return self.__balance
37
```

Метод `__init__()` этого класса размещается в строках 9–12. Он принимает аргументы для номера счета, процентной ставки и остатка. Эти аргументы используются для инициализации атрибутов данных с именами `__account_num`, `__interest_rate` и `__balance`.

Методы `set_account_num()` (задать номер счета), `set_interest_rate()` (задать процентную ставку) и `set_balance()` (задать остаток счета), расположенные в строках 17–24, являются методами-мутаторами атрибутов данных. Методы `get_account_num()` (получить номер счета), `get_interest_rate()` (получить процентную ставку) и `get_balance()` (получить остаток счета), расположенные в строках 29–36, являются методами-получателями.

Класс `CD` представлен в следующей части программы 11.7.

Программа 11.7 (accounts.py, строки 38–65)

```
38 # Класс CD представляет счет
39 # депозитного сертификата (CD).
40 # Это подкласс класса SavingsAccount.
41
42 class CD(SavingsAccount):
43
44     # Метод __init__ принимает аргументы для
45     # номера счета, процентной ставки, остатка
46     # и даты погашения.
47
```

```

48     def __init__(self, account_num, int_rate, bal, mat_date):
49         # Вызвать метод __init__ надкласса.
50         SavingsAccount.__init__(self, account_num, int_rate, bal)
51
52         # Инициализировать атрибут __maturity_date.
53         self.__maturity_date = mat_date
54
55     # Метод set_maturity_date является методом-мутатором
56     # атрибута __maturity_date.
57
58     def set_maturity_date(self, mat_date):
59         self.__maturity_date = mat_date
60
61     # Метод get_maturity_date является методом-получателем
62     # атрибута __maturity_date.
63
64     def get_maturity_date(self):
65         return self.__maturity_date

```

Метод `__init__()` класса `CD` размещен в строках 48–53. Он принимает аргументы для номера счета, процентной ставки, остатка и даты погашения. Стока 50 вызывает метод `__init__()` класса `SavingsAccount`, передавая аргументы для номера счета, процентной ставки и остатка. После исполнения метода `__init__()` класса `SavingsAccount` будут созданы и инициализированы атрибуты `__account_num`, `__interest_rate` и `__balance`. Затем инструкция в строке 53 создает атрибут `__maturity_date`.

Метод `set_maturity_date()` (задать дату погашения) в строках 58–59 является методом-мутатором атрибута `__maturity_date`, а метод `get_maturity_date()` (получить дату погашения) в строках 64–65 — методом-получателем этого атрибута.

Для того чтобы протестировать эти классы, мы применим код из программы 11.8. Она создает экземпляр класса `SavingsAccount` для представления сберегательного счета и экземпляр класса `CD` для представления счета депозитного сертификата.

Программа 11.8 (account_demo.py)

```

1 # Эта программа создает экземпляр класса SavingsAccount
2 # и экземпляр класса CD.
3
4 import accounts
5
6 def main():
7     # Получить номер счета, процентную ставку,
8     # и остаток сберегательного счета.
9     print('Введите данные о сберегательном счете.')
10    acct_num = input('Номер счета: ')
11    int_rate = float(input('Процентная ставка: '))
12    balance = float(input('Остаток: '))
13

```

```
14 # Создать объект SavingsAccount.  
15 savings = accounts.SavingsAccount(acct_num, int_rate,  
16                                     balance)  
17  
18 # Получить номер счета, процентную ставку,  
19 # остаток счета и дату погашения счета CD.  
20 print('Введите данные о счете CD.')21 acct_num = input('Номер счета: ')  
22 int_rate = float(input('Процентная ставка: '))  
23 balance = float(input('Остаток: '))  
24 maturity = input('Дата погашения: ')  
25  
26 # Создать объект CD.  
27 cd = accounts.CD(acct_num, int_rate, balance, maturity)  
28  
29 # Показать введенные данные.  
30 print('Вот введенные Вами данные: ')  
31 print()  
32 print('Сберегательный счет')  
33 print('-----')  
34 print(f'Номер счета: {savings.get_account_num()}')  
35 print(f'Процентная ставка: {savings.get_interest_rate()}')  
36 print(f'Остаток: ${savings.get_balance():,.2f}')  
37 print()  
38 print('Счет депозитного сертификата (CD)')  
39 print('-----')  
40 print(f'Номер счета: {cd.get_account_num()}')  
41 print(f'Процентная ставка: {cd.get_interest_rate()}')  
42 print(f'Остаток: ${cd.get_balance():,.2f}')  
43 print(f'Дата погашения: {cd.get_maturity_date()}')  
44  
45 # Вызвать главную функцию.  
46 if __name__ == '__main__':  
47     main()
```

Вывод программы (вводимые данные выделены жирным шрифтом)

Введите данные о сберегательном счете.

Номер счета: **1234SA**

Процентная ставка: **3.5**

Остаток: **1000.00**

Введите данные о счете CD.

Номер счета: **2345CD**

Процентная ставка: **5.6**

Остаток: **2500.00**

Дата погашения: **12/12/2024**

Вот введенные Вами данные:

Сберегательный счет

Номер счета: 1234SA

Процентная ставка: 3.5

Остаток: \$1,000.00

Счет депозитного сертификата (CD)

Номер счета: 2345CD

Процентная ставка: 5.6

Остаток: \$2,500.00

Дата погашения: 12/12/2024



Контрольная точка

- 11.1. В этом разделе мы рассмотрели надклассы и подклассы. Какой из них является общим классом, а какой конкретизированным классом?
- 11.2. Что значит, когда говорят, что между двумя объектами существует отношение классификации, т. е. отношение "род — вид"?
- 11.3. Что подкласс наследует от своего надкласса?
- 11.4. Взгляните на приведенный ниже фрагмент кода, который представляет собой первую строку определения класса. Как называется надкласс? Как называется подкласс (здесь используются классы "Канарейка" и "Птица")?

```
class Canary(Bird):
```

11.2

Полиморфизм

Ключевые положения

Полиморфизм позволяет подклассам иметь методы с теми же именами, что и у методов в их надклассах. Это дает программе возможность вызывать правильный метод в зависимости от типа объекта, который используется для его вызова.

Термин "*полиморфизм*" относится к способности объекта принимать различные формы. Эта способность является мощным механизмом объектно-ориентированного программирования. В данном разделе мы рассмотрим два важнейших компонента полиморфного поведения.

- ◆ Возможность определять метод в надклассе и затем определять метод с тем же именем в подклассе. Когда метод подкласса имеет то же имя, что и у метода надкласса, часто говорят, что метод подкласса *переопределяет* метод надкласса.
- ◆ Возможность вызывать правильный вариант переопределенного метода в зависимости от типа объекта, который используется для его вызова. Если объект подкласса используется

для вызова переопределенного метода, то исполнится именно вариант метода подкласса. Если для вызова переопределенного метода используется объект надкласса, то исполнится именно вариант метода надкласса.

Вы уже видели переопределение метода в работе. Все подклассы, исследованные нами в этой главе, имеют метод `__init__()`, который переопределяет метод `__init__()` надкласса. Когда создается экземпляр подкласса, автоматически вызывается метод `__init__()` именно этого подкласса.

Переопределение методов работает и для других методов класса. Возможно, лучший способ описать полиморфизм состоит в том, чтобы его продемонстрировать, поэтому рассмотрим простой пример. В программе 11.9 приведен код для класса `Mammal` (Млекопитающее), который находится в модуле `animals` (животные).

Программа 11.9 (animals.py, строки 1–22)

```
1 # Класс Mammal представляет род млекопитающих.
2
3 class Mammal:
4
5     # Метод __init__ принимает аргумент для
6     # вида млекопитающего.
7
8     def __init__(self, species):
9         self.__species = species
10
11    # Метод show_species выводит сообщение
12    # о виде млекопитающего.
13
14    def show_species(self):
15        print('Я -', self.__species)
16
17    # Метод make_sound издает характерный
18    # для всех млекопитающих звук.
19
20    def make_sound(self):
21        print('Гррррррр')
22
```

Класс `Mammal` имеет три метода: `__init__()`, `show_species()` (Показать вид млекопитающего) и `make_sound()` (Издать звук). Вот пример кода, который создает экземпляр класса и вызывает эти методы:

```
import animals
mammal = animals.Mammal('обычное млекопитающее')
mammal.show_species()
mammal.make_sound()
```

Этот фрагмент кода покажет приведенный ниже результат:

Я – обычное млекопитающее

Грррррр

Следующая часть программы 11.9 показывает класс Dog (Собака), который тоже находится в модуле `animals` и является подклассом класса `Mammal`.

Программа 11.9 (animals.py, строки 23–38)

```

23 # Класс Dog является подклассом класса Mammal.
24
25 class Dog(Mammal):
26
27     # Метод __init__ вызывает метод __init__
28     # надкласса, передавая 'собака' в качестве вида.
29
30     def __init__(self):
31         Mammal.__init__(self, 'собака')
32
33     # Метод make_sound переопределяет
34     # метод make_sound надкласса.
35
36     def make_sound(self):
37         print('Гав-гав!')
38

```

Хотя класс `Dog` наследует методы `__init__()` и `make_sound()`, которые находятся в классе `Mammal`, эти методы не отвечают требованиям класса `Dog`. И поэтому класс `Dog` имеет собственные методы `__init__()` и `make_sound()`, выполняющие действия, которые более подходят для собаки. Мы говорим, что методы `__init__()` и `make_sound()` в классе `Dog` переопределяют методы `__init__()` и `make_sound()` в классе `Mammal`. Вот пример фрагмента кода, который создает экземпляр класса `Dog` и вызывает эти методы:

```

import animals
dog = animals.Dog()
dog.show_species()
dog.make_sound()

```

Этот фрагмент кода покажет приведенный ниже результат:

Я – собака

Гав-гав!

Когда мы используем объект `Dog`, чтобы вызвать методы `show_species()` и `make_sound()`, исполняются именно те версии этих методов, которые находятся в классе `Dog`. А теперь взгляните на программу 11.10, демонстрирующую класс `Cat` (Кот), который тоже находится в модуле `animals` и является еще одним подклассом класса `Mammal`.

Программа 11.9 (animals.py, строки 39–53)

```

39 # Класс Cat является подклассом класса Mammal.
40
41 class Cat(Mammal):
42
43     # Метод __init__ вызывает метод __init__
44     # надкласса, передавая 'кот' в качестве вида.
45
46     def __init__(self):
47         Mammal.__init__(self, 'кот')
48
49     # Метод make_sound переопределяет
50     # метод make_sound надкласса.
51
52     def make_sound(self):
53         print('Мяу!')

```

Класс Cat тоже переопределяет методы `__init__()` и `make_sound()` класса Mammal. Вот пример фрагмента кода, который создает экземпляр класса Cat и вызывает эти методы:

```

import animals
cat = animals.Cat()
cat.show_species()
cat.make_sound()

```

Этот фрагмент кода покажет следующее:

```

Я – кот
Мяу!

```

Когда мы используем объект Cat, чтобы вызвать методы `show_species()` и `make_sound()`, исполняются именно те версии этих методов, которые находятся в классе Cat.

Функция `isinstance`

Полиморфизм обеспечивает большую гибкость при разработке программ. Например, взгляните на приведенную ниже функцию:

```

def show_mammal_info(creature):
    creature.show_species()
    creature.make_sound()

```

В эту функцию в качестве аргумента можно передать любой объект, и раз она имеет методы `show_species()` и `make_sound()`, то вызовет эти методы. В сущности, в эту функцию можно передать любой объект, который является "видом" млекопитающего Mammal (или подклассом Mammal). Программа 11.10 это демонстрирует.

Программа 11.10 (polymorphism_demo.py)

```

1 # Эта программа демонстрирует полиморфизм.
2
3 import animals
4

```

```

5 def main():
6     # Создать объект Mammal, объект Dog
7     # и объект Cat.
8     mammal = animals.Mammal('обычное животное')
9     dog = animals.Dog()
10    cat = animals.Cat()
11
12    # Показать информацию о каждом животном.
13    print('Вот несколько животных и')
14    print('звуки, которые они издают.')
15    print('-----')
16    show_mammal_info(mammal)
17    print()
18    show_mammal_info(dog)
19    print()
20    show_mammal_info(cat)
21
22 # Функция show_mammal_info принимает объект
23 # в качестве аргумента и вызывает свои методы
24 # show_species и make_sound.
25
26 def show_mammal_info(creature):
27     creature.show_species()
28     creature.make_sound()
29
30 # Вызвать главную функцию.
31 if __name__ == '__main__':
32     main()

```

Вывод программы

Вот несколько животных и
звуки, которые они издают.

Я - обычное животное

Гrrrrrrr

Я - собака

Гав-гав!

Я - кот

Мяу!

Но что произойдет, если в эту функцию передать объект, который не является млекопитающим Mammal и не является подклассом Mammal? Например, что случится, когда будет выполниться программа 11.11?

Программа 11.11 (wrong_type.py)

```
1 def main():
2     # Передать символьное значение в функцию show_mammal_info
3     show_mammal_info('Я - последовательность символов')
4
5 # Функция show_mammal_info принимает объект
6 # в качестве аргумента и вызывает свои методы
7 # show_species и make_sound.
8
9 def show_mammal_info(creature):
10    creature.show_species()
11    creature.make_sound()
12
13 # Вызвать главную функцию.
14 if __name__ == '__main__':
15     main()
```

В строке 3 мы вызываем функцию `show_mammal_info`, передавая строковое значение в качестве аргумента. Однако когда интерпретатор попытается исполнить строку 10, будет вызвано исключение `AttributeError` (Ошибка атрибута), потому что строковый тип не имеет метода под названием `show_species()`.

Наступление этого исключения можно предотвратить при помощи встроенной функции `isinstance`. Ее применяют с целью определения, является ли объект экземпляром конкретного класса или подклассом этого класса. Вот общий формат вызова этой функции:

```
isinstance(объект, класс)
```

В данном формате `объект` — это ссылка на объект, `класс` — это имя класса. Если объект, на который ссылается `объект`, является экземпляром `класса` или экземпляром `подкласса`, то эта функция возвращает истину. В противном случае она возвращает ложь. В программе 11.12 показан способ ее применения в функции `show_mammal_info`.

Программа 11.12 (polymorphism_demo2.py)

```
1 # Эта программа демонстрирует полиморфизм.
2
3 import animals
4
5 def main():
6     # Создать объект Mammal, объект Dog
7     # и объект Cat.
8     mammal = animals.Mammal('обычное животное')
9     dog = animals.Dog()
10    cat = animals.Cat()
11
12    # Показать информацию о каждом животном.
13    print('Вот несколько животных и')
```

```

14     print('звуки, которые они издают.')
15     print('-----')
16     show_mammal_info(mammal)
17     print()
18     show_mammal_info(dog)
19     print()
20     show_mammal_info(cat)
21     print()
22     show_mammal_info('Я – последовательность символов')
23
24 # Функция show_mammal_info принимает объект
25 # в качестве аргумента и вызывает свои методы
26 # show_species и make_sound.
27
28 def show_mammal_info(creature):
29     if isinstance(creature, animals.Mammal):
30         creature.show_species()
31         creature.make_sound()
32     else:
33         print('Это не млекопитающее!')
34
35 # Вызвать главную функцию.
36 if __name__ == '__main__':
37     main()

```

Вывод программы

Вот несколько животных и
звуки, которые они издают.

Я – обычное животное

Гrrrrrrr

Я – собака

Гав-гав!

Я – кот

Мяу!

Это не млекопитающее!

В строках 16, 18 и 20 мы вызываем функцию `show_mammal_info`, передавая ссылки на объект `Mammal`, объект `Dog` и объект `Cat`. Однако в строке 22 мы вызываем функцию и передаем в качестве аргумента строковое значение. В функции `show_mammal_info` инструкция `if` в строке 29 вызывает функцию `isinstance`, чтобы определить, является ли аргумент экземпляром `Mammal` (или его подклассом). Если нет, то выводится сообщение об ошибке.



Контрольная точка

11.5. Взгляните на приведенные ниже определения классов:

```
class Vegetable:
    def __init__(self, vegtype):
        self.vegtype = vegtype
    def message(self):
        print("Я - овощ.")

class Potato(Vegetable):
    def __init__(self):
        Vegetable.__init__(self, 'картофель')
    def message(self):
        print("Я - картофель.")
```

Что покажут приведенные ниже инструкции с учетом этих определений классов?

```
v = Vegetable('овощной продукт')
p = Potato()
v.message()
p.message()
```

Вопросы для повторения

Множественный выбор

- В отношении наследования _____ является родовым классом.
 - подкласс;
 - надкласс;
 - ведомый класс;
 - дочерний класс.
- В отношении наследования _____ является конкретизированным, или видовым, классом.
 - надкласс;
 - ведущий класс;
 - подкласс;
 - родительский класс.
- Предположим, что в программе используются два класса: *Airplane* (Самолет) и *JumboJet* (Аэробус). Какой из них, скорее всего, будет подклассом?
 - самолет;
 - аэробус;
 - оба;
 - ни один.

4. Этот механизм объектно-ориентированного программирования позволяет вызывать правильный вариант переопределенного метода, когда для его вызова используется экземпляр подкласса.
 - а) полиморфизм;
 - б) наследование;
 - в) обобщение;
 - г) конкретизация.
5. Один из приведенных ниже вариантов применяется для определения, является ли объект экземпляром класса.
 - а) оператор `in`;
 - б) функция `is_object_of`;
 - в) функция `isinstance`;
 - г) сообщения об ошибках, выводимые, когда программа аварийно завершается.

Истина или ложь

1. Полиморфизм позволяет писать методы в подклассе, которые имеют то же имя, что и у методов в надклассе.
2. Вызвать метод `__init__()` надкласса из метода `__init__()` подкласса невозможно.
3. Подкласс может иметь метод с тем же именем, что и у метода в надклассе.
4. Переопределять можно только метод `__init__()`.
5. Функция `isinstance` не применяется для определения, является ли объект экземпляром подкласса некоторого класса.

Короткий ответ

1. Что подкласс наследует от своего надкласса?
2. Взгляните на приведенное ниже определение класса. Как называется надкласс? Как называется подкласс (здесь используются классы "Тигр" и "Кошачие")?
`class Tiger(Felis):`
3. Что такое переопределенный метод?

Алгоритмический тренажер

1. Напишите первую строку определения класса `Poodle` (Пудель). Этот класс должен расширять класс `Dog`.
2. Взгляните на приведенные ниже определения классов:

```
class Plant:
    def __init__(self, plant_type):
        self.__plant_type = plant_type
    def message(self):
        print("Я - планета.")
```

```
class Tree(Plant):  
    def __init__(self):  
        Plant.__init__(self, 'дерево')  
    def message(self):  
        print("Я - дерево.")
```

Что покажут приведенные ниже инструкции с учетом этих определений классов?

```
p = Plant('саженец')  
t = Tree()  
p.message()  
t.message()
```

3. Взгляните на приведенное ниже определение:

```
class Beverage:  
    def __init__(self, bev_name):  
        self._bev_name = bev_name
```

Напишите программный код для класса с именем `Cola` (Кока-кола), который является подклассом класса `Beverage` (Напиток). Метод `__init__()` класса `Cola` должен вызывать метод `__init__()` класса `Beverage`, передавая строковое значение 'кока-кола' в качестве аргумента.

Упражнения по программированию

1. Классы `Employee` и `ProductionWorker`. Напишите класс `Employee` (Сотрудник), который содержит атрибуты приведенных ниже данных:

- имя сотрудника;
- номер сотрудника.

Затем напишите класс `ProductionWorker` (Рабочий), который является подклассом класса `Employee`. Класс `ProductionWorker` должен содержать атрибуты приведенных ниже данных:

- номер смены (целое число, к примеру, 1 или 2);
- ставка почасовой оплаты труда.

Рабочий день разделен на две смены: дневную и вечернюю. Атрибут смены будет содержать целочисленное значение, представляющее смену, в которую сотрудник работает. Дневная смена является сменой 1, вечерняя смена — сменой 2. Напишите соответствующие методы-получатели и методы-мутаторы для каждого класса.

Затем напишите программу, которая создает объект класса `ProductionWorker` и предлагает пользователю ввести данные по каждому атрибуту данных этого объекта. Сохраните данные в объекте и примените в этом объекте методы-получатели, чтобы получить эти данные и вывести их на экран.

2. Класс `ShiftSupervisor`. На некой фабрике начальник смены является штатным сотрудником, который руководит сменой. В дополнение к фиксированному окладу начальник смены получает годовую премию за выполнение его сменой производственного плана. Напишите класс `ShiftSupervisor` (Начальник смены), который является подклассом

класса `Employee`, созданного в задаче по программированию 1. Класс `ShiftSupervisor` должен содержать атрибут данных для годового оклада и атрибут данных для годовой производственной премии, которую заработал начальник смены. Продемонстрируйте класс, написав программу, которая применяет объект `ShiftSupervisor`.

3. Классы `Person` и `Customer`.



Видеозапись "Классы `Person` и `Customer`" (*Person and Customer Classes*)

Напишите класс `Person` с атрибутами данных для имени, адреса и телефонного номера человека. Затем напишите класс `Customer` (Клиент), который является подклассом класса `Person`. Класс `Customer` должен иметь атрибут данных для номера клиента и атрибут булевых данных, указывающий, хочет ли клиент быть в списке рассылки или нет. Продемонстрируйте экземпляр класса `Customer` в простой программе.

12.1 Введение в рекурсию

Ключевые положения

Рекурсивная функция — это функция, которая вызывает саму себя.

Ранее вы встречались с экземплярами функций, которые вызывают другие функции. В частности, программе главная функция (`main`) вызывает функцию `A`, которая затем может вызывать функцию `B`. Однако бывают случаи, когда функция также вызывает саму себя. Такая функция называется *рекурсивной*. Взгляните на функцию `message`, которая приведена в программе 12.1.

Программа 12.1 (endless_recursion.py)

```
1 # Эта программа демонстрирует рекурсивную функцию.
2
3 def main():
4     message()
5
6 def message():
7     print('Это рекурсивная функция.')
8     message()
9
10 # Вызвать главную функцию.
11 if __name__ == '__main__':
12     main()
```

Вывод программы

```
Это рекурсивная функция.
Это рекурсивная функция.
Это рекурсивная функция.
Это рекурсивная функция.
```

... И этот результат будет повторяться бесконечно!

Функция `message` выводит на экран строковый литерал 'Это рекурсивная функция.' и затем вызывает саму себя. При каждом вызове функцией самой себя цикл повторяется. Вы заметили, в чем проблема с этой функцией? В ней не предусмотрен способ останова рекурсивных вызовов. Эта функция выглядит как бесконечный цикл, потому что отсутствует

программный код, который остановил бы ее бесконечные повторы. Если запустить эту программу, то для прерывания ее выполнения придется нажать комбинацию клавиш **<Ctrl>+<C>**.

Подобно циклу, рекурсивная функция должна иметь какой-то способ управлять количеством своих повторов. В программе 12.2 приведена видоизмененная версия функции `message`. В ней функция `message` получает аргумент, который задает количество раз, которые функция должна выводить сообщение.

Программа 12.2 (recursive.py)

```

1 # Эта программа имеет рекурсивную функцию.
2
3 def main():
4     # Передав аргумент 5 в функцию message,
5     # мы сообщаем ей, что нужно показать
6     # сообщение пять раз.
7     message(5)
8
9 def message(times):
10    if times > 0:
11        print('Это рекурсивная функция.')
12        message(times - 1)
13
14 # Вызвать главную функцию.
15 if __name__ == '__main__':
16     main()

```

Вывод программы

```

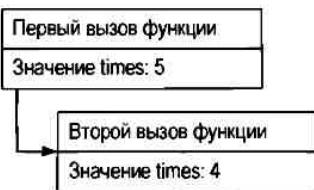
Это рекурсивная функция.

```

В строке 10 этой программы функция `message` содержит инструкцию `if`, которая управляет повторением. Сообщение 'Это рекурсивная функция.' будет выводиться до тех пор, пока параметр `times` больше нуля, при этом функция будет вызывать саму себя повторно, передавая уменьшенный аргумент.

В строке 7 главная функция вызывает функцию `message`, передавая аргумент 5. Во время первого вызова функции инструкция `if` выводит сообщение, и затем функция вызывает саму себя, передавая в качестве аргумента 4. На рис. 12.1 проиллюстрирован этот процесс.

Схема на рис. 12.1 показывает два отдельных вызова функции `message`. Во время каждого вызова функции в оперативной памяти создается новый экземпляр параметра `times`. При первом вызове функции параметр `times` имеет значение 5. Когда функция себя вызывает, создается новый экземпляр параметра `times`, и в него передается значение 4. Этот цикл повторяется до тех пор, пока наконец в функцию в качестве аргумента не будет передан 0 (рис. 12.2).



В первый раз эта функция вызывается из главной функции
Вызовы со второго по шестой являются рекурсивными

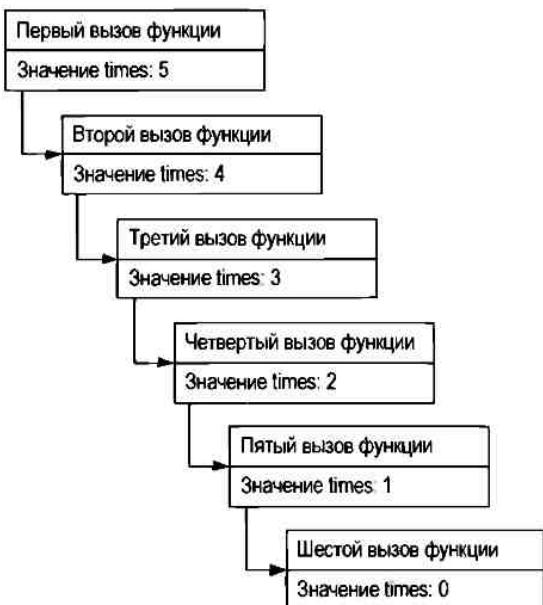


РИС. 12.1. Первые два вызова функции

РИС. 12.2. Шесть вызовов функции message

Как видно из рисунка, функция вызывается шесть раз. В первый раз она вызывается из главной функции (`main`), а остальные пять раз она вызывает саму себя. Количество раз, которые функция вызывает саму себя, называется *глубиной рекурсии*. В этом примере глубина рекурсии равняется пяти. Когда функция достигает своего шестого вызова, параметр `times` равен 0. В этой точке условное выражение инструкции `if` становится ложным, и поэтому функция возвращается. Поток управления программы возвращается из шестого экземпляра функции в точку в пятом экземпляре непосредственно после вызова рекурсивной функции (рис. 12.3).



РИС. 12.3. Поток управления возвращается в точку после вызова рекурсивной функции

Поскольку больше нет инструкций, которые будут исполняться после вызова функции, пятый экземпляр функции возвращает поток управления программы назад в четвертый экземпляр. Это повторяется до тех пор, пока все экземпляры функции не вернутся.

12.2 Решение задач на основе рекурсии

Ключевые положения

Задача может быть решена на основе рекурсии, если ее разделить на уменьшенные задачи, которые по структуре идентичны общей задаче.

Пример кода, показанный в программе 12.2, демонстрирует механизм работы рекурсивной функции. Рекурсия может оказаться мощным инструментом для решения повторяющихся задач и обычно является предметом изучения на более старших курсах информатики. Возможно, вам пока еще не совсем ясно, каким образом рекурсия используется для решения задач.

Прежде всего обратите внимание, что рекурсия никогда не является непременным условием для решения задачи. Любая задача, которая может быть решена рекурсивно, также может быть решена с помощью цикла. В действительности, рекурсивные алгоритмы обычно менее эффективны, чем итеративные алгоритмы. Это связано с тем, что процесс вызова функции требует выполнения компьютером нескольких действий. Эти действия включают выделение памяти под параметры и локальные переменные и для хранения адреса местоположения программы, куда поток управления возвращается после завершения функции. Такие действия, которые иногда называются *накладными расходами*, происходят при каждом вызове функции. Накладные расходы не требуются при использовании цикла.

Вместе с тем, некоторые повторяющиеся задачи легче решаются на основе рекурсии, чем на основе цикла. Там, где цикл приводит к меньшему времени исполнения, программист быстрее разработает рекурсивный алгоритм. В целом рекурсивная функция работает следующим образом:

- ◆ если в настоящий момент задача может быть решена без рекурсии, то функция ее решает и возвращается;
- ◆ если в настоящий момент задача не может быть решена, то функция ее сводит к уменьшенной и при этом аналогичной задаче и вызывает саму себя для решения этой уменьшенной задачи.

Для того чтобы применить такой подход, во-первых, мы идентифицируем по крайней мере один случай, в котором задача может быть решена без рекурсии. Он называется *базовым случаем*. Во-вторых, мы определяем то, как задача будет решаться рекурсивно во всех остальных случаях. Это называется *рекурсивным случаем*. В рекурсивном случае мы все время должны сводить задачу к уменьшенному варианту исходной задачи. С каждым рекурсивным вызовом задача уменьшается. В результате будет достигнут базовый случай, и рекурсия прекратится.

Применение рекурсии для вычисления факториала числа

Для того чтобы исследовать применение рекурсивных функций, давайте возьмем пример из математики. В математике запись в форме $n!$ обозначает факториал числа n . Факториал неотрицательного числа определяется приведенными ниже правилами.

Если $n = 0$, то

$$n! = 1.$$

Если $n > 0$, то

$$n! = 1 \cdot 2 \cdot 3 \cdots n.$$

Заменим запись в форме $n!$ на $\text{factorial}(n)$, которая очень похожа на программный код, и перепишем эти правила следующим образом.

Если $n = 0$, то $\text{factorial}(n) = 1$.

Если $n > 0$, то $\text{factorial}(n) = 1 \cdot 2 \cdot 3 \cdots n$.

Эти правила определяют, что при $n = 0$ его факториал равняется 1. Когда же $n > 0$, его факториал является произведением всех положительных целых чисел от 1 до n . Например, $\text{factorial}(6)$ вычисляется так: $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6$.

Во время разработки рекурсивного алгоритма для вычисления факториала любого числа сначала идентифицируется базовый случай, являющийся той частью вычисления, которую можно решить без рекурсии. Это как раз тот случай, где n равняется 0, как показано ниже.

Если $n = 0$, то $\text{factorial}(n) = 1$.

Это решение задачи, когда $n = 0$, но вот что делать, когда $n > 0$? А это именно тот рекурсивный случай, или часть задачи, для решения которой применяется рекурсия. Вот как это выражается формально.

Если $n > 0$, то $\text{factorial}(n) = n \cdot \text{factorial}(n - 1)$.

В этой формуле говорится о том, что если $n > 0$, то факториал n равняется произведению n на факториал $n - 1$. Обратите внимание, как рекурсивный вызов работает с уменьшенным вариантом задачи, $n - 1$. Таким образом, наше рекурсивное правило вычисления факториала числа может выглядеть так.

Если $n = 0$, то $\text{factorial}(n) = 1$.

Если $n > 0$, то $\text{factorial}(n) = n \cdot \text{factorial}(n - 1)$.

В программе 12.3 представлена реализация функции factorial .

Программа 12.3 (factorial.py)

```

1 # Эта программа применяет рекурсию
2 # для вычисления факториала числа.
3
4 def main():
5     # Получить от пользователя число.
6     number = int(input('Введите неотрицательное целое число: '))
7
8     # Получить факториал числа.
9     fact = factorial(number)
10
11    # Показать факториал.
12    print(f'Факториал числа {number} равняется {fact}.')
13
14 # Функция factorial применяет рекурсию, чтобы
15 # вычислить факториал своего аргумента, который,
16 # как предполагается, является неотрицательным.
17 def factorial(num):
18     if num == 0:
19         return 1

```

```

20     else:
21         return num * factorial(num - 1)
22
23 # Вызвать главную функцию.
24 if __name__ == '__main__':
25     main()

```

Вывод программы (вводимые данные выделены жирным шрифтом)

Введите неотрицательное целое число: **4**

Факториал числа 4 равняется 24

В демонстрационном выполнении программы функция `factorial` вызывается с аргументом 4, который передается в параметр `num`. Поскольку `num` не равняется 0, выражение `else` инструкции `if` исполняет инструкцию:

```
return num * factorial(num - 1)
```

Хотя это и инструкция `return`, она не возвращается немедленно. Прежде чем возвращаемое значение будет определено, должно быть определено значение `factorial(num - 1)`. Функция `factorial` вызывается рекурсивно вплоть до пятого вызова, в котором будет обнулен параметр `num`. На рис. 12.4 проиллюстрированы значение `num` и возвращаемое значение во время каждого вызова функции.

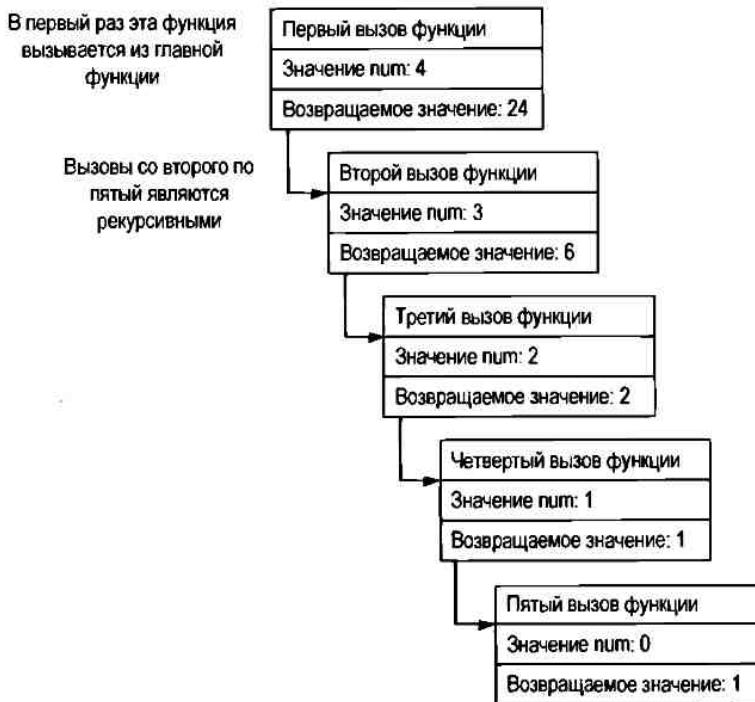


РИС. 12.4. Значение `num` и возвращаемое значение во время каждого вызова функции

Данный рисунок демонстрирует, почему рекурсивный алгоритм должен уменьшать задачу с каждым рекурсивным вызовом. В конечном счете рекурсия должна остановиться, чтобы решение было достигнуто.

Если каждый рекурсивный вызов работает с уменьшенным вариантом задачи, то работа рекурсивных вызовов сводится к базовому случаю. Базовый случай не требует рекурсии, и поэтому он останавливает цепочку рекурсивных вызовов.

Обычно задача уменьшается путем уменьшения значения одного или нескольких параметров с каждым рекурсивным вызовом. В нашей функции `factorial` значение параметра `num` приближается к 0 с каждым рекурсивным вызовом. Когда этот параметр достигает 0, данная функция возвращает значение, не делая больше рекурсивных вызовов.

Прямая и косвенная рекурсия

Примеры, которые мы рассматривали до сих пор, демонстрируют рекурсивные функции, или функции, которые вызывают сами себя непосредственно. Такой вызов называется *прямой рекурсией*. В программе можно также создавать *косвенную рекурсию*. Это происходит, когда функция A вызывает функцию B, которая в свою очередь вызывает функцию A. В рекурсии может участвовать даже несколько функций. Например, функция A может вызывать функцию B, которая вызывает функцию C, которая вызывает функцию A.



Контрольная точка

- 12.1. Что означает, когда говорят, что рекурсивный алгоритм имеет накладные расходы, которые превышают накладные расходы итеративного алгоритма?
- 12.2. Что такое базовый случай?
- 12.3. Что такое рекурсивный случай?
- 12.4. Что заставляет рекурсивный алгоритм прекратить вызывать самого себя?
- 12.5. Что такое прямая рекурсия? Что такое косвенная рекурсия?

12.3

Примеры алгоритмов на основе рекурсии

В этом разделе мы рассмотрим функцию `range_sum`, в которой рекурсия применяется для суммирования диапазона значений в списке. Функция принимает аргументы: список, содержащий диапазон значений, которые будут просуммированы; целое число, определяющее индексную позицию начального значения в диапазоне; целое число, определяющее индексную позицию конечного значения в диапазоне. Вот пример того, как функция могла бы использоваться:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
my_sum = range_sum(numbers, 3, 7)
print(my_sum)
```

Вторая инструкция в этом фрагменте кода определяет, что функция `range_sum` должна вернуть сумму значений в списке чисел в индексах с 3 по 7. Возвращаемое значение, которое в этом случае будет равняться 30, присваивается переменной `my_sum`. Вот определение функции `range_sum`:

```
def range_sum(num_list, start, end):
    if start > end:
        return 0
```

```

    else:
        return num_list[start] + range_sum(num_list, start + 1, end)

```

Базовый случай этой функции наступает, когда параметр `start` больше параметра `end`. Если это условие является истинным, то функция возвращает значение 0. В противном случае функция исполняет инструкцию:

```
return num_list[start] + range_sum(num_list, start + 1, end)
```

Эта инструкция возвращает сумму `num_list[start]` плюс значение, возвращаемое рекурсивным вызовом. Обратите внимание, что в рекурсивном вызове начальное значение в диапазоне равняется `start + 1`. Эта инструкция предписывает "вернуть значение первого элемента в диапазоне плюс сумму значений остальных элементов в диапазоне". Программа 12.4 демонстрирует данную функцию.

Программа 12.4 (range_sum.py)

```

1 # Эта программа демонстрирует функцию range_sum.
2
3 def main():
4     # Создать список чисел.
5     numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
6
7     # Получить сумму значений в индексных
8     # позициях, начиная с 2 вплоть до 5.
9     my_sum = range_sum(numbers, 2, 5)
10
11    # Показать сумму.
12    print(f'Сумма значений со 2 по 5 позицию равняется {my_sum}.')
13
14 # Функция range_sum возвращает сумму заданного
15 # диапазона значений в списке num_list. Параметр start
16 # задает индексную позицию начального значения.
17 # Параметр end задает индексную позицию конечного значения.
18 def range_sum(num_list, start, end):
19     if start > end:
20         return 0
21     else:
22         return num_list[start] + range_sum(num_list, start + 1, end)
23
24 # Вызвать главную функцию.
25 if __name__ == '__main__':
26     main()

```

Вывод программы

Сумма значений со 2 по 5 позицию равняется 18

Последовательность Фибоначчи

Некоторые математические задачи предназначены для рекурсивного решения. Одним хорошо известным примером является вычисление чисел Фибоначчи. Числа Фибоначчи, назван-

ные в честь средневекового итальянского математика Леонардо Фибоначчи (родившегося примерно в 1170 году), представлены приведенной ниже последовательностью:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Обратите внимание, что после второго числа каждое последующее число последовательности будет суммой двух предыдущих чисел. Последовательность Фибоначчи может быть определена следующим образом.

Если $n = 0$, то $\text{Fib}(n) = 0$.

Если $n = 1$, то $\text{Fib}(n) = 1$.

Если $n > 1$, то $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$.

Рекурсивная функция вычисления n -го числа в последовательности Фибоначчи приводится ниже:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Обратите внимание, что эта функция имеет два базовых случая: когда $n = 0$ и когда $n = 1$. В любом из них функция возвращает значение, не делая рекурсивного вызова. Программа 12.5 демонстрирует эту функцию, выводя первые 10 чисел в последовательности Фибоначчи.

Программа 12.5 (fibonacci.py)

```
1 # Эта программа применяет рекурсию для печати чисел
2 # последовательности Фибоначчи.
3
4 def main():
5     print('Первые 10 чисел')
6     print('последовательности Фибоначчи:')
7
8     for number in range(1, 11):
9         print(fib(number))
10
11 # Функция fib возвращает n-е число
12 # последовательности Фибоначчи.
13 def fib(n):
14     if n == 0:
15         return 0
16     elif n == 1:
17         return 1
18     else:
19         return fib(n - 1) + fib(n - 2)
20
```

```
21 # Вызвать главную функцию.
22 if __name__ == '__main__':
23     main()
```

Вывод программы

Первые 10 чисел последовательности Фибоначчи:

```
1
1
2
3
5
8
13
21
34
55
```

Нахождение наибольшего общего делителя

Нашим следующим примером рекурсии является вычисление наибольшего общего делителя двух чисел (НОД — greatest common divisor, GCD). НОД двух положительных целочисленных x и y определяется следующим образом.

Если x можно разделить на y без остатка, то $\text{gcd}(x, y) = y$.

В противном случае $\text{gcd}(x, y) = \text{gcd}(y, \text{остаток от деления } x/y)$.

Это определение обозначает, что НОД чисел x и y равняется числу y , если деление x/y не имеет остатка. Данное условие является базовым случаем. В противном случае ответом является НОД чисел y и остатка от x/y . В программе 12.6 приведен рекурсивный метод вычисления НОД.

Программа 12.6 (gcd.py)

```
1 # Эта программа применяет рекурсию для нахождения
2 # наибольшего общего делителя (НОД или GCD) двух чисел.
3
4 def main():
5     # Получить два числа.
6     num1 = int(input('Введите целое число: '))
7     num2 = int(input('Введите еще одно целое число: '))
8
9     # Показать НОД (GCD).
10    print(f'Наибольший общий делитель '
11          f'этих двух чисел равен {gcd(num1, num2)}.')
12
13 # Функция gcd возвращает наибольший общий
14 # делитель двух чисел.
```

```

15 def gcd(x, y):
16     if x % y == 0:
17         return y
18     else:
19         return gcd(x, x % y)
20
21 # Вызвать главную функцию.
22 if __name__ == '__main__':
23     main()

```

Вывод программы (вводимые данные выделены жирным шрифтом)

Введите целое число: **49**

Введите еще одно целое число: **28**

Наибольший общий делитель

этих двух чисел равен **7**

Ханойские башни

Ханойские башни — это математическая головоломка, которая в информатике часто используется для демонстрации мощи рекурсии. В этой головоломке используются три стержня и несколько колец с отверстиями в центре. Кольца нанизаны на один из стержней (рис. 12.5).

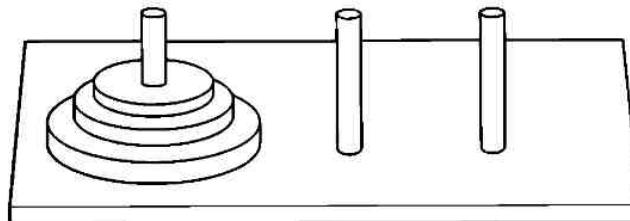


РИС. 12.5. Стержни и кольца в игре "Ханойские башни"

Обратите внимание, что кольца нанизаны на крайний левый стержень в порядке уменьшения их размера, т. е. самое большое кольцо находится внизу. Данная головоломка основывается на легенде, по которой у группы монахов в ханойском храме есть подобный набор стержней с 64 кольцами. Задача монахов состоит в перемещении колец с первого стержня на третий стержень. Средний стержень может использоваться в качестве временного хранения. Более того, при перемещении колец монахи должны соблюдать приведенные ниже правила:

- ◆ за один ход можно перемещать только одно кольцо;
- ◆ кольцо нельзя помещать поверх меньшего по размеру кольца;
- ◆ все кольца должны находиться на стержне за исключением случая, когда их перемещают.

Согласно легенде, когда монахи переместят все кольца с первого стержня на последний, наступит конец света¹.

¹ На случай, если вы волнуетесь по поводу того, закончат ли монахи свою работу и не вызовут ли они тем самым конец света, то можно расслабиться. Если монахи будут перемещать кольца со скоростью 1 кольцо в секунду, то им потребуется приблизительно 585 млрд лет, чтобы переместить все 64 кольца!

Цель этой головоломки — переместить все кольца с первого стержня на третий стержень, соблюдая те же самые правила, что и у монахов. Давайте рассмотрим несколько демонстрационных решений этой головоломки для разных количеств кольец. Если имеется всего одно кольцо, то решение простое: переместить кольцо со стержня 1 на стержень 3. Если имеются два кольца, то решение требует трех перемещений:

- ◆ переместить кольцо 1 на стержень 2;
- ◆ переместить кольцо 2 на стержень 3;
- ◆ переместить кольцо 1 на стержень 3.

Обратите внимание, что этот подход использует стержень 2 в качестве временного хранилища. Сложность перемещений возрастает вместе с увеличением количества колец. Для того чтобы переместить три кольца, требуется семь перемещений (рис. 12.6).

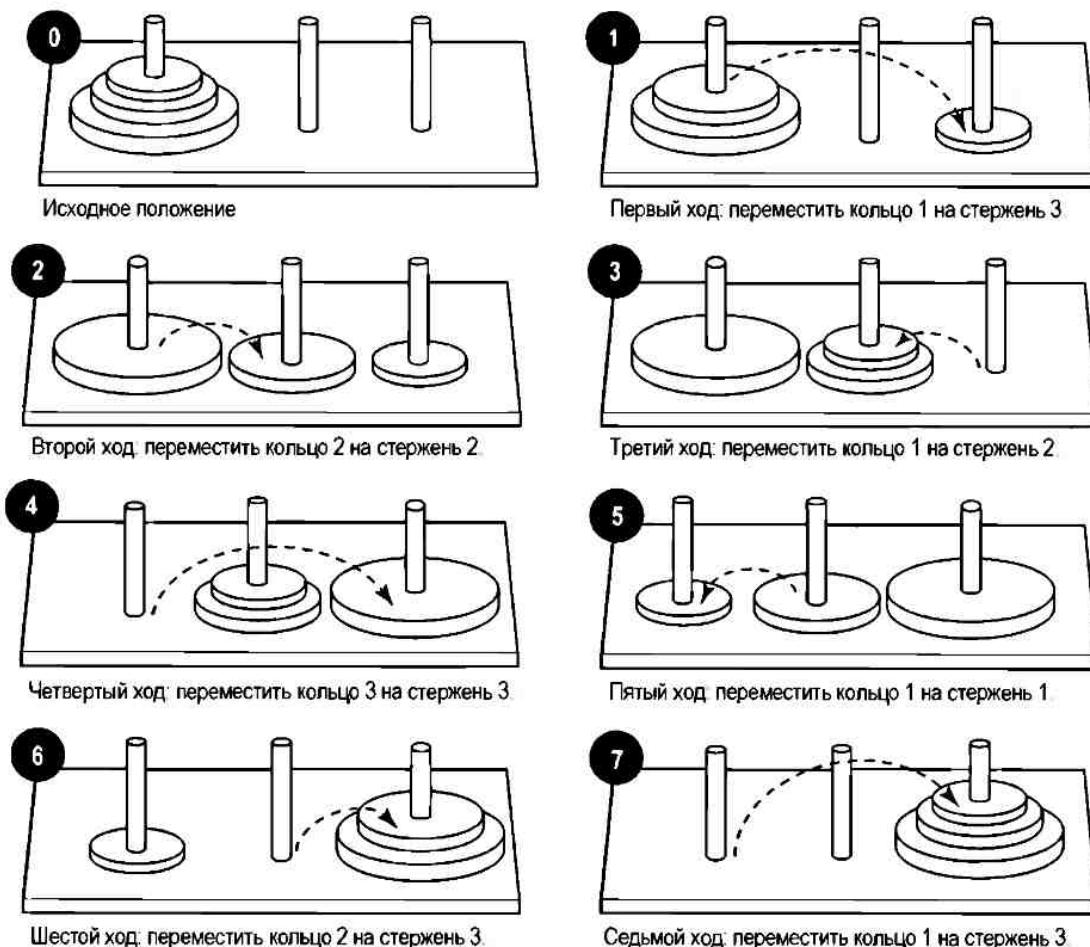


РИС. 12.6. Шаги перемещения трех колец

Приведенное ниже утверждение описывает общее решение задачи.

Переместить n колец со стержня 1 на стержень 3, используя стержень 2 в качестве временного стержня.

Представленное далее краткое резюме описывает рекурсивный алгоритм, который имитирует решение этой задачи. Обратите внимание, что в данном алгоритме для хранения номеров стержней используются переменные A, B и C.

Для того чтобы переместить n колец со стержня A на стержень C с использованием стержня B в качестве временного, необходимо сделать следующее.

Если $n > 0$:

Переместить $n - 1$ кольцо со стержня A на стержень B, используя стержень C в качестве временного стержня.

Переместить оставшееся кольцо со стержня A на стержень C.

Переместить $n - 1$ кольцо со стержня B на стержень C, используя стержень A в качестве временного стержня.

Базовый случай для этого алгоритма достигается, когда больше не останется колец, подлежащих перемещению. Приведенный ниже фрагмент кода демонстрирует функцию, которая реализует этот алгоритм. Эта функция практически ничего не перемещает. Она лишь содержит инструкции с указанием всех ходов, которые нужно сделать, чтобы переместить кольца.

```
def move_discs(num, from_peg, to_peg, temp_peg):  
    if num > 0:  
        move_discs(num - 1, from_peg, temp_peg, to_peg)  
        print('Переместить кольцо со стержня', from_peg, 'на стержень', to_peg)  
        move_discs(num - 1, temp_peg, to_peg, from_peg)
```

Данная функция принимает аргументы в следующие ниже параметры:

- ◆ num — количество перемещаемых колец;
- ◆ from_peg — стержень, с которого взять кольцо;
- ◆ to_peg — стержень, на который кольцо перемещается;
- ◆ temp_peg — стержень, используемый в качестве временного.

Если num больше 0, то это означает, что есть кольца, которые следует переместить. Первый рекурсивный вызов будет таким:

```
move_discs(num - 1, from_peg, temp_peg, to_peg)
```

Эта инструкция содержит указание, что все кольца, кроме одного, нужно переместить со стержня from_peg на стержень temp_peg, используя стержень to_peg в качестве временного. Далее идет инструкция:

```
print('Переместить кольцо со стержня', from_peg, 'на стержень', to_peg)
```

Эта инструкция просто выводит сообщение о том, что кольцо должно быть перемещено со стержня from_peg на стержень to_peg. Далее идет еще один рекурсивный вызов, который исполняется следующим образом:

```
move_discs(num - 1, temp_peg, to_peg, from_peg)
```

Эта инструкция содержит указание, что все кольца, кроме одного, нужно переместить со стержня temp_peg на стержень to_peg, используя стержень from_peg в качестве временного. Программе 12.7 демонстрирует данную функцию, показывая решение головоломки о ханойских башнях.

Программа 12.7 (towers_of_hanoi.py)

```

1 # Эта программа имитирует головоломку 'Ханойские башни'.
2
3 def main():
4     # Задать несколько исходных значений.
5     num_discs = 3
6     from_peg = 1
7     to_peg = 3
8     temp_peg = 2
9
10    # Решить головоломку.
11    move_discs(num_discs, from_peg, to_peg, temp_peg)
12    print('Все кольца перемещены!')
13
14 # Функция moveDiscs демонстрирует процесс перемещения
15 # колец в головоломке 'Ханойские башни'.
16 # Параметры функции:
17 # num: количество перемещаемых колец.
18 # from_peg: стержень, с которого взять кольцо.
19 # to_peg: стержень, на который кольцо перемещается.
20 # temp_peg: временный стержень.
21 def move_discs(num, from_peg, to_peg, temp_peg):
22     if num > 0:
23         move_discs(num - 1, from_peg, temp_peg, to_peg)
24         print(f'Переместить кольцо со стержня {from_peg} на стержень {to_peg}')
25         move_discs(num - 1, temp_peg, to_peg, from_peg)
26
27 # Вызвать главную функцию.
28 if __name__ == '__main__':
29     main()

```

Вывод программы

```

Переместить кольцо со стержня 1 на стержень 3
Переместить кольцо со стержня 1 на стержень 2
Переместить кольцо со стержня 3 на стержень 2
Переместить кольцо со стержня 1 на стержень 3
Переместить кольцо со стержня 2 на стержень 1
Переместить кольцо со стержня 2 на стержень 3
Переместить кольцо со стержня 1 на стержень 3
Все кольца перемещены!

```

Рекурсия против циклов

Любой алгоритм, использующий рекурсию, может быть запрограммирован с помощью цикла. Оба этих подхода успешно выполняют повторения, но какой из них применять лучше всего?

Есть несколько причин, объясняющих, почему не следует использовать рекурсию. Вызовы рекурсивной функции, разумеется, менее эффективны, чем циклы. При каждом вызове функции система несет накладные расходы, которые не требуются для цикла. Во многих случаях решение на основе цикла более очевидное, чем рекурсивное. Практически подавляющая часть итерационных задач программирования лучше всего решается с помощью циклов.

Вместе с тем некоторые задачи легче решаются на основе рекурсии, чем на основе цикла. Например, математическое определение формулы НОД хорошо укладывается в рекурсивный подход. Если для определенной задачи рекурсивное решение является очевидным, и рекурсивный алгоритм не сильно замедляет производительность системы, то рекурсия будет хорошим проектным решением. Однако если задача легче решается с помощью цикла, следует принять подход на основе цикла.

Вопросы для повторения

Множественный выбор

1. Рекурсивная функция _____.
 - а) вызывает другую функцию;
 - б) аварийно останавливает программу;
 - в) вызывает саму себя;
 - г) может вызываться всего один раз.
2. Функция вызывается один раз из главной функции программы и затем вызывает сама себя четыре раза. В этом случае глубина рекурсии будет равна _____.
 - а) одному;
 - б) четырем;
 - в) пяти;
 - г) девяти.
3. Часть задачи, которая может быть решена без рекурсии, называется _____.
случаем.
 - а) базовым;
 - б) разрешимым;
 - в) известным;
 - г) итеративным.
4. Часть задачи, которая может быть решена с использованием рекурсии, называется _____.
случаем.
 - а) базовым;
 - б) итеративным;
 - в) неизвестным;
 - г) рекурсивным.

5. Когда функция явным образом саму себя вызывает, это называется _____ рекурсией.
 - а) явной;
 - б) модальной;
 - в) прямой;
 - г) косвенной.
6. Ситуация, когда функция А вызывает функцию В, которая вызывает функцию А, называется _____ рекурсией.
 - а) имплицитной;
 - б) модальной;
 - в) прямой;
 - г) косвенной.
7. Любая задача, которая может быть решена на основе рекурсии, может также быть решена с помощью _____.
 - а) структуры принятия решения;
 - б) цикла;
 - в) последовательной структуры;
 - г) выбирающей структуры.
8. Действия, такие как выделение оперативной памяти под параметры и локальные переменные, предпринимаемые компьютером при вызове функции, называются _____.
 - а) накладными расходами;
 - б) первоначальной настройкой;
 - в) очисткой;
 - г) синхронизацией.
9. Рекурсивный алгоритм в рекурсивном случае должен _____.
 - а) решить задачу без рекурсии;
 - б) свести задачу к уменьшенному варианту исходной задачи;
 - в) подтвердить, что произошла ошибка и прервать программу;
 - г) свести задачу к увеличенному варианту исходной задачи.
10. Рекурсивный алгоритм в базовом случае должен _____.
 - а) решить задачу без рекурсии;
 - б) свести задачу к уменьшенному варианту исходной задачи;
 - в) подтвердить, что произошла ошибка и прервать программу;
 - г) свести задачу к увеличенному варианту исходной задачи.

Истина или ложь

1. Алгоритм, в котором применяется цикл, обычно будет работать быстрее, чем эквивалентный рекурсивный алгоритм.

2. Некоторые задачи могут быть решены только на основе рекурсии.
3. Не во всех рекурсивных алгоритмах обязательно должен иметься базовый случай.
4. В базовом случае рекурсивный метод вызывает сам себя с уменьшенным вариантом исходной задачи.

Короткий ответ

1. Каким является базовый случай функции `message` из программы 12.2, представленной ранее в этой главе?
2. В этой главе правила вычисления факториала числа состояли в следующем.

Если $n = 0$, то $\text{factorial}(n) = 1$.

Если $n > 0$, то $\text{factorial}(n) = n \cdot \text{factorial}(n - 1)$.

Каким будет базовый случай, если вы разрабатываете функцию на основе этих правил?
Каким будет рекурсивный случай?
3. Всегда ли требуется рекурсия для решения задачи? Какой еще подход применяется для решения повторяющейся по своей природе задачи?
4. Почему при использовании рекурсии для решения задачи рекурсивная функция должна вызывать саму себя для решения уменьшенного варианта исходной задачи?
5. Каким образом рекурсивная функция уменьшает задачу?

Алгоритмический тренажер

1. Что покажет приведенная ниже программа?

```
def main():
    num = 0
    show_me(num)
def show_me(arg):
    if arg < 10:
        show_me(arg + 1)
    else:
        print(arg)
main()
```

2. Что покажет приведенная ниже программа?

```
def main():
    num = 0
    show_me(num)
def show_me(arg):
    print(arg)
    if arg < 10:
        show_me(arg + 1)
main()
```

3. В приведенной ниже функции применен цикл. Перепишите ее в виде рекурсивной функции, которая выполняет ту же самую операцию.

```
def traffic_sign(n):
    while n > 0:
        print('Не парковаться')
        n = n - 1
```

Упражнения по программированию

- Рекурсивная печать.** Разработайте рекурсивную функцию, которая принимает целочисленный аргумент *n* и распечатывает числа от 1 до *n*.
- Рекурсивное умножение.**



Видеозапись "Рекурсивное умножение" (Recursive Multiplication)

Разработайте рекурсивную функцию, которая принимает два аргумента в параметры *x* и *y*. Данная функция должна вернуть значение произведения *x* на *y*. При этом умножение должно быть выполнено, как повторяющееся сложение, следующим образом:

$$7 \cdot 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4.$$

(Для упрощения функции исходите из того, что *x* и *y* будут всегда содержать положительные ненулевые целые числа.)

- Рекурсивные строки.** Напишите рекурсивную функцию, которая принимает целочисленный аргумент *n*. Данная функция должна вывести на экран *n* строк, состоящих из звездочек; при этом первая строка должна показать 1 звездочку, вторая строка — 2 звездочки и так до *n*-й строки, которая должна показать *n* звездочек.
- Максимальное значение в списке.** Разработайте функцию, которая принимает список в качестве аргумента и возвращает самое большое значение в списке. В данной функции для нахождения максимального значения должна использоваться рекурсия.
- Рекурсивная сумма списка.** Разработайте функцию, которая принимает список чисел в качестве аргумента. Она должна рекурсивно вычислить сумму всех чисел в списке и вернуть это значение.
- Сумма чисел.** Разработайте функцию, которая принимает целочисленный аргумент и возвращает сумму всех целых чисел от 1 до числа, переданного в качестве аргумента. Например, если в качестве аргумента передано 50, то данная функция вернет сумму чисел 1, 2, 3, 4, ..., 50. Для вычисления суммы примените рекурсию.
- Рекурсивный метод возведения в степень.** Разработайте функцию, в которой рекурсия применяется для возведения числа в степень. Данная функция должна принимать два аргумента: число, которое будет возведено в степень, и показатель степени. Исходите из того, что показатель степени является неотрицательным целым числом.
- Функция Аккерманна.** Функция Аккерманна является рекурсивным математическим алгоритмом, который используется для проверки, насколько успешно система оптимизирует свою производительность в случае рекурсии. Разработайте функцию *ackermann(m, n)*, которая решает функцию Аккерманна. Примените в своей функции следующую логику:

Если *m* = 0, то вернуть *n* + 1.

Если *n* = 0, то вернуть *ackermann(m - 1, 1)*.

Иначе вернуть *ackermann(m - 1, ackermann(m, n - 1))*.

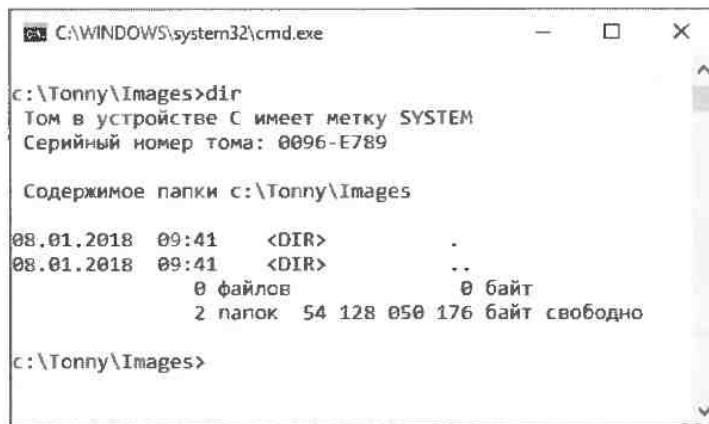
После того как вы разработаете свою функцию, протестируйте ее с использованием небольших значений для *m* и *n*.

13.1 Графические интерфейсы пользователя

Ключевые положения

Графический интерфейс пользователя позволяет взаимодействовать с операционной системой и другими программами с помощью графических элементов, таких как значки, кнопки и диалоговые окна.

Интерфейс пользователя является частью компьютера, с которым пользователь взаимодействует. Аппаратная часть пользовательского интерфейса состоит из устройств, таких как клавиатура и видеодисплей. Программная часть пользовательского интерфейса отвечает за то, как операционная система компьютера принимает команды от пользователя. В течение многих лет единственным способом, благодаря которому пользователь мог взаимодействовать с операционной системой, являлся *интерфейс командной строки* (рис. 13.1). Интерфейс командной строки обычно выводит на экран подсказку, и пользователь набирает команду, которая затем исполняется.



```
C:\WINDOWS\system32\cmd.exe
c:\Tonny\Images>dir
Том в устройстве C имеет метку SYSTEM
Серийный номер тома: 0096-E789

Содержимое папки c:\Tonny\Images

08.01.2018 09:41    <DIR>    .
08.01.2018 09:41    <DIR>    ..
          0 файлов          0 байт
          2 папок  54 128 050 176 байт свободно

c:\Tonny\Images>
```

РИС. 13.1. Интерфейс командной строки

Многие компьютерные пользователи, в особенности новички, считают, что интерфейсы командной строки сложно использовать. И причина тому — обилие команд, которые необходимо изучить, причем каждая команда имеет собственный синтаксис, во многом как у программной инструкции. Если команда набрана неправильно, то она работать не будет.

В 1980-х годах в коммерческих операционных системах вошел в употребление новый тип интерфейса, именуемый графическим интерфейсом пользователя. *Графический интерфейс*

пользователя (graphical user interface, GUI — на английском эта аббревиатура произносится как "гуи") позволяет пользователю взаимодействовать с операционной системой и другими программами через графические элементы на экране. GUI также популяризовали использование мыши как устройства ввода данных. Вместо того чтобы требовать от пользователя набора команд на клавиатуре, GUI позволяют ему указывать на графические элементы и щелкать кнопкой мыши для их активации.

Значительная часть взаимодействия с GUI выполняется через *диалоговые окна*, т. е. небольшие окна, которые выводят информацию и позволяют пользователю выполнять действия. На рис. 13.2 показан пример диалогового окна в операционной системе Windows, которое позволяет пользователю вносить изменения в интернет-настройки системы. Вместо того чтобы набирать команды в соответствии с заданным синтаксисом, пользователь взаимодействует с графическими элементами — значками, кнопками и полосами прокрутки.

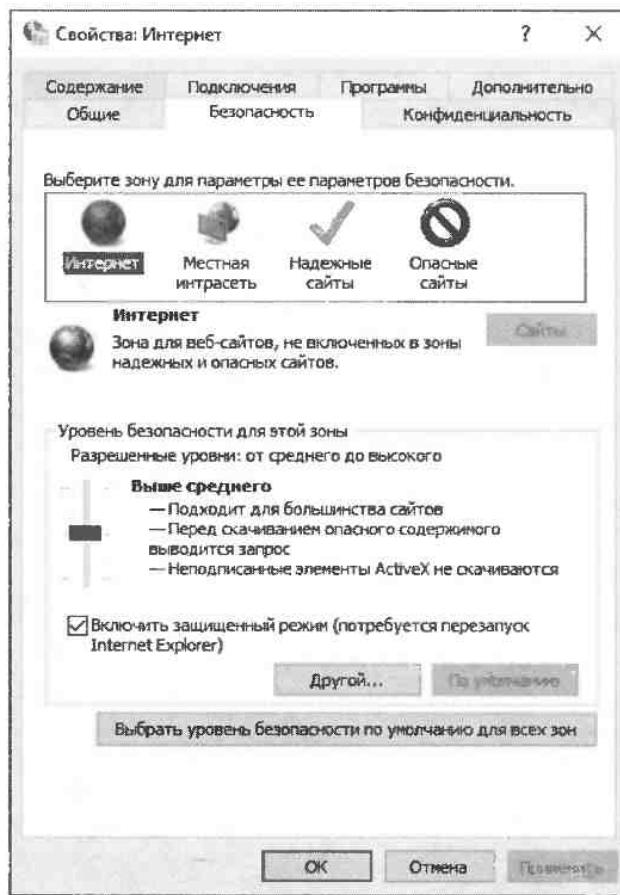


РИС. 13.2. Диалоговое окно

Программы с GUI, управляемые событиями

В текст-ориентированной среде, такой как интерфейс командной строки, программы определяют порядок, в котором все происходит. Например, рассмотрим программу, которая вычисляет площадь прямоугольника. Сначала программа предлагает пользователю ввести ширину прямоугольника. Пользователь вводит ее. Затем программа предлагает ввести длину

прямоугольника. Пользователь вводит длину, после чего программа вычисляет площадь. У пользователя нет выбора, кроме как ввести данные в том порядке, в котором его просят.

Однако в среде GUI теперь уже пользователь определяет порядок, в котором все происходит. Например, на рис. 13.3 показано окно программы с GUI (написанной на Python), которая вычисляет площадь прямоугольника. Пользователь может ввести длину и ширину в любом порядке, в котором он пожелает. Если сделана ошибка, то пользователь может удалить введенные данные и набрать их заново. Когда пользователь готов вычислить площадь, он нажимает кнопку **Вычислить площадь**, и программа выполняет это вычисление. Поскольку программы с GUI должны реагировать на действия пользователя, говорится, что они являются *событийно-управляемыми*. Действия пользователя приводят к возникновению событий, таких как щелчок кнопки, и программа должна реагировать на эти события.

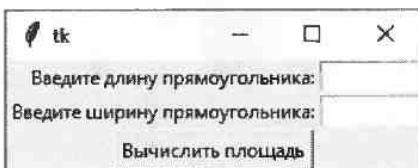


РИС. 13.3. Программа с GUI



Контрольная точка

- 13.1. Что такое пользовательский интерфейс?
- 13.2. Как работает интерфейс командной строки?
- 13.3. Что определяет порядок, в котором все происходит, когда пользователь выполняет программу в текстоориентированной среде, такой как командная строка?
- 13.4. Что такое событийно-управляемая программа?

13.2 Использование модуля *tkinter*

Ключевые положения

В Python для создания простых программ с GUI используется модуль *tkinter*.

В Python нет встроенного в язык функционала программирования GUI. Однако он поставляется с модулем *tkinter*, который позволяет создавать простые программы с GUI. Имя модуля "tkinter" является сокращением от "Tk interface" ("интерфейс с Tk"). Такое название связано с тем, что модуль предоставляет программистам на языке Python возможность пользоваться GUI-библиотекой под названием Tk. Библиотека Tk также используется во многих других языках программирования.



ПРИМЕЧАНИЕ

В Python имеется целый ряд библиотек GUI. И поскольку модуль *tkinter* поставляется вместе с Python, в этой главе мы будем использовать только его¹.

¹ В исходный код главы 13 добавлена имплементация приводимых в этой главе примеров с использованием стилизованного модуля *tk*. — Прим. пер.

Программа с GUI выводит окно с различными элементами графического интерфейса, или *виджетами*¹, с которыми пользователь может взаимодействовать или которые может просматривать. Модуль `tkinter` предоставляет 15 виджетов (табл. 13.1). В этой главе мы не сможем охватить все виджеты модуля `tkinter`, однако продемонстрируем способы создания простых программ с GUI, которые собирают входные данные и выводят их на экран.

Таблица 13.1. Виджеты модуля `tkinter`

Виджет	Описание
<code>Button</code>	Кнопка, которая при нажатии вызывает наступление действия
<code>Canvas</code>	Прямоугольная область, которая используется для отображения графики
<code>Checkbutton</code>	Кнопка, которая может быть в положении "включено" либо "выключено"
<code>Entry</code>	Область, в которую пользователь может ввести одну строку входных данных с клавиатуры
<code>Frame</code>	Контейнер, который может содержать другие виджеты
<code>Label</code>	Область, которая выводит на экран одну строку текста или изображение
<code>Listbox</code>	Список, из которого пользователь может выбрать значение
<code>Menu</code>	Список пунктов меню, которые выводятся на экран, когда пользователь нажимает на виджете <code>Menubutton</code>
<code>Menubutton</code>	Меню, которое выводится на экран и на которое пользователь может нажать мышью
<code>Message</code>	Выводит многочисленные строки текста
<code>Radiobutton</code>	Виджет, который может быть либо выбран, либо не выбран. Виджеты <code>Radiobutton</code> обычно появляются в группах и позволяют пользователю выбирать один из нескольких вариантов
<code>Scale</code>	Виджет, который позволяет пользователю выбирать значение путем перемещения ползунка вдоль шкалы
<code>Scrollbar</code>	Может использоваться с некоторыми другими типами виджетов для обеспечения возможности прокрутки
<code>Text</code>	Виджет, который позволяет пользователю вводить многочисленные строки текстового ввода
<code>Toplevel</code>	Контейнер, аналогичный виджету <code>Frame</code> , но в отличие от него выводимый на экран в собственном окне

Самой простой программой с GUI, которую можно продемонстрировать, является программа, выводящая на экран пустое окно. В программе 13.1 показано, как это делается при помощи модуля `tkinter`. Во время запуска программы на экране появляется окно (рис. 13.4). Для выхода из программы просто щелкните на стандартной для Windows кнопке закрытия окна (x) в его правом верхнем углу.

¹ Слово "виджет" (widget) образовано в результате сложения двух английских слов: `windows` (окна) и `gadget` (приспособление), — и впервые появилось в среде ОС Windows, обозначая компонент графического интерфейса. — *Прим. пер.*

ПРИМЕЧАНИЕ

Программы с использованием модуля `tkinter` не всегда работают надежно в среде IDLE, т. к. IDLE сам тоже использует модуль `tkinter`. Вы всегда можете воспользоваться редактором IDLE для написания программ с GUI, но для достижения наилучших результатов их следует выполнять из командной строки операционной системы.

Программа 13.1 (empty_window1.py)

```
1 # Эта программа показывает пустое окно.
2
3 import tkinter
4
5 def main():
6     # Создать виджет главного окна.
7     main_window = tkinter.Tk()
8
9     # Войти в главный цикл tkinter.
10    tkinter.mainloop()
11
12 # Вызвать главную программу.
13 if __name__ == '__main__':
14     main()
```

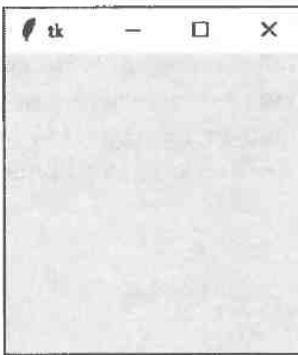


РИС. 13.4. Окно, выводимое на экран программой 13.1

Строка 3 импортирует модуль `tkinter`. В главной функции строка 7 создает экземпляр класса `Tk` модуля `tkinter` и присваивает его переменной `main_window`. Этот объект — корневой виджет, который является главным окном в программе. Стока 10 вызывает функцию `mainloop` модуля `tkinter`. Она работает как бесконечный цикл до тех пор, пока главное окно не будет закрыто.

Большинство программистов при написании программ с GUI предпочитают принимать объектно-ориентированный подход. Вместо того чтобы писать функцию для создания экранных элементов программы, общепринятой практикой является написание класса с методом `__init__()`, который создает GUI. Когда создается экземпляр класса, на экране появляется GUI. Для того чтобы это продемонстрировать, в программе 13.2 приведена объектно-ориентированная версия нашей предыдущей программы, которая отображает на экране пустое окно (см. рис. 13.4).

Программа 13.2 (empty_window2.py)

```

1 # Эта программа показывает пустое окно.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Создать виджет главного окна.
8         self.main_window = tkinter.Tk()
9
10    # Войти в главный цикл tkinter.
11    tkinter.mainloop()
12
13 # Создать экземпляр класса MyGUI.
14 if __name__ == '__main__':
15     my_gui = MyGUI()

```

Строки 5–11 являются определением класса MyGUI. Метод `__init__()` этого класса начинается в строке 6. Стока 8 создает корневой виджет и присваивает его атрибуту класса `main_window`. Стока 11 выполняет функцию `mainloop` модуля `tkinter`. Инструкция в строке 14 создает экземпляр класса MyGUI. Это приводит к выполнению метода `__init__()` данного класса, который выводит на экран пустое окно.

При необходимости можно отобразить текст в заголовке окна, вызвав метод `title()` оконного объекта. В качестве аргумента надо передать текст, который вы хотите вывести на экран. Программа 13.3 демонстрирует пример. При запуске программы на экран выводится окно, показанное на рис. 13.5. (Возможно, вам придется изменить размер окна, чтобы увидеть весь заголовок.)

Программа 13.3 (window_with_title.py)

```

1 # Эта программа показывает пустое окно.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Создать виджет главного окна.
8         self.main_window = tkinter.Tk()
9
10    # Показать заголовок.
11    self.main_window.title('Мой первый GUI')
12
13    # Войти в главный цикл tkinter.
14    tkinter.mainloop()
15

```

```

16 # Создать экземпляр класса MyGUI.
17 if __name__ == '__main__':
18     my_gui = MyGUI()

```

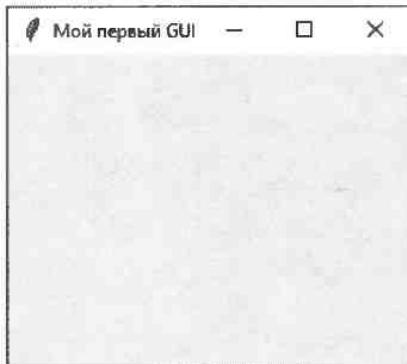


РИС. 13.5. Окно, выводимое на экран программой 13.3



Контрольная точка

- 13.5. Кратко опишите каждый из приведенных ниже виджетов модуля `tkinter`:
 - а) `Label`;
 - б) `Entry`;
 - в) `Button`;
 - г) `Frame`.
- 13.6. Как создать корневой виджет?
- 13.7. Что делает функция `mainloop` модуля `tkinter`?



13.3 Вывод текста с помощью виджетов *Label*

Ключевые положения

Виджет `Label` используется для вывода в окне надписи.



Видеозапись "Создание простого приложения с GUI" (*Creating a Simple GUI*)

Виджет `Label` используется для вывода в окне односторонней надписи. В целях создания виджета `Label` следует создать экземпляр класса `Label` модуля `tkinter`. Программа 13.4 создает окно, содержащее виджет `Label`, который выводит на экран надпись "Привет, мир!" (рис. 13.6).

Программа 13.4 (hello_world.py)

```

1 # Эта программа показывает надпись с текстом.
2
3 import tkinter
4

```

```

5 class MyGUI:
6     def __init__(self):
7         # Создать виджет главного окна.
8         self.main_window = tkinter.Tk()
9
10    # Создать виджет Label, содержащий
11    # надпись 'Привет, мир!'
12    self.label = tkinter.Label(self.main_window,
13                               text='Привет, мир!')
14
15    # Вызвать метод pack виджета Label.
16    self.label.pack()
17
18    # Войти в главный цикл tkinter.
19    tkinter.mainloop()
20
21 # Создать экземпляр класса MyGUI.
22 if __name__ == '__main__':
23     my_gui = MyGUI()

```

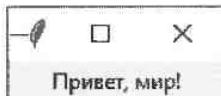


РИС. 13.6. Окно, выводимое на экран программой 13.4

Класс `MyGUI` в этой программе очень похож на тот, который вы видели ранее в программе 13.2. Метод `__init__()` создает GUI во время создания экземпляра класса. Стока 8 создает корневой виджет и присваивает его переменной `self.main_window`. Приведенная ниже инструкция расположена в строках 12 и 13:

```
self.label = tkinter.Label(self.main_window,
                           text='Привет, мир!')
```

Эта инструкция создает виджет `Label` и присваивает его переменной `self.label`. Первым аргументом внутри круглых скобок является `self.main_window`, т. е. ссылка на корневой виджет. Этот аргумент указывает, что мы хотим, чтобы виджет `Label` принадлежал корневому виджету. Вторым аргументом является `text='Привет, мир!'`. Этот аргумент определяет текст, который мы хотим вывести на экран в надписи.

Инструкция в строке 16 вызывает метод `pack()` виджета `Label`. Метод `pack()` определяет, где виджет должен быть расположен, и делает его видимым, когда корневой виджет выводится на экран. (Метод `pack()` вызывается для каждого виджета в окне.) Стока 19 вызывает метод `mainloop()` модуля `tkinter`, который выводит на экран главное окно программы, показанное на рис. 13.6.

Давайте рассмотрим еще один пример. Программа 13.5 выводит на экран окно с двумя виджетами `Label` (рис. 13.7).

Программа 13.5 (hello_world2.py)

```

1 # Эта программа показывает два виджета Label с надписями.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Создать виджет главного окна.
8         self.main_window = tkinter.Tk()
9
10        # Создать два виджета Label.
11        self.label1 = tkinter.Label(self.main_window,
12                                text='Привет, мир!')
13        self.label2 = tkinter.Label(self.main_window,
14                                text='Это моя программа с GUI.')
15
16        # Вызвать метод pack обоих виджетов Label.
17        self.label1.pack()
18        self.label2.pack()
19
20        # Войти в главный цикл tkinter.
21        tkinter.mainloop()
22
23 # Создать экземпляр класса MyGUI.
24 if __name__ == '__main__':
25     my_gui = MyGUI()

```

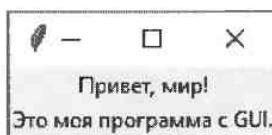


РИС. 13.7. Окно, выводимое на экран программой 13.5

Обратите внимание, что теперь выводятся два виджета Label, размещенные один под другим. Размещение виджетов можно изменить, указав аргумент для метода pack(), как показано в программе 13.6. Во время запуска программы она выводит окно (рис. 13.8).

Программа 13.6 (hello_world3.py)

```

1 # Эта программа использует аргумент side='left'
2 # в методе pack для изменения расстановки виджетов.
3
4 import tkinter
5

```

```

6 class MyGUI:
7     def __init__(self):
8         # Создать виджет главного окна.
9         self.main_window = tkinter.Tk()
10
11     # Создать два виджета Label.
12     self.label1 = tkinter.Label(self.main_window,
13                               text='Привет, мир!')
14     self.label2 = tkinter.Label(self.main_window,
15                               text='Это моя программа с GUI.')
16
17     # Вызвать метод pack обоих виджетов Label.
18     self.label1.pack(side='left')
19     self.label2.pack(side='left')
20
21     # Войти в главный цикл tkinter.
22     tkinter.mainloop()
23
24 # Создать экземпляр класса MyGUI.
25 if __name__ == '__main__':
26     my_gui = MyGUI()

```

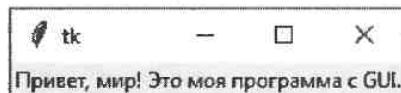


РИС. 13.8. Окно, выводимое на экран программой 13.6

В строках 18 и 19 вызывается метод `pack()` каждого виджета `Label`, передающий аргумент `side='left'`. Этот аргумент определяет, что виджет должен быть расположен в родительском виджете максимально слева. Поскольку сначала в `main_window` был добавлен виджет `label1`, он появится в крайней левой области окна. Затем был добавлен виджет `label2`, поэтому он появляется рядом с виджетом `label1`. В результате надписи отобразятся рядом друг с другом. Допустимыми аргументами `side`, которые можно передавать в метод `pack()`, являются `side='top'`, `side='bottom'`, `side='left'` и `side='right'`.

Добавление границ в виджет `Label`

При создании виджета `Label` можно дополнительно отобразить границу вокруг него. Вот пример:

```

self.label = tkinter.Label(self.main_window,
                          text='Привет, мир!',
                          borderwidth=1,
                          relief='solid')

```

Обратите внимание, что мы передаем аргументы `borderwidth=1` и `relief='solid'`. Аргумент `borderwidth` задает толщину границы в пикселях. В этом примере граница будет иметь тол-

щину 1 пиксел. Аргумент `relief` задает стиль границы. В этом примере граница будет сплошной линией. На рис. 13.9 показан выводимый на экран виджет `Label`.

В следующем коде показано, как создать тот же виджет надписи со сплошной границей толщиной 4 пикселя:

```
self.label = tkinter.Label(self.main_window,
                           text='Привет, мир!',
                           borderwidth=4,
                           relief='solid')
```

На рис. 13.10 показан выводимый на экран виджет `Label`.

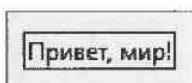


РИС. 13.9. Надпись, выводимая на экран с однотипной сплошной границей



РИС. 13.10. Надпись, выводимая на экран с четырехпиксельной сплошной границей

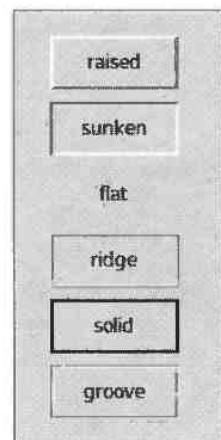


РИС. 13.11. Примеры разных стилей границ

В табл. 13.2 описаны различные значения, которые можно передать в качестве аргумента `relief`. Каждое значение приводит к тому, что граница отображается в том или ином стиле. На рис. 13.11 приведен пример каждого стиля.

Таблица 13.2. Варианты рельефа границы

Значение аргумента <code>relief</code>	Описание
<code>relief='flat'</code>	Граница скрыта и нет 3D-эффекта
<code>relief='raised'</code>	Виджет имеет приподнятый 3D-вид
<code>relief='sunken'</code>	Виджет имеет погруженный 3D-вид
<code>relief='ridged'</code>	Граница вокруг виджета имеет рифленый 3D-вид
<code>relief='solid'</code>	Граница выводится в виде сплошной линии без 3D-эффекта
<code>relief='groove'</code>	Граница вокруг виджета отображается в виде канавки

Заполнение

Заполнение (padding) — это пространство, которое появляется вокруг виджета. Существует два типа заполнения: внутреннее и внешнее. *Внутреннее заполнение* появляется вокруг

внутреннего края виджета, а *внешнее* — вокруг внешнего края виджета. На рис. 13.12 показана разница между двумя типами заполнения. Оба виджета *Label* имеют сплошную границу толщиной в 1 пиксель. У виджета слева — 20 пикселов внутреннего заполнения, а у виджета справа — 20 пикселов внешнего заполнения. Как видно из рисунка, внутреннее заполнение увеличивает размер виджета, в то время как внешнее заполнение увеличивает пространство вокруг виджета.



РИС. 13.12. Внутреннее и внешнее заполнение

Добавление внутреннего заполнения

Для того чтобы добавить внутреннее заполнение в виджет, надо передать в метод `pack()` виджета следующие аргументы:

- ◆ `ipadx=n;`
- ◆ `ipady=n;`

В обоих аргументах *n* — это число пикселов. Аргумент `ipadx` задает число пикселов внутреннего горизонтального заполнения, а аргумент `ipady` — число пикселов внутреннего вертикального заполнения. Это показано на рис. 13.13.

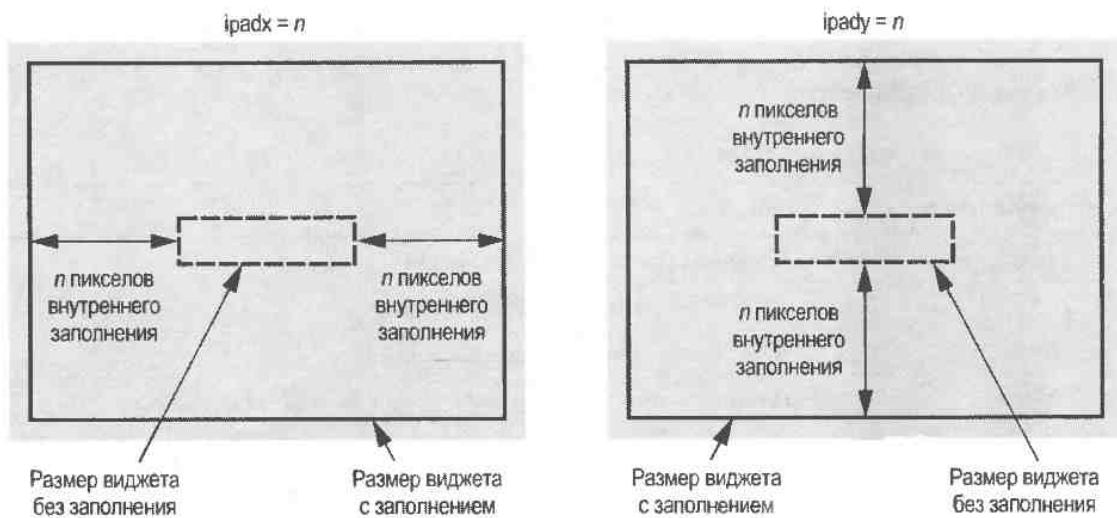


РИС. 13.13. Горизонтальное и вертикальное внутреннее заполнение

Программа 13.7 демонстрирует внутреннее заполнение. Она выводит на экран два виджета *Label* с 20 пикселями горизонтального и вертикального внутреннего заполнения. Графический интерфейс программы показан на рис. 13.14.

Программа 13.7 (internal_padding.py)

```

1 # Эта программа демонстрирует внутреннее заполнение.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать виджет главного окна.
7         self.main_window = tkinter.Tk()
8
9         # Создать два виджета Label со сплошными границами.
10        self.label1 = tkinter.Label(self.main_window,
11                                text='Привет, мир!',
12                                borderwidth=1,
13                                relief='solid')
14
15        self.label2 = tkinter.Label(self.main_window,
16                                text='Это моя программа с GUI.',
17                                borderwidth=1,
18                                relief='solid')
19
20        # Вывести на экран виджеты Label с 20 пикселями
21        # горизонтального внутреннего и вертикального внутреннего заполнения.
22        self.label1.pack(ipadx=20, ipady=20)
23        self.label2.pack(ipadx=20, ipady=20)
24
25        # Войти в главный цикл tkinter.
26        tkinter.mainloop()
27
28 # Создать экземпляр класса MyGUI.
29 if __name__ == '__main__':
30     my_gui = MyGUI()

```

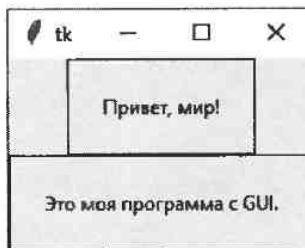


РИС. 13.14. Окно, выводимое на экран программой 13.7

Инструкции, которые появляются в строках 10–13 и 15–18, создают два виджета Label с именами label1 и label2. Каждый из них создается со сплошной границей в 1 пикселе. Строки 22 и 23 вызывают метод pack() виджетов, передавая аргументы ipadx=20 и ipady=20.

Добавление внешнего заполнения

Для того чтобы добавить внешнее заполнение в виджет, следует передать в метод `pack()` виджета следующие аргументы:

- ◆ `padx=n;`
- ◆ `pady=n;`

В обоих аргументах *n* — это число пикселов. Аргумент `padx` задает число пикселов внешнего горизонтального заполнения, а аргумент `pady` — число пикселов внешнего вертикального заполнения. Это показано на рис. 13.15.

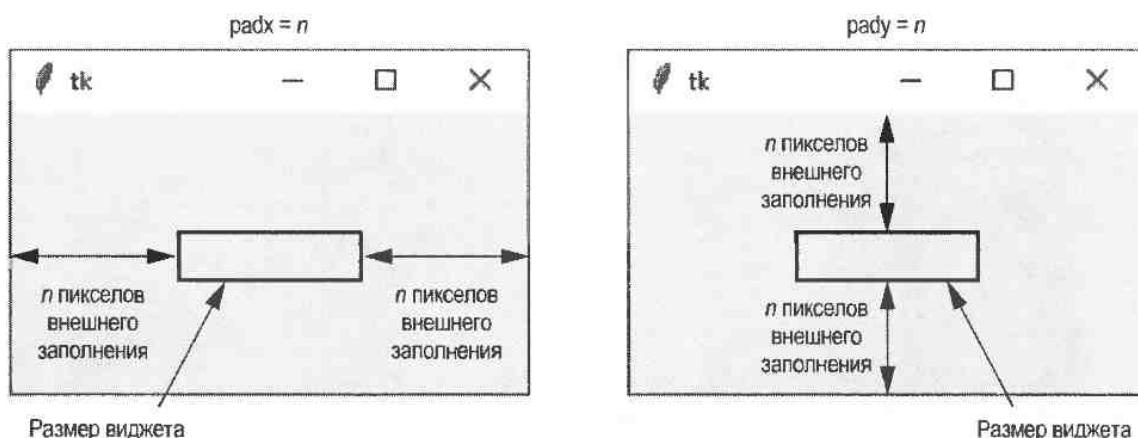


РИС. 13.15. Горизонтальное внешнее и вертикальное внешнее заполнение

Программа 13.8 демонстрирует внутреннее заполнение. Она выводит на экран два виджета `Label` с 20 пикселями горизонтального и вертикального внешнего заполнения. Графический интерфейс программы показан на рис. 13.16.

Программа 13.8 (external_padding.py)

```

1 # Эта программа демонстрирует внешнее заполнение.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать виджет главного окна.
7         self.main_window = tkinter.Tk()
8
9         # Создать два виджета Label со сплошными границами.
10        self.label1 = tkinter.Label(self.main_window,
11                                text='Привет, мир!',
12                                borderwidth=1,
13                                relief='solid')
14

```

```

15     self.label2 = tkinter.Label(self.main_window,
16                             text='Это моя программа с GUI.',
17                             borderwidth=1,
18                             relief='solid')
19
20     # Вывести на экран виджеты Label с 20 пикселями
21     # горизонтального внешнего и вертикального внешнего заполнения.
22     self.label1.pack(ipadx=20, ipady=20)
23     self.label2.pack(ipadx=20, ipady=20)
24
25     # Войти в главный цикл tkinter.
26     tkinter.mainloop()
27
28 # Создать экземпляр класса MyGUI.
29 if __name__ == '__main__':
30     my_gui = MyGUI()

```

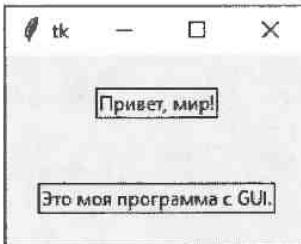


РИС. 13.16. Окно, выводимое на экран программой 13.8

Одновременное добавление внутреннего и внешнего заполнения

Виджет можно вывести на экран одновременно с внутренним и внешним заполнением. Например, мы можем изменить строки 22 и 23 программы 13.7 следующим образом:

```

self.label1.pack(ipadx=20, ipady=20, padx=20, pady=20)
self.label2.pack(ipadx=20, ipady=20, padx=20, pady=20)

```

И графический интерфейс программы будет выглядеть так, как показано на рис. 13.17.

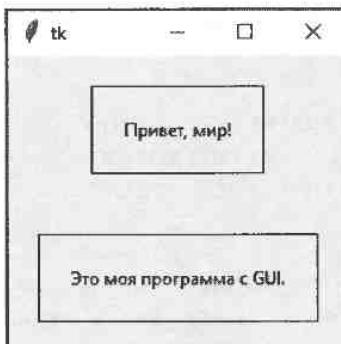


РИС. 13.17. Виджеты Label с внутренним и внешним заполнением

Добавление разного количества внешнего заполнения с каждой стороны

Иногда могут потребоваться разные количества заполнения с каждой стороны виджета. Например, вам могут понадобиться 5-пиксельное заполнение с левой стороны виджета и 10-пиксельное заполнение с правой стороны. Или же 20-пиксельное заполнение вверху виджета и 8-пиксельное заполнение внизу. В этом случае вы можете использовать следующие общие форматы для аргументов `padx` и `pady` метода `pack()`:

```
padx=(слева, справа)  
pady=(сверху, снизу)
```

Если вы предоставите кортеж из двух целых чисел для аргумента `padx`, то значения кортежа зададут заполнение для левой и правой сторон виджета. Аналогично если вы предоставите кортеж из двух целых чисел для аргумента `pady`, то значения кортежа зададут заполнение для верхней и нижней частей виджета. Например, предположим, что переменная `label` ссылается на виджет `Label`. Следующий фрагмент кода добавляет 10 пикселов внешнего заполнения в левую часть виджета, 5 пикселов справа, 20 пикселов сверху и 10 пикселов снизу:

```
self.label1.pack(padx=(10, 5), pady=(20, 10))
```



ПРИМЕЧАНИЕ

Этот технический прием работает только с **внешним заполнением**. Внутреннее заполнение должно быть однородным.



Контрольная точка

- 13.8. Что делает метод `pack()` виджета?
- 13.9. Как будут расположены виджеты `Label` в их родительском виджете, если создать два виджета `Label` и вызвать их методы `pack()` без аргументов?
- 13.10. Какой аргумент следует передать в метод `pack()` виджета, чтобы он был расположен максимально слева в родительском виджете?

- 13.11. Модифицируйте следующую ниже инструкцию таким образом, чтобы она создавала надпись с границей шириной 3 пикселя и имела рельефный 3D-вид:

```
self.label = tkinter.Label(self.main_window, text='Привет, мир')
```

- 13.12. Модифицируйте следующую ниже инструкцию таким образом, чтобы она упаковывала виджет `my_label` с 10 пикселями горизонтального внутреннего заполнения и 20 пикселями вертикального внутреннего заполнения:

```
self.label1.pack()
```

- 13.13. Модифицируйте следующую ниже инструкцию таким образом, чтобы она упаковывала виджет `my_label` с 10 пикселями горизонтального внешнего заполнения и 20 пикселями вертикального внешнего заполнения:

```
self.label1.pack()
```

- 13.14. Модифицируйте следующую ниже инструкцию таким образом, чтобы она упаковывала виджет `my_label` с 10 пикселями горизонтального внутреннего и внешнего заполнения и 10 пикселями вертикального внутреннего и внешнего заполнения:

```
self.label1.pack()
```

13.4 Упорядочение виджетов с помощью рамок *Frame*

Ключевые положения

Виджет *Frame* является контейнером, который может содержать другие виджеты. Рамки *Frame* применяются для упорядочения виджетов в окне.

Виджет *Frame* является контейнером и может содержать другие виджеты. Рамки *Frame* полезны для упорядочения и размещения групп виджетов в окне. Например, можно разместить набор виджетов внутри одной рамки *Frame* и расположить их определенным способом, затем разместить набор виджетов в другой рамке *Frame* и расположить их по-другому. Программа 13.9 это демонстрирует. При ее запуске на экран выводится окно, как на рис. 13.18.

Программа 13.9 (frame_demo.py)

```
1 # Эта программа создает надписи в двух разных рамках.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Создать виджет главного окна.
8         self.main_window = tkinter.Tk()
9
10        # Создать две рамки: одну для верхней части окна,
11        # другую для нижней части.
12        self.top_frame = tkinter.Frame(self.main_window)
13        self.bottom_frame = tkinter.Frame(self.main_window)
14
15        # Создать три виджета Label
16        # для верхней рамки.
17        self.label1 = tkinter.Label(self.top_frame,
18                                text='Мигнуть')
19        self.label2 = tkinter.Label(self.top_frame,
20                                text='Моргнуть')
21        self.label3 = tkinter.Label(self.top_frame,
22                                text='Кивнуть')
23
24        # Упаковать надписи, расположенные в верхней рамке.
25        # Применить аргумент side='top', чтобы их
26        # расположить одну под другой.
27        self.label1.pack(side='top')
28        self.label2.pack(side='top')
29        self.label3.pack(side='top')
30
31        # Создать три виджета Label
32        # для нижней рамки.
```

```

33         self.label4 = tkinter.Label(self.bottom_frame,
34                             text='Мигнуть')
35         self.label5 = tkinter.Label(self.bottom_frame,
36                             text='Моргнуть')
37         self.label6 = tkinter.Label(self.bottom_frame,
38                             text='Кивнуть')
39
40     # Упаковать надписи, расположенные в нижней рамке.
41     # Применить аргумент side='left', чтобы их
42     # расположить горизонтально слева в рамке.
43     self.label4.pack(side='left')
44     self.label5.pack(side='left')
45     self.label6.pack(side='left')
46
47     # Да, и мы также должны упаковать рамки!
48     self.top_frame.pack()
49     self.bottom_frame.pack()
50
51     # Войти в главный цикл tkinter.
52     tkinter.mainloop()
53
54 # Создать экземпляр класса MyGUI.
55 if __name__ == '__main__':
56     my_gui = MyGUI()

```

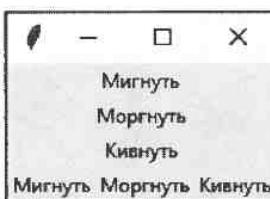


РИС. 13.18. Окно, выводимое на экран программой 13.9

Взглянем на строки 12 и 13:

```

self.top_frame = tkinter.Frame(self.main_window)
self.bottom_frame = tkinter.Frame(self.main_window)

```

Эти строки программы создают два объекта Frame. Аргумент `self.main_window`, который появляется внутри круглых скобок, добавляет рамки Frame в виджет `main_window`.

Строки 17–22 создают три виджета Label. Обратите внимание, что эти виджеты добавляются в виджет `self.top_frame`. Затем строки 27–29 вызывают метод `pack()` каждого виджета Label, передавая `side='top'` в качестве аргумента. Это приводит к тому, что три виджета выводятся один под другим внутри рамки Frame (см. рис. 13.18).

Строки 33–38 создают еще три виджета Label, которые добавляются в виджет `self.bottom_frame`. Затем строки 43–45 вызывают метод `pack()` каждого виджета Label,

передавая `side='left'` в качестве аргумента. Это приводит к тому, что три виджета появятся в рамке `Frame` в горизонтальном положении (рис. 13.9).

Строки 48 и 49 вызывают метод `pack()` виджета `Frame`, который делает видимыми виджет `Frame`. Стока 52 выполняет функцию `mainloop` модуля `tkinter`.

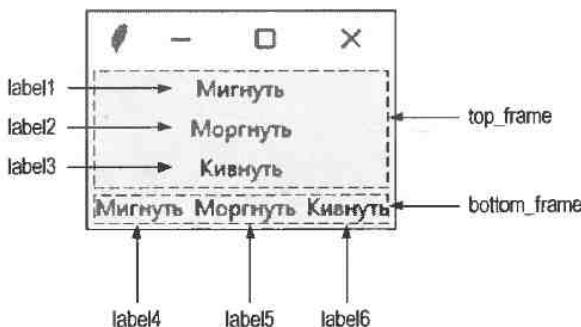


РИС. 13.19. Расстановка виджетов

13.5 Виджеты *Button* и информационные диалоговые окна

Ключевые положения

Виджет `Button` используется для создания в окне стандартной кнопки. Когда пользователь нажимает кнопку, вызываются заданная функция или заданный метод.

Информационное диалоговое окно — это простое окно, оно выводит пользователю сообщение и содержит кнопку **OK**, которая закрывает диалоговое окно. Для вывода информационного диалогового окна используется функция `showinfo` модуля `tkinter.messagebox`.

▶ Видеозапись "Реагирование на нажатие клавиш" (Responding to Button Clicks)

`Button` — это виджет, который пользователь может нажать, чтобы вызвать выполнение действия. При создании виджета `Button` можно определить текст, который должен появиться на поверхности кнопки, и имя функции обратного вызова. **Функция обратного вызова** — это функция или метод, которые исполняются, когда пользователь нажимает кнопку.



ПРИМЕЧАНИЕ

Функция обратного вызова также называется **обработчиком события**, потому что она обрабатывает событие, которое происходит, когда пользователь нажимает кнопку.

В целях демонстрации работы этого виджета рассмотрим программу 13.10. Она выводит на экран окно, показанное на рис. 13.20. Когда пользователь нажимает кнопку, программа выводит на экран отдельное информационное диалоговое окно (рис. 13.21). Для вывода информационного диалогового окна используется функция `showinfo`, которая находится в мо-

дуле `tkinter.messagebox`. (Для использования функции `showinfo` необходимо импортировать модуль `tkinter.messagebox`.) Вот общий формат вызова функции `showinfo`:

```
tkinter.messagebox.showinfo(заголовок, сообщение)
```

В данном формате `заголовок` — это строковый литерал, который выводится на экран в области заголовка диалогового окна, `сообщение` — это строковый литерал с информационным сообщением, которое выводится на экран в главной части диалогового окна.

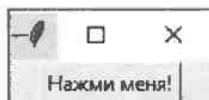


РИС. 13.20. Главное окно, выводимое на экран программой 13.10

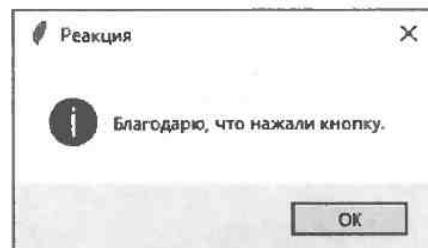


РИС. 13.21. Информационное диалоговое окно, выводимое на экран программой 13.10

Программа 13.10 (button_demo.py)

```
1 # Эта программа демонстрирует виджет Button.
2 # Когда пользователь нажимает кнопку Button,
3 # на экран выводится информационное диалоговое окно.
4
5 import tkinter
6 import tkinter.messagebox
7
8 class MyGUI:
9     def __init__(self):
10         # Создать виджет главного окна.
11         self.main_window = tkinter.Tk()
12
13         # Создать виджет Button.
14         # На кнопке должен появиться текст 'Нажми меня!'.
15         # Когда пользователь нажимает кнопку,
16         # должен быть выполнен метод do_something.
17         self.my_button = tkinter.Button(self.main_window,
18                                         text='Нажми меня!',
19                                         command=self.do_something)
20
21         # Упаковать виджет Button.
22         self.my_button.pack()
23
24         # Войти в главный цикл tkinter.
25         tkinter.mainloop()
26
```

```

27     # Метод do_something является функцией обратного
28     # вызова для виджета Button.
29
30     def do_something(self):
31         # Показать информационное диалоговое окно.
32         tkinter.messagebox.showinfo('Реакция',
33                                     'Благодарю, что нажали кнопку.')
34
35 # Создать экземпляр класса MyGUI.
36 if __name__ == '__main__':
37     my_gui = MyGUI()

```

Строка 5 импортирует модуль `tkinter`, а строка 6 — модуль `tkinter.messagebox`. Стока 11 создает корневой виджет и присваивает его переменной `main_window`.

Инструкция в строках 17–19 создает виджет `Button`. Первым аргументом внутри круглых скобок `self.main_window` является родительский виджет. Аргумент `text='Нажми меня!'` определяет, что строковый литерал 'Нажми меня!' должен появиться на поверхности кнопки. Аргумент `command='self.do_something'` задает метод `do_something()` класса в качестве функции обратного вызова. Когда пользователь нажмет кнопку, исполнится метод `do_something()`.

Метод `do_something()` расположен в строках 30–33. Он просто вызывает функцию `tkinter.messagebox.showinfo` для вывода на экран информационного окна, показанного на рис. 13.21. Для того чтобы закрыть это диалоговое окно, пользователь должен нажать кнопку **OK**.

Создание кнопки выхода из программы

Программы с GUI обычно имеют кнопку **Выйти** (или кнопку **Отмена**), которая закрывает программу, когда пользователь ее нажимает. Для кнопки **Выйти** в программе Python нужно просто создать виджет `Button`, который в качестве функции обратного вызова вызывает метод `destroy()` корневого виджета. Программа 13.11 демонстрирует, как это делается. Она представляет собой видоизмененную версию программы 13.10, в которую добавлен второй виджет `Button` (рис. 13.22).

Программа 13.11 (quit_button.py)

```

1 # Эта программа содержит кнопку 'Выйти', которая
2 # при ее нажатии вызывает метод destroy класса Tk.
3
4 import tkinter
5 import tkinter.messagebox
6
7 class MyGUI:
8     def __init__(self):
9         # Создать виджет главного окна.
10        self.main_window = tkinter.Tk()
11

```

```

12     # Создать виджет Button.
13     # На кнопке должен появиться текст 'Нажми меня!'.
14     # Когда пользователь нажимает кнопку,
15     # должен быть выполнен метод do_something.
16     self.my_button = tkinter.Button(self.main_window,
17                                     text='Нажми меня!',
18                                     command=self.do_something)
19
20     # Создать кнопку 'Выйти'. При нажатии этой кнопки вызывается
21     # метод destroy корневого виджета (переменная
22     # main_window ссылается на корневой виджет, поэтому функцией
23     # обратного вызова является self.main_window.destroy.)
24     self.quit_button = tkinter.Button(self.main_window,
25                                     text='Выйти',
26                                     command=self.main_window.destroy)
27
28
29     # Упаковать виджеты Button.
30     self.my_button.pack()
31     self.quit_button.pack()
32
33     # Войти в главный цикл tkinter.
34     tkinter.mainloop()
35
36     # Метод do_something является функцией обратного
37     # вызова для виджета Button.
38
39     def do_something(self):
40         # Показать информационное диалоговое окно.
41         tkinter.messagebox.showinfo('Реакция',
42                                     'Благодарю, что нажали кнопку.')
43
44 # Создать экземпляр класса MyGUI.
45 if __name__ == '__main__':
46     my_gui = MyGUI()

```

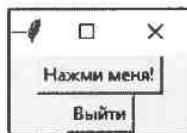


РИС. 13.22. Информационное диалоговое окно, выводимое на экран программой 13.11

Инструкция в строках 24–26 создает кнопку **Выйти**. Обратите внимание, что метод `self.main_window.destroy()` используется в качестве функции обратного вызова. Когда пользователь нажимает кнопку, вызывается этот метод, и программа завершает работу.

13.6 Получение входных данных с помощью виджета *Entry*

Ключевые положения

Виджет *Entry* — это прямоугольная область, в которую пользователь может вводить входные данные. Для извлечения данных, введенных в виджет *Entry*, предназначен его метод *get()*.

Виджет *Entry* — это прямоугольная область, в которую пользователь может вводить текст. Его используют для сбора входных данных в программе с GUI. Как правило, программа будет иметь в окне один или несколько виджетов *Entry* вместе с кнопкой, которую пользователь нажимает для передачи данных, введенных в элементе *Entry*. Функция обратного вызова кнопки получает данные из элемента *Entry* окна и обрабатывает их.

Для извлечения данных, введенных пользователем в виджет *Entry*, применяется его метод *get()*. Метод возвращает строковое значение, поэтому такое значение необходимо привести к надлежащему типу данных, если, к примеру, виджет *Entry* используется для ввода чисел.

В целях демонстрации его работы мы рассмотрим программу, которая предоставляет возможность пользователю вводить в виджет *Entry* расстояние в километрах и затем нажимать кнопку, чтобы увидеть это расстояние, преобразованное в мили. Вот формула преобразования километров в мили:

$$\text{мили} = \text{километры} \times 0.6214.$$

На рис. 13.23 представлено окно, которое эта программа выводит на экран. Для того чтобы расположить виджеты в позициях, показанных на рисунке, мы разместим их в двух рамках (рис. 13.24). Выводящий подсказку виджет *Label* и виджет *Entry* будут расположены в верхней рамке *top_frame*, и их методы *pack()* будут вызываться с аргументом *side='left'*. В результате этого они появятся в рамке горизонтально. Кнопки **Преобразовать** и **Выйти** будут расположены в нижней рамке *bottom_frame*, и их методы *pack()* тоже будут вызываться с аргументом *side='left'*.

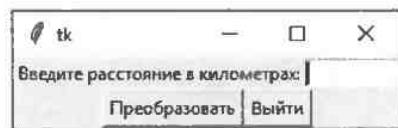


РИС. 13.23. Окно программы *kilo_converter*



РИС. 13.24. Окно, сформированное рамками *Frame*

В программе 13.12 приведен соответствующий код, а на рис. 13.25 показано, что происходит, когда пользователь вводит в виджет *Entry* число 1000 и нажимает кнопку **Преобразовать**.

Программа 13.12 (*kilo_converter.py*)

```
1 # Эта программа конвертирует расстояния в километрах
2 # в мили. Полученный результат выводится
3 # в информационном диалоговом окне.
```

```
4
5 import tkinter
6 import tkinter.messagebox
7
8 class KiloConverterGUI:
9     def __init__(self):
10
11         # Создать главное окно.
12         self.main_window = tkinter.Tk()
13
14         # Создать две рамки, чтобы сгруппировать виджеты.
15         self.top_frame = tkinter.Frame(self.main_window)
16         self.bottom_frame = tkinter.Frame(self.main_window)
17
18         # Создать виджеты для верхней рамки.
19         self.prompt_label = tkinter.Label(self.top_frame,
20                                         text='Введите расстояние в километрах:')
21         self.kilo_entry = tkinter.Entry(self.top_frame,
22                                         width=10)
23
24         # Упаковать виджеты верхней рамки.
25         self.prompt_label.pack(side='left')
26         self.kilo_entry.pack(side='left')
27
28         # Создать виджеты Button для нижней рамки.
29         self.calc_button = tkinter.Button(self.bottom_frame,
30                                         text='Преобразовать',
31                                         command=self.convert)
32         self.quit_button = tkinter.Button(self.bottom_frame,
33                                         text='Выйти',
34                                         command=self.main_window.destroy)
35
36         # Упаковать кнопки.
37         self.calc_button.pack(side='left')
38         self.quit_button.pack(side='left')
39
40         # Упаковать рамки.
41         self.top_frame.pack()
42         self.bottom_frame.pack()
43
44         # Войти в главный цикл tkinter.
45         tkinter.mainloop()
46
47         # Метод convert является функцией обратного вызова
48         # для кнопки 'Преобразовать'.
```

```

49  def convert(self):
50      # Получить значение, введенное пользователем
51      # в виджет kilo_entry.
52      kilo = float(self.kilo_entry.get())
53
54      # Конвертировать километры в мили.
55      miles = kilo * 0.6214
56
57      # Показать результаты в информационном диалоговом окне.
58      tkinter.messagebox.showinfo('Результаты',
59                                  str(kilo) +
60                                  ' километров эквивалентно ' +
61                                  str(miles) + ' милям.')
62
63 # Создать экземпляр класса KiloConverterGUI.
64 if __name__ == '__main__':
65     kilo_conv = KiloConverterGUI()

```

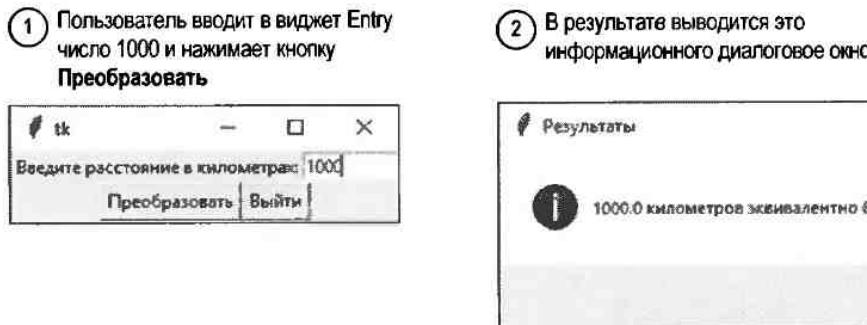


РИС. 13.25. Информационное диалоговое окно

Метод `convert()`, показанный в строках 49–60, является функцией обратного вызова кнопки **Преобразовать**. Инструкция в строке 52 вызывает метод `get()` элемента `kilo_entry`, чтобы извлечь данные, которые были введены в этот виджет. Полученное значение приводится к вещественному типу `float` и затем присваивается переменной `kilo`. Вычисление в строке 55 выполняет преобразование, и полученный результат присваивается переменной `miles`. Затем инструкция в строках 58–61 выводит информационное диалоговое окно с сообщением, которое показывает конвертированное значение.

13.7

Применение виджетов *Label* в качестве полей вывода

Ключевые положения

Когда объект `StringVar` связан с виджетом `Label`, виджет `Label` выводит на экран любые данные, которые хранятся в объекте `StringVar`.

Ранее вы видели применение информационного диалогового окна для отображения выходных данных. Если вы не хотите показывать отдельное диалоговое окно для выходных данных своей программы, то для их динамического отображения имеется возможность использовать виджет `Label` в главном окне программы. В этом случае в главном окне просто создаются пустые виджеты `Label`, а затем пишется программный код, который при нажатии кнопки выводит в этих виджетах требуемые данные.

Модуль `tkinter` предоставляет класс `StringVar`, который может использоваться вместе с виджетом `Label` для вывода данных. Сначала нужно создать объект `StringVar`. Затем создать виджет `Label` и связать его с объектом `StringVar`. С этого момента любое значение, которое потом сохраняется в объекте `StringVar`, будет автоматически выводиться на экран в виджет `Label`.

Программа 13.13 демонстрирует, как это делается. Она представляет собой видоизмененную версию `kilo_converter`, которую вы видели в программе 13.12. Вместо вывода информационного диалогового окна данная версия программы выводит количество миль в виджет `Label` в главном окне.

Программа 13.13 (kilo_converter2.py)

```
1 # Эта программа конвертирует расстояния в километрах
2 # в мили. Полученный результат выводится
3 # в виджет Label в главном окне.
4
5 import tkinter
6
7 class KiloConverterGUI:
8     def __init__(self):
9
10         # Создать главное окно.
11         self.main_window = tkinter.Tk()
12
13         # Создать три рамки, чтобы сгруппировать виджеты.
14         self.top_frame = tkinter.Frame()
15         self.mid_frame = tkinter.Frame()
16         self.bottom_frame = tkinter.Frame()
17
18         # Создать виджеты для верхней рамки.
19         self.prompt_label = tkinter.Label(self.top_frame,
20                                         text='Введите расстояние в километрах:')
21         self.kilo_entry = tkinter.Entry(self.top_frame,
22                                         width=10)
23
24         # Упаковать виджеты верхней рамки.
25         self.prompt_label.pack(side='left')
26         self.kilo_entry.pack(side='left')
27
```

```
28     # Создать виджеты для средней рамки.
29     self.descr_label = tkinter.Label(self.mid_frame,
30                                     text='Преобразовано в мили:')
31
32     # Объект StringVar нужен для того, чтобы его связать
33     # с выходной надписью. Для сохранения последовательности
34     # пробелов используется метод set данного объекта.
35     self.value = tkinter.StringVar()
36
37     # Создать надпись Label и связать ее с объектом
38     # StringVar. Любые значения, хранящиеся
39     # в объекте StringVar, будут автоматически
40     # выводиться в надписи Label.
41     self.miles_label = tkinter.Label(self.mid_frame,
42                                     textvariable=self.value)
43
44     # Создать виджеты для средней рамки.
45     self.descr_label.pack(side='left')
46     self.miles_label.pack(side='left')
47
48     # Создать виджеты Button для нижней рамки.
49     self.calc_button = tkinter.Button(self.bottom_frame,
50                                     text='Преобразовать',
51                                     command=self.convert)
52     self.quit_button = tkinter.Button(self.bottom_frame,
53                                     text='Выйти',
54                                     command=self.main_window.destroy)
55
56     # Упаковать кнопки.
57     self.calc_button.pack(side='left')
58     self.quit_button.pack(side='left')
59
60     # Упаковать рамки.
61     self.top_frame.pack()
62     self.mid_frame.pack()
63     self.bottom_frame.pack()
64
65     # Войти в главный цикл tkinter.
66     tkinter.mainloop()
67
68     # Метод convert является функцией обратного вызова
69     # для кнопки 'Преобразовать'.
70
71     def convert(self):
72         # Получить значение, введенное
73         # пользователем в виджет kilo_entry.
74         kilo = float(self.kilo_entry.get())
```

```

75
76     # Конвертировать километры в мили.
77     miles = kilo * 0.6214
78
79     # Конвертировать мили в строковое значение
80     # и сохранить ее в объекте StringVar. В результате
81     # виджет miles_label будет автоматически обновлен.
82     self.value.set(miles)
83
84 # Создать экземпляр класса KiloConverterGUI.
85 if __name__ == '__main__':
86     kilo_conv = KiloConverterGUI()

```

При запуске эта программа выводит окно, как на рис. 13.26. На рис. 13.27 показано, что происходит, когда пользователь вводит 1000 км и нажимает кнопку **Преобразовать**. Количество миль выводится в главном окне в надписи **Label**.

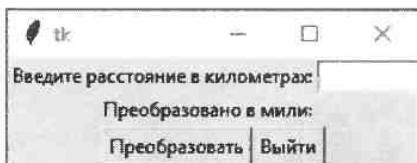


РИС. 13.26. Окно, выводимое на экран при запуске программы

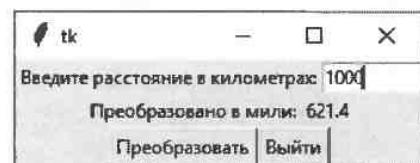


РИС. 13.27. Окно, показывающее результат преобразования 1000 км в мили

Давайте разберем этот программный код. Строки 14–16 создают три рамки: `top_frame`, `mid_frame` и `bottom_frame`. Строки 19–26 создают виджеты для верхней рамки и вызывают их метод `pack()`.

Строки 29–30 создают виджет `Label` с текстом 'Преобразовано в мили:', который вы видите в главном окне на рис. 13.26. Затем строка 35 создает объект `StringVar` и присваивает его переменной `value`. Стока 41 создает виджет `Label` с именем `miles_label`, который мы будем использовать для вывода количества миль. Отметим, что в строке 42 указан аргумент `textvariable=self.value`. Он создает связь между виджетом `Label` и объектом `StringVar`, на который ссылается переменная `value`. Любое значение, которое хранится в объекте `StringVar`, будет выведено на экран в этом виджете `Label`.

Строки 45 и 46 упаковывают два виджета `Label`, которые находятся в рамке `mid_frame`. Строки 49–58 создают виджеты `Button` и упаковывают их. Строки 61–63 упаковывают объекты `Frame`. На рис. 13.28 показано, каким образом размещаются различные виджеты в трех рамках этого окна.

Метод `convert()` в строках 71–82 представляет собой функцию обратного вызова кнопки **Преобразовать**. Инструкция в строке 74 вызывает метод `get()` виджета `kilo_entry`, чтобы извлечь значение, которое было введено в этот виджет. Это значение приводится к вещественному типу `float` и затем присваивается переменной `kilo`. Выражение в строке 77 выполняет преобразование и присваивает полученный результат переменной `miles`. Затем

инструкция в строке 82 вызывает метод `set()` объекта `StringVar`, передавая `miles` в качестве аргумента. В результате значение, на которое ссылается переменная `miles`, сохраняется в объекте `StringVar` и одновременно выводится в виджет `miles_label`.

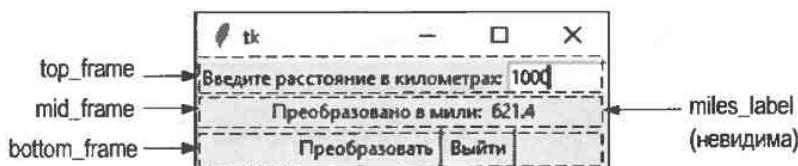


РИС. 13.28. Макет главного окна программы `kilo_converter2`

В ЦЕНТРЕ ВНИМАНИЯ



Создание программы с GUI

Кэтрин — преподаватель. В главе 3 мы подробно изучили разработку программы, которую ее студенты могут применять для вычисления среднего балла из трех оценок за контрольные работы. Программа предлагает студенту ввести все оценки и затем показывает средний балл. Кэтрин попросила вас разработать программу с GUI, которая выполняет аналогичную работу. Она хотела бы, чтобы программа имела три виджета `Entry`, в которые можно вводить оценки, и кнопку, которая при ее нажатии выводит на экран средний балл.

Прежде чем приступить к написанию программного кода, полезно нарисовать эскиз окна программы (рис. 13.29). Эскиз показывает тип каждого виджета. (Нумерация на эскизе поможет при составлении списка всех виджетов.)



РИС. 13.29. Эскиз окна

Изучив этот эскиз, мы можем составить список виджетов, в которых нуждаемся (табл. 13.3). При составлении списка мы включим краткое описание каждого виджета и имя, которое присвоим каждому виджету при его создании.

Из эскиза видно, что в окне имеется пять строк виджетов. Для их размещения мы также создадим пять объектов `Frame`. На рис. 13.30 представлена расстановка виджетов в пяти объектах `Frame`.

Таблица 13.3. Виджеты к задаче о среднем балле

№ виджета на рис. 13.29	Тип элемента	Описание	Имя
1	Label	Предлагает пользователю ввести оценку 1	test1_label
2	Label	Предлагает пользователю ввести оценку 2	test2_label
3	Label	Предлагает пользователю ввести оценку 3	test3_label
4	Label	Идентифицирует средний балл, который впоследствии будет выведен к этой надписи	result_label
5	Entry	Место, где пользователь вводит оценку 1	test1_entry
6	Entry	Место, где пользователь вводит оценку 2	test2_entry
7	Entry	Место, где пользователь вводит оценку 3	test3_entry
8	Label	Программа покажет средний балл в этой надписи	avg_label
9	Button	При нажатии этой кнопки программа вычислит средний балл и покажет его в компоненте averageLabel	calc_button
10	Button	При нажатии этой кнопки программа завершит работу	quit_button

Программа 13.11 содержит соответствующий код, а на рис. 13.31 показано окно с введенными пользователем данными.

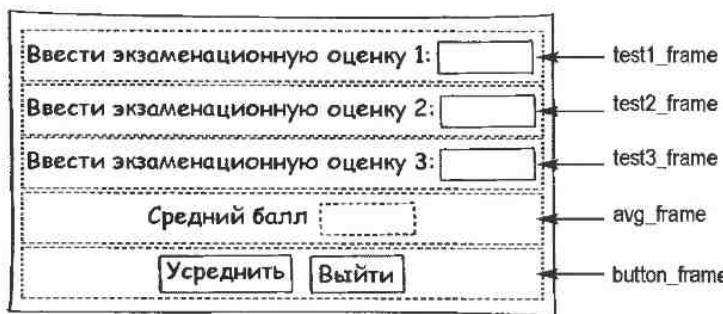


РИС. 13.30. Применение рамок Frame для упорядочения виджетов

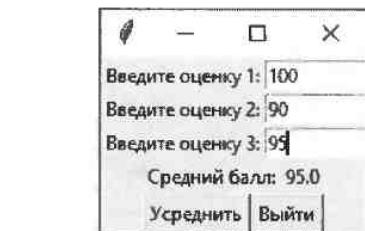


РИС. 13.31. Окно программы test_averages

Программа 13.14 (test_averages.py)

```

1 # Эта программа применяет GUI для получения трех
2 # оценок и вывода среднего балла.
3
4 import tkinter
5
6 class TestAvg:
7     def __init__(self):
8         # Создать главное окно.
9         self.main_window = tkinter.Tk()
10

```

```
11  # Создать пять рамок.
12  self.test1_frame = tkinter.Frame(self.main_window)
13  self.test2_frame = tkinter.Frame(self.main_window)
14  self.test3_frame = tkinter.Frame(self.main_window)
15  self.avg_frame = tkinter.Frame(self.main_window)
16  self.button_frame = tkinter.Frame(self.main_window)
17
18  # Создать и упаковать виджеты для оценки 1.
19  self.test1_label = tkinter.Label(self.test1_frame,
20          text='Введите оценку 1:')
21  self.test1_entry = tkinter.Entry(self.test1_frame,
22          width=10)
23  self.test1_label.pack(side='left')
24  self.test1_entry.pack(side='left')
25
26  # Создать и упаковать виджеты для оценки 2.
27  self.test2_label = tkinter.Label(self.test2_frame,
28          text='Введите оценку 2:')
29  self.test2_entry = tkinter.Entry(self.test2_frame,
30          width=10)
31  self.test2_label.pack(side='left')
32  self.test2_entry.pack(side='left')
33
34  # Создать и упаковать виджеты для оценки 3.
35  self.test3_label = tkinter.Label(self.test3_frame,
36          text='Введите оценку 3:')
37  self.test3_entry = tkinter.Entry(self.test3_frame,
38          width=10)
39  self.test3_label.pack(side='left')
40  self.test3_entry.pack(side='left')
41
42  # Создать и упаковать виджеты для среднего балла.
43  self.result_label = tkinter.Label(self.avg_frame,
44          text='Средний балл:')
45  self.avg = tkinter.StringVar() # Для обновления avg_label
46  self.avg_label = tkinter.Label(self.avg_frame,
47          textvariable=self.avg)
48  self.result_label.pack(side='left')
49  self.avg_label.pack(side='left')
50
51  # Создать и упаковать виджеты Button.
52  self.calc_button = tkinter.Button(self.button_frame,
53          text='Усреднить',
54          command=self.calc_avg)
55  self.quit_button = tkinter.Button(self.button_frame,
56          text='Выйти',
57          command=self.main_window.destroy)
```

```
58     self.calc_button.pack(side='left')
59     self.quit_button.pack(side='left')
60
61     # Упаковать рамки.
62     self.test1_frame.pack()
63     self.test2_frame.pack()
64     self.test3_frame.pack()
65     self.avg_frame.pack()
66     self.button_frame.pack()
67
68     # Запустить главный цикл.
69     tkinter.mainloop()
70
71 # Метод calc_avg является функцией обратного вызова
72 # для виджета calc_button.
73
74 def calc_avg(self):
75     # Получить три оценки
76     # и сохранить их в переменных.
77     self.test1 = float(self.test1_entry.get())
78     self.test2 = float(self.test2_entry.get())
79     self.test3 = float(self.test3_entry.get())
80
81     # Вычислить средний балл.
82     self.average = (self.test1 + self.test2 +
83     self.test3) / 3.0
84
85     # Обновить виджет avg_label,
86     # сохранив значение self.average в объекте
87     # StringVar, на который ссылается avg.
88     self.avg.set(self.average)
89
90 # Создать экземпляр класса TestAvg.
91 if __name__ == '__main__':
92     test_avg = TestAvg()
```



Контрольная точка

13.15. Как извлечь данные из виджета Entry?

13.16. Какой тип данных имеет значение, извлекаемое из виджета Entry?

13.17. В каком модуле находится класс StringVar?

13.18. Что получится, если установить связь объекта StringVar с виджетом Label?

13.8 Радиокнопки и флаговые кнопки

Ключевые положения

Радиокнопки обычно появляются в группах из двух или нескольких кнопок и позволяют пользователю выбирать один из нескольких возможных вариантов. Флаговые кнопки, которые могут появляться самостоятельно или в группах, обеспечивают выбор в формате да/нет или включено/выключено.

Радиокнопки

Радиокнопки, или просто *переключатели*, полезны, когда нужно, чтобы пользователь выбрал один из нескольких возможных вариантов. На рис. 13.32 показано окно, содержащее группу радиокнопок. Каждая радиокнопка имеет кружок и может иметь два состояния: быть выбранной или невыбранной. Когда радиокнопка выбрана, кружок заполнен, а когда радиокнопка не выбрана, он пустой.

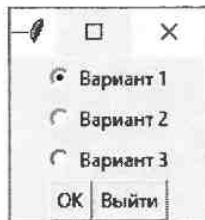


РИС. 13.32. Группа радиокнопок

Для создания виджетов Radiobutton используется класс Radiobutton модуля tkinter. Одновременно можно выбрать только один виджет Radiobutton в контейнере, в частности в рамке Frame. Нажатие на Radiobutton приводит к его выбору и автоматическому сбросу выбора любой другой кнопки Radiobutton в том же контейнере. Поскольку в контейнере одновременно можно выбрать только одну кнопку Radiobutton, нередко их называют *взаимоисключающими*.

ПРИМЕЧАНИЕ

Название "радиокнопка" имеет отношение к старым автомобильным радиоприемникам, в которых имелись кнопки для выбора станций. В таких радиоприемниках за один раз можно было нажимать всего одну кнопку. При нажатии на кнопку он автоматически выталкивал любую другую нажатую кнопку.

Модуль tkinter предоставляет класс IntVar, который используется вместе с виджетами Radiobutton. При создании группы кнопок Radiobutton все они связываются с одним и тем же объектом IntVar. Помимо этого, каждому виджету Radiobutton необходимо присвоить уникальное целочисленное значение. Когда выбирается один из виджетов Radiobutton, он сохраняет свое уникальное целочисленное значение в объекте IntVar.

Программа 13.15 демонстрирует способ создания и применения виджетов Radiobutton. На рис. 13.33 представлено окно, которое эта программа выводит на экран. Когда пользователь нажимает кнопку OK, появляется информационное диалоговое окно, сообщающее о номере выбранного виджета Radiobutton.

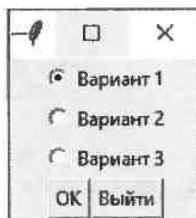


РИС. 13.33. Окно, выводимое на экран программой 13.15

Программа 13.15 (radiobutton_demo.py)

```

1 # Эта программа демонстрирует группу виджетов Radiobutton.
2 import tkinter
3 import tkinter.messagebox
4
5 class MyGUI:
6     def __init__(self):
7         # Создать главное окно.
8         self.main_window = tkinter.Tk()
9
10        # Создать две рамки: одну для виджетов Radiobutton
11        # и еще одну для обычных виджетов Button.
12        self.top_frame = tkinter.Frame(self.main_window)
13        self.bottom_frame = tkinter.Frame(self.main_window)
14
15        # Создать объект IntVar для использования
16        # с виджетами Radiobutton.
17        self.radio_var = tkinter.IntVar()
18
19        # Назначить объекту IntVar значение 1.
20        self.radio_var.set(1)
21
22        # Создать виджеты Radiobutton в рамке top_frame.
23        self.rbl = tkinter.Radiobutton(self.top_frame,
24                                       text='Вариант 1',
25                                       variable=self.radio_var,
26                                       value=1)
27        self.rb2 = tkinter.Radiobutton(self.top_frame,
28                                       text='Вариант 2',
29                                       variable=self.radio_var,
30                                       value=2)
31        self.rb3 = tkinter.Radiobutton(self.top_frame,
32                                       text='Вариант 3',
33                                       variable=self.radio_var,
34                                       value=3)

```

```
36     # Упаковать виджеты Radiobutton.
37     self.rb1.pack()
38     self.rb2.pack()
39     self.rb3.pack()
40
41     # Создать кнопку 'OK' и кнопку 'Выйти'.
42     self.ok_button = tkinter.Button(self.bottom_frame,
43                                     text='OK',
44                                     command=self.show_choice)
45     self.quit_button = tkinter.Button(self.bottom_frame,
46                                     text='Выйти',
47                                     command=self.main_window.destroy)
48
49     # Упаковать виджеты Button.
50     self.ok_button.pack(side='left')
51     self.quit_button.pack(side='left')
52
53     # Упаковать рамки.
54     self.top_frame.pack()
55     self.bottom_frame.pack()
56
57     # Запустить главный цикл.
58     tkinter.mainloop()
59
60 # Метод show_choice является функцией обратного вызова
61 # для кнопки OK.
62 def show_choice(self):
63     tkinter.messagebox.showinfo('Выбор', 'Выбран вариант ' +
64                               str(self.radio_var.get()))
65
66 # Создать экземпляр класса MyGUI.
67 if __name__ == '__main__':
68     my_gui = MyGUI()
```

Строка 17 создает объект `IntVar` с именем `radio_var`. Стока 20 вызывает метод `set()` объекта `radio_var`, чтобы в этом объекте сохранить целочисленное значение 1. (Вы вскоре увидите, зачем это нужно.)

Строки 23–26 создают первый виджет `Radiobutton`. Аргумент `variable=self.radio_var` (в строке 25) связывает этот виджет `Radiobutton` с объектом `radio_var`. Аргумент `value=1` (в строке 26) присваивает этому виджету `Radiobutton` целое число 1. В результате всегда, когда этот виджет `Radiobutton` будет выбираться, в объекте `radio_var` будет сохраняться значение 1.

Строки 27–30 создают второй виджет `Radiobutton`. Обратите внимание, что этот виджет `Radiobutton` тоже связан с объектом `radio_var`. Аргумент `value=2` (в строке 30) присваивает этому виджету `Radiobutton` целое число 2. В результате всегда, когда этот виджет `Radiobutton` будет выбираться, в объекте `radio_var` будет сохраняться значение 2.

Строки 31–34 создают третий виджет `RadioButton`. Этот виджет `RadioButton` тоже связан с объектом `radio_var`. Аргумент `value=3` (в строке 34) присваивает этому виджету `RadioButton` целое число 3. В результате всегда, когда этот виджет `RadioButton` будет выбираться, в объекте `radio_var` будет сохраняться значение 3.

Метод `show_choice()` в строках 62–64 является функцией обратного вызова для кнопки **OK**. При выполнении этого метода он вызывает метод `get()` объекта `radio_var`, чтобы извлечь хранящееся в объекте значение. Это значение выводится на экран в информационном диалоговом окне.

Вы заметили, что при запуске программы первоначально выбран первый виджет `RadioButton`? Это вызвано тем, что в строке 20 мы присвоили объекту `radio_var` значение 1. Объект `radio_var` используется не только для определения номера выбранного виджета `RadioButton`. Он также нужен для предварительного выбора того или иного виджета `RadioButton`. Когда мы сохраняем значение отдельно взятого виджета `RadioButton` в объекте `radio_var`, этот виджет `RadioButton` становится выбранным.

Использование функций обратного вызова с радиокнопками

Прежде чем программа 13.15 определит, какой виджет `RadioButton` был выбран, она ждет, когда пользователь нажмет кнопку **OK**. Если нужно, можете указывать функцию обратного вызова с виджетами `RadioButton`. Вот пример:

```
self.rbl = tkinter.Radiobutton(self.top_frame,
                               text='Вариант 1',
                               variable=self.radio_var,
                               value=1,
                               command=self.my_method)
```

В этом фрагменте кода применен аргумент `command=self.my_method`, который сообщает, что метод `my_method()` является функцией обратного вызова. Метод `my_method()` будет выполнен сразу после того, как виджет `RadioButton` будет выбран.

Флаговые кнопки

Флаговая кнопка, или *флажок*, представляет собой небольшое поле с надписью рядом. Окно, показанное на рис. 13.34, имеет три флаговые кнопки.

Подобно радиокнопкам, флаговые кнопки могут иметь два состояния: быть выбранными либо невыбранными. При нажатии флаговой кнопки в соответствующем поле появляется

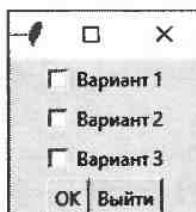


РИС. 13.34. Группа флаговых кнопок

маленькая галочка. Несмотря на то что флаговые кнопки нередко выводятся в группах, они не используются для создания взаимоисключающих вариантов выбора. Вместо этого пользователю разрешается выбрать любые флаговые кнопки (одну или несколько одновременно), которые показаны в группе.

Для создания виджетов Checkbutton нужен класс Checkbutton модуля tkinter. Как и с виджетами Radiobutton, вместе с виджетом Checkbutton используется объект IntVar. Но, в отличие от виджета Radiobutton, с каждым виджетом Checkbutton связывается отдельный объект IntVar. При выборе виджета Checkbutton связанный с ним объект IntVar будет содержать значение 1. При снятии галочки в поле виджета Checkbutton связанный с ним объект IntVar будет содержать значение 0.

Программа 13.16 демонстрирует способ создания и применения виджетов Checkbutton. На рис. 13.35 показано окно, которое программа выводит на экран. Когда пользователь нажимает кнопку **OK**, появляется информационное диалоговое окно с информацией о номерах выбранных виджетов Checkbutton.

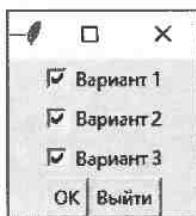


РИС. 13.35. Окно, выводимое программой 13.16

Программа 13.16 (checkbutton_demo.py)

```

1 # Эта программа демонстрирует группу виджетов Checkbutton.
2 import tkinter
3 import tkinter.messagebox
4
5 class MyGUI:
6     def __init__(self):
7         # Создать главное окно.
8         self.main_window = tkinter.Tk()
9
10        # Создать две рамки. Одну для виджетов Checkbutton
11        # и еще одну для обычных виджетов Button.
12        self.top_frame = tkinter.Frame(self.main_window)
13        self.bottom_frame = tkinter.Frame(self.main_window)
14
15        # Создать три объекта IntVar для использования
16        # с виджетами Checkbutton.
17        self.cb_var1 = tkinter.IntVar()
18        self.cb_var2 = tkinter.IntVar()
19        self.cb_var3 = tkinter.IntVar()
20

```

```
21     # Назначить объектам IntVar значения 0.
22     self.cb_var1.set(0)
23     self.cb_var2.set(0)
24     self.cb_var3.set(0)
25
26     # Создать виджеты Checkbutton в рамке top_frame.
27     self.cb1 = tkinter.Checkbutton(self.top_frame,
28                                     text='Вариант 1',
29                                     variable=self.cb_var1)
30     self.cb2 = tkinter.Checkbutton(self.top_frame,
31                                     text='Вариант 2',
32                                     variable=self.cb_var2)
33     self.cb3 = tkinter.Checkbutton(self.top_frame,
34                                     text='Вариант 3',
35                                     variable=self.cb_var3)
36
37     # Упаковать виджеты Checkbutton.
38     self.cb1.pack()
39     self.cb2.pack()
40     self.cb3.pack()
41
42     # Создать кнопку 'OK' и кнопку 'Выход'.
43     self.ok_button = tkinter.Button(self.bottom_frame,
44                                     text='OK',
45                                     command=self.show_choice)
46     self.quit_button = tkinter.Button(self.bottom_frame,
47                                     text='Выход',
48                                     command=self.main_window.destroy)
49
50     # Упаковать виджеты Button.
51     self.ok_button.pack(side='left')
52     self.quit_button.pack(side='left')
53
54     # Упаковать рамки.
55     self.top_frame.pack()
56     self.bottom_frame.pack()
57
58     # Запустить главный цикл.
59     tkinter.mainloop()
60
61     # Метод show_choice является функцией обратного вызова
62     # для кнопки 'OK'.
63
64     def show_choice(self):
65         # Создать строковое значение с сообщением.
66         self.message = 'Вы выбрали:\n'
```

```

68 # Определить, какие виджеты Checkbuttons были выбраны,
69 # и составить соответствующее сообщение.
70 if self.cb_var1.get() == 1:
71     self.message = self.message + '1\n'
72 if self.cb_var2.get() == 1:
73     self.message = self.message + '2\n'
74 if self.cb_var3.get() == 1:
75     self.message = self.message + '3\n'
76
77 # Вывести сообщение в информационном диалоговом окне.
78 tkinter.messagebox.showinfo('Выбор', self.message)
79
80 # Создать экземпляр класса MyGUI.
81 if __name__ == '__main__':
82     my_gui = MyGUI()

```



Контрольная точка

- 13.19. Вы хотите, чтобы пользователь мог выбирать только одно значение из группы значений. Какой тип компонента вы будете использовать для этих значений: радиокнопки или флаговые кнопки?
- 13.20. Вы хотите, чтобы пользователь мог выбирать любое количество значений из группы значений. Какой тип компонента вы будете использовать для этих значений: радиокнопки или флаговые кнопки?
- 13.21. Каким образом используется объект `IntVar` для определения, какой именно виджет `RadioButton` был выбран в группе виджетов `RadioButton`?
- 13.22. Каким образом используется объект `IntVar` для определения, был ли выбран виджет `Checkbutton`?

13.9

Виджеты *Listbox*

Ключевые положения

Виджет `Listbox` выводит на экран список элементов и позволяет пользователю выбирать элемент из этого списка.

Виджет `Listbox` выводит на экран список элементов и позволяет пользователю выбирать один или несколько элементов из этого списка. Программа 13.17 демонстрирует пример. На рис. 13.36 показано окно, выводимое на экран программой.

Программа 13.17 (listbox_example1.py)

```

1 # Эта программа демонстрирует простой виджет Listbox.
2 import tkinter
3

```

```

4 class ListboxExample:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Listbox.
10        self.listbox = tkinter.Listbox(self.main_window)
11        self.listbox.pack(padx=10, pady=10)
12
13        # Заполнить виджет Listbox данными.
14        self.listbox.insert(0, 'Понедельник')
15        self.listbox.insert(1, 'Вторник')
16        self.listbox.insert(2, 'Среда')
17        self.listbox.insert(3, 'Четверг')
18        self.listbox.insert(4, 'Пятница')
19        self.listbox.insert(5, 'Суббота')
20        self.listbox.insert(6, 'Воскресенье')
21
22        # Запустить главный цикл.
23        tkinter.mainloop()
24
25 # Создать экземпляр класса ListboxExample.
26 if __name__ == '__main__':
27     listbox_example = ListboxExample()

```

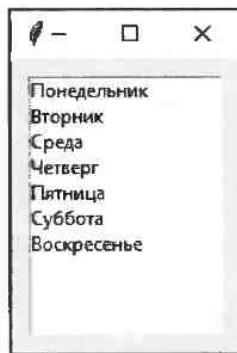


РИС. 13.36. Окно, выводимое на экран программой 13.17

Давайте рассмотрим эту программу подробнее. Стока 10 создает виджет Listbox. Аргумент `self.main_window` внутри скобок является родительским виджетом. Стока 11 вызывает метод `pack()` виджета Listbox с аргументами для отображения 10 пикселов горизонтального и вертикального внешнего заполнения.

Строки 14–20 вставляют элементы в Listbox, вызывая метод `insert()` виджета. Вот инструкция, которая появляется в строке 14:

```
self.listbox.insert(0, 'Понедельник')
```

Первым аргументом является индекс вставляемого элемента. Это просто число, которое определяет положение элемента в списке. Первый элемент имеет индекс 0, следующий — индекс 1 и т. д. Последнее значение индекса равно $n - 1$, где n — число элементов в списке. Вторым аргументом является добавляемый элемент. Таким образом, инструкция в строке 14 вставляет в Listbox строковый литерал 'Понедельник' с индексом 0. Стока 15 вставляет в Listbox строковый литерал 'Вторник' с индексом 1 и т. д.

Задание размера виджета *Listbox*

Каждый элемент, вставленный в список, выводится на экран в одной строке. Высота списка по умолчанию составляет 10 строк, а ширина по умолчанию — 20 символов. При создании списка вы можете задавать разные размеры, как показано ниже:

```
self.listbox = tkinter.Listbox(self.main_window, height=7, width=12)
```

В этом примере аргумент `height=7` приводит к тому, что Listbox будет иметь высоту 7 строк, а аргумент `width=12` — к тому, что Listbox будет иметь ширину 12 символов. Если вы передадите аргумент `height=0`, то высота списка будет ровно такой, какая нужна, чтобы вывести на экран все элементы, содержащиеся в списке. Если вы передадите аргумент `width=0`, то виджет Listbox будет достаточно широким, чтобы вывести на экран самый широкий элемент, который виджет Listbox содержит.

Использование цикла для заполнения виджета *Listbox*

При использовании метода `insert()` для вставки элемента в виджет Listbox можно передать константу `tkinter.END` как индекс, и элемент будет добавлен в конец существующего списка элементов Listbox. Это полезно при использовании цикла для заполнения виджета Listbox, как показано в программе 13.18. Цикл, который появляется в строках 20–21, вставляет все элементы списка дней в виджет Listbox. Когда исполняется инструкция в строке 21, элемент, на который ссылается `day`, вставляется в конец списка элементов виджета Listbox. На рис. 13.37 представлено окно, выводимое на экран программой.

Программа 13.18 (listbox_example2.py)

```
1 # Эта программа демонстрирует простой виджет Listbox.
2 import tkinter
3
4 class ListboxExample:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Listbox.
10        self.listbox = tkinter.Listbox(
11            self.main_window, height=0, width=0)
12        self.listbox.pack(padx=10, pady=10)
13
14        # Создать список с днями недели.
15        days = ['Понедельник', 'Вторник', 'Среда',
```

```

16         'Четверг', 'Пятница', 'Суббота',
17         'Воскресенье']
18
19     # Заполнить виджет Listbox данными.
20     for day in days:
21         self.listbox.insert(tkinter.END, day)
22
23     # Запустить главный цикл.
24     tkinter.mainloop()
25
26 # Создать экземпляр класса ListboxExample.
27 if __name__ == '__main__':
28     listbox_example = ListboxExample()

```

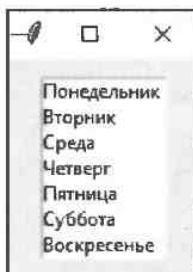


РИС. 13.37. Окно, выводимо на экран программой 13.18

Выбор элементов в виджете *Listbox*

Пользователь может выбирать элемент в списке, щелкнув на нем мышью. Когда элемент выбран, он отображается в списке с выделенным фоном. Виджет *Listbox* имеет четыре разных режима выбора, описанных в табл. 13.3. Некоторые режимы позволяют пользователю выбирать только один элемент за раз, в то время как другие режимы обеспечивают выбор нескольких элементов.

Таблица 13.4. Режимы выбора в виджете *Listbox*

Режим выбора	Описание
<code>tkinter.BROWSE</code>	Пользователь может выбрать по одному элементу за раз, щелкнув в виджете <i>Listbox</i> . Кроме того, если пользователь щелкает и перетаскивает мышь внутри списка, то будет выбран элемент, который в данный момент находится под курсором
<code>tkinter.EXTENDED</code>	Пользователь может выбрать группу соседних элементов, щелкнув по первому элементу и перетащив мышь к последнему элементу
<code>tkinter.MULTIPLE</code>	Можно выбрать несколько элементов. Когда вы щелкаете по невыбранному элементу, он становится выбранным. Щелкнув по выбранному элементу, вы делаете его невыбранным
<code>tkinter.SINGLE</code>	Пользователь может выбрать по одному элементу за раз, щелкнув в виджете <i>Listbox</i>

По умолчанию используется режим выбора `tkinter.BROWSE`. При создании списка можно выбрать другой режим, передав аргумент `selectmode=selection_mode` при создании экземпляра класса `Listbox`. Вот пример:

```
self.listbox = tkinter.Listbox(self.main_window, selectmode=tkinter.EXTENDED)
```

Извлечение выбранного элемента или элементов

Виджет `Listbox` имеет метод `curselection()`, который возвращает кортеж, содержащий индексы элементов, выбранных в данный момент в списке. Вот некоторые важные моменты, которые следует помнить о кортеже, возвращаемом из указанного метода.

- ◆ Если в списке не выбран ни один элемент, то кортеж будет пустым.
- ◆ Если выбран элемент и используется режим выбора `tkinter.BROWSE` или `tkinter.SINGLE`, то кортеж будет содержать только один элемент, поскольку эти режимы выбора позволяют пользователю выбирать только по одному элементу за раз.
- ◆ Если выбран один элемент или несколько элементов и используется режим выбора `tkinter.EXTENDED` или `tkinter.MULTIPLE`, то кортеж может содержать несколько элементов, поскольку эти режимы выбора позволяют пользователю выбирать несколько элементов.

Имея кортеж, возвращаемый методом `curselection()`, вы можете использовать метод `get()` виджета `Listbox` для извлечения выбранного элемента или элементов из данного виджета. Метод `get()` принимает целочисленный индекс в качестве аргумента и возвращает элемент, расположенный в этом индексе в виджете `Listbox`. Например, предположим, что в приведенном ниже фрагменте кода переменная `self.listbox` ссылается на виджет `Listbox`:

```
indexes = self.listbox.curselection()
for i in indexes:
    tkinter.messagebox.showinfo(self.listbox.get(i))
```

В этом фрагменте кода вызывается метод `curselection()`, и возвращаемый кортеж присваивается переменной `indexes`. Затем цикл `for` прокручивает элементы кортежа, используя `i` в качестве целевой переменной. В цикле метод `get()` виджета `Listbox` вызывается с аргументом `i` в качестве аргумента. Возвращаемое значение выводится в диалоговом окне.

В программе 13.19 приведен полный исходный код, демонстрирующий процедуру получения выбранного элемента из виджета `Listbox`. Пользователь выбирает элемент, а затем нажимает кнопку **Получить элемент**. Выбранный элемент извлекается из списка и выводится в диалоговом окне. На рис. 13.38 показано окно программы и диалоговое окно, в котором выводится элемент, выбранный пользователем из виджета `Listbox`.

Программа 13.19 (dog_listbox.py)

```
1 # Эта программа получает выбранный пользователем вариант из виджета Listbox.
2 import tkinter
3 import tkinter.messagebox
4
5 class ListBoxSelection:
6     def __init__(self):
7         # Создать главное окно.
8         self.main_window = tkinter.Tk()
9
```

```

10     # Создать виджет Listbox.
11     self.dog_listbox = tkinter.Listbox(
12         self.main_window, width=0, height=0)
13     self.dog_listbox.pack(padx=10, pady=5)
14
15     # Создать список с названиями пород собак.
16     dogs = ['Лабрадор', 'Пудель', 'Дог', 'Терьер']
17
18     # Заполнить виджет Listbox содержимым списка.
19     for dog in dogs:
20         self.dog_listbox.insert(tkinter.END, dog)
21
22     # Создать кнопку, чтобы получать выбранный элемент.
23     self.get_button = tkinter.Button(
24         self.main_window, text='Получить элемент',
25         command=self.__retrieve_dog)
26     self.get_button.pack(padx=10, pady=5)
27
28     # Запустить главный цикл.
29     tkinter.mainloop()
30
31 def __retrieve_dog(self):
32     # Получить индекс выбранного элемента.
33     indexes = self.dog_listbox.curselection()
34
35     # Если элемент был выбран, то показать его.
36     if (len(indexes) > 0):
37         tkinter.messagebox.showinfo(
38             message=self.dog_listbox.get(indexes[0]))
39     else:
40         tkinter.messagebox.showinfo(
41             message='Ни один элемент не выбран.')
42
43 # Создать экземпляр класса ListBoxSelection.
44 if __name__ == '__main__':
45     listbox_selection = ListBoxSelection()

```

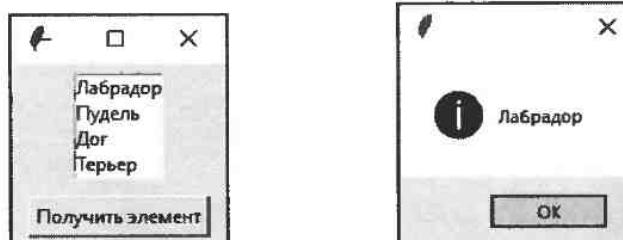


РИС. 13.38. Окна, выводимые программой 13.19

Давайте рассмотрим эту программу подробнее. Виджет `Listbox` создается в строках 11–12 программы. Поскольку мы не указали режим выбора, в виджете будет использоваться режим выбора, принятый по умолчанию, т. е. `tkinter.BROWSE`. Напомним, что этот режим выбора позволяет пользователю выбирать только по одному элементу за раз. Стока 16 создает список с названиями пород собак, а цикл в строках 19–20 вставляет элементы этого списка в виджет `Listbox`. Строки 23–25 создают виджет `Button` с использованием метода `self.__retrieve_dog()`, указанного в качестве функции обратного вызова.

Метод `self.__retrieve_dog()` появляется в строках 31–41. В строке 33 вызывается метод `curselection()` виджета `Listbox` для получения кортежа, содержащего индекс выбранного пользователем элемента. Поскольку виджет `Listbox` использует режим выбора `tkinter.BROWSE`, мы знаем, что кортеж будет содержать либо 0 элементов (если пользователь не выбрал элемент), либо 1 элемент. Инструкция `if` в строке 36 вызывает функцию `len`, чтобы определить, содержит ли кортеж `indexes` более 0 элементов. Если это так, то код в строках 37–38 получает элемент, выбранный из виджета `Listbox`, и выводит его в диалоговом окне. Если кортеж `indexes` пуст (что указывает на отсутствие выбранного из виджета элемента), то выполняется код в строках 40–41, выводящий сообщение 'Ни один элемент не выбран.'.

Удаление элементов из виджета `Listbox`

Вы можете удалить элемент из списка, вызвав метод `delete()` виджета `Listbox`. При этом вы передаете индекс элемента, который хотите удалить. Например, предположим, что в следующей ниже инструкции переменная `self.listbox` ссылается на виджет `Listbox`:

```
self.listbox.delete(0)
```

После исполнения этой инструкции первый элемент в списке (с индексом 0) будет удален. Все элементы, которые появятся после удаленного элемента, будут сдвинуты на одну позицию в верхнюю часть виджета `Listbox`. Если вы хотите удалить элемент, который в данный момент выбран в виджете `Listbox`, то в качестве индекса вы можете использовать специальное значение `tkinter.ACTIVE`. Приведем пример:

```
self.listbox.delete(tkinter.ACTIVE)
```

После исполнения этой инструкции элемент, выбранный в данный момент в виджете `Listbox`, будет удален. Все элементы, которые появятся после удаленного элемента, будут сдвинуты на одну позицию в верхнюю часть виджета `Listbox`.

Вы также можете удалить *диапазон*, представляющий собой группу смежных элементов в виджете `Listbox`. Если вы передадите методу `delete()` два целочисленных аргумента, то первый аргумент будет индексом первого элемента в диапазоне, а второй аргумент — индексом последнего элемента в диапазоне. Вот пример:

```
self.listbox.delete(0, 4)
```

После выполнения этой инструкции элементы, которые отображаются в индексах от 0 до 4 в списке, будут удалены. Все элементы, которые появятся после удаленных элементов, будут смещены в верхнюю часть виджета `Listbox`.

Если вы хотите удалить все элементы в виджете, то следует вызвать метод `delete()` с 0 в качестве первого аргумента и `tkinter.END` в качестве второго аргумента. Приведем пример:

```
self.listbox.delete(0, tkinter.END)
```

Исполнение функции обратного вызова, когда пользователь щелкает на элементе виджета *Listbox*

Если вы хотите, чтобы программа немедленно выполнила действие, когда пользователь выбирает элемент в списке, вы можете написать функцию обратного вызова, а затем привязать эту функцию к виджету *Listbox*. Когда пользователь выбирает элемент в виджете, функция обратного вызова будет исполнена немедленно.

Для привязки функции обратного вызова к виджету *Listbox* надо вызывать функцию *bind* виджета *Listbox*. Вот общий формат:

```
listbox.bind('<<ListboxSelect>>', функция_обратного_вызова)
```

В общем формате *listbox* — это имя виджета *Listbox*, а *функция_обратного_вызова* — имя функции обратного вызова. Например, предположим, что в программе есть виджет *Listbox* с именем *self.listbox* и вы хотите привязать функцию обратного вызова *self.do_something* к этому виджету. Следующая ниже инструкция показывает, как это делается:

```
self.listbox.bind('<<ListboxSelect>>', self.do_something)
```

Когда пользователь выбирает элемент в списке, вызывается функция обратного вызова *self.do_something*, и событийный объект передается в качестве аргумента функции обратного вызова. *Событийный объект* — это объект, содержащий информацию о событии. В этой книге событийные объекты никак не используются, но, когда мы пишем код для функции обратного вызова, следует иметь в виду, что функции потребуется параметр для событийного объекта.

В ЦЕНТРЕ ВНИМАНИЯ

Программа часовых поясов



В этой рубрике мы рассмотрим графическую программу, которая позволяет пользователю выбирать город из виджета *Listbox*. Когда пользователь нажимает кнопку, программа выводит на экран название часового пояса этого города. На рис. 13.39 показан эскиз пользовательского интерфейса программы с именами виджетов.

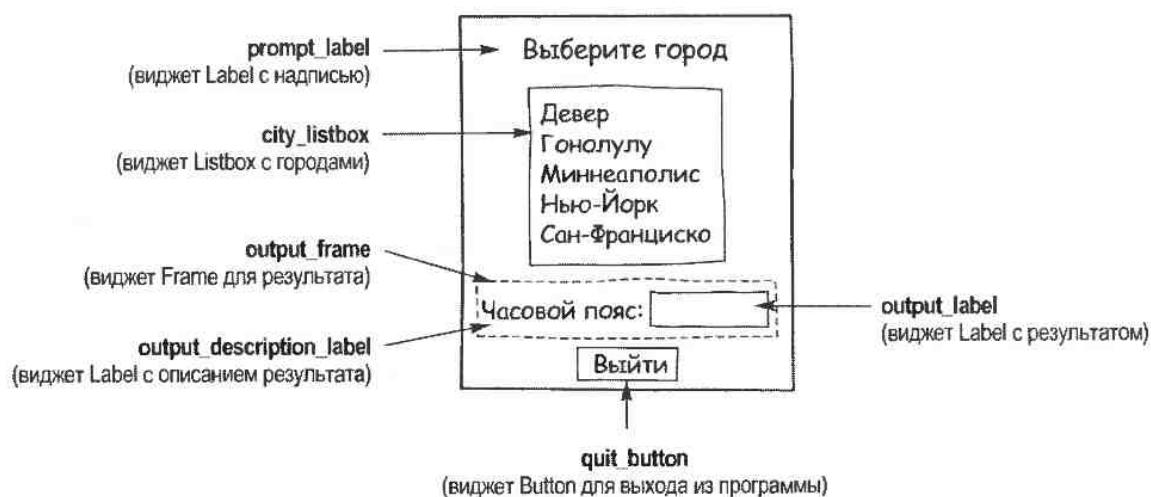


РИС. 13.39. Эскиз программы часовых поясов


```
31      # Создать и упаковать виджет Listbox.
32      self.city_listbox = tkinter.Listbox(
33          self.main_window, height=0, width=0)
34      self.city_listbox.pack(padx=5, pady=5)
35
36
37      # Привязать функцию обратного вызова к виджету Listbox.
38      self.city_listbox.bind(
39          '<<ListboxSelect>>', self.__display_time_zone)
40
41      # Заполнить виджет Listbox.
42      for city in self.__cities:
43          self.city_listbox.insert(tkinter.END, city)
44
45      # Этот метод создает рамку output_frame и ее содержимое.
46      def __build_output_frame(self):
47          # Создать рамку.
48          self.output_frame = tkinter.Frame(self.main_window)
49          self.output_frame.pack(padx=5)
50
51          # Создать виджет Label с надписью "Часовой пояс:".
52          self.output_description_label = tkinter.Label(
53              self.output_frame, text='Часовой пояс:')
54          self.output_description_label.pack(
55              side='left', padx=(5, 1), pady=5)
56
57          # Создать переменную StringVar для хранения имени часового пояса.
58          self.__timezone = tkinter.StringVar()
59
60          # Создать метку Label, которая выводит имя часового пояса.
61          self.output_label = tkinter.Label(
62              self.output_frame, borderwidth=1, relief='solid',
63              width=15, textvariable=self.__timezone)
64          self.output_label.pack(side='right', padx=(1, 5), pady=5)
65
66      # Этот метод создает кнопку "Выйти".
67      def __build_quit_button(self):
68          self.quit_button = tkinter.Button(
69              self.main_window, text='Выйти',
70              command=self.main_window.destroy)
71          self.quit_button.pack(padx=5, pady=5)
72
73      # Функция обратного вызова для виджета city_listbox.
74      def __display_time_zone(self, event):
75          # Получить текущие варианты выбора.
76          index = self.city_listbox.curselection()
```

```
78     # Получить город.
79     city = self.city_listbox.get(index[0])
80
81     # Определить временной пояс.
82     if city == 'Денвер':
83         self._timezone.set('Горный')
84     elif city == 'Гонолулу':
85         self._timezone.set('Гавайско-алаеутский')
86     elif city == 'Миннеаполис':
87         self._timezone.set('Центральный')
88     elif city == 'Нью-Йорк':
89         self._timezone.set('Восточный')
90     elif city == 'Сан-Франциско':
91         self._timezone.set('Тихоокеанский')
92
93 # Создать экземпляр класса TimeZone
94 if __name__ == '__main__':
95     time_zone = TimeZone()
```

Давайте рассмотрим класс часовых поясов `TimeZone`.

- ◆ Метод `__init__` появляется в строках 6–18 программы. Он создает главное окно, вызывает другие методы для создания виджетов, выводимых на экран главным окном, и запускает главный цикл.
- ◆ Метод `__build_prompt_label` появляется в строках 21–24. Он создает виджет `Label` с надписью 'Выберите город'.
- ◆ Метод `__build_listbox` появляется в строках 27–43. Он создает виджет `Listbox`, который показывает названия городов, привязывает список к функции обратного вызова и заполняет список названиями городов.
- ◆ Метод `__build_output_frame` появляется в строках 46–64. Он создает виджет `Frame`, содержащий виджет `Label` с надписью 'Часовой пояс' и виджет `output_label`. С виджетом `output_label` связана переменная объекта `StringVar` с именем `_timezone`.
- ◆ Метод `__build_quit_button` появляется в строках 67–71. Он создает виджет `quit_button`, который закрывает окно и завершает программу.
- ◆ Метод `__display_time_zone` появляется в строках 74–91. Это функция обратного вызова виджета `Listbox`. Указанный метод будет вызываться всякий раз, когда пользователь выбирает элемент в списке. (Обратите внимание, что этот метод имеет параметр `event`. Указанный параметр необходим, поскольку событийный объект будет передаваться методу при его вызове. В этой программе событийный объект не используется, но нам все равно нужен параметр для получения объекта.) Этот метод получает выбранный в виджете `Listbox` город и использует инструкцию `if-elif` для определения правильного часового пояса. Переменная `_timezone` устанавливается равной имени часового пояса. Это приводит к отображению имени часового пояса в виджете `output_label`.

Добавление полос прокрутки в виджет Listbox

По умолчанию виджет Listbox не показывает полосы прокрутки. Если высота списка меньше, чем число элементов в виджете, некоторые элементы не будут отображаться. Кроме того, если ширина виджета меньше ширины элемента в нем, то часть этого элемента не будет отображаться. В таких случаях в виджет Listbox следует добавлять полосы прокрутки, чтобы пользователь мог прокручивать содержимое виджета.

С виджетом Listbox можно использовать вертикальную полосу прокрутки и/или горизонтальную полосу прокрутки. Вертикальная полоса прокрутки позволяет прокручивать содержимое списка вверх и вниз, или вертикально. Горизонтальная полоса прокрутки позволяет прокручивать содержимое списка влево и вправо, или горизонтально. На рис. 13.41 показан пример.



РИС. 13.41. Вертикальная и горизонтальная полосы прокрутки

Добавление вертикальной полосы прокрутки

Добавление полосы прокрутки в виджет Listbox требует нескольких шагов. Вот краткое описание процедуры добавления вертикальной полосы прокрутки:

1. Создать рамку для размещения виджета Listbox и полосы прокрутки.

Рекомендуется создавать рамку специально для виджета и его полосы прокрутки. Это облегчает их правильное расположение с помощью метода pack().

2. Упаковать рамку.

Упаковать рамку с любым необходимым заполнением.

3. Создать виджет Listbox внутри рамки.

Создать виджет Listbox с нужными высотой и шириной. Обязательно назначить рамку, созданную на шаге 1, в качестве родительского виджета для виджета Listbox.

4. Упаковать виджет Listbox в левую часть рамки.

Это стандартное правило для того, чтобы вертикальная полоса прокрутки списка отображалась в правой части списка (см. рис. 13.41). Поэтому, когда вы вызываете метод pack() виджета Listbox, следует передавать ему аргумент side='left', чтобы упаковывать виджет в левую часть рамки.

5. Создать вертикальную полосу прокрутки внутри рамки.

Для создания полосы прокрутки необходимо создать экземпляр класса `Scrollbar` модуля `tkinter`. При передаче аргументов методу `__init__` класса следует назначить созданную на шаге 1 рамку в качестве родительского виджета. Кроме того, следует передать аргумент `orient=tkinter.VERTICAL`.

6. Упаковать полосу прокрутки в правую часть рамки.

Как уже упоминалось ранее, это стандартное правило для вертикальной полосы прокрутки виджета `Listbox`, отображаемой в правой части указанного виджета (см. рис. 13.41). Поэтому, когда вы вызываете метод `pack()` виджета `Scrollbar`, следует передать аргумент `side='right'`, чтобы упаковать виджет в правую часть рамки. Кроме того, вы должны передать аргумент `fill=tkinter.Y`, чтобы полоса прокрутки расширялась от верхней части рамки до нижней.

7. Сконфигурировать полосу прокрутки для вызова метода `yview` виджета `Listbox` при перемещении ползунка полосы прокрутки.

Виджет `Listbox` имеет метод `yview()`, который заставляет содержимое списка прокручиваться по вертикали. Для вызова метода `yview` виджета `Listbox` в любое время при перемещении ползунка полосы прокрутки необходимо настроить виджет `Scrollbar`. Это делается путем вызова метода `config()` виджета `Scrollbar` с передачей аргумента `command=listbox.yview` (где `listbox` — это имя виджета `Listbox`).

8. Сконфигурировать список так, чтобы он вызывал метод `set()` полосы прокрутки при каждом обновлении списка.

Всякий раз, когда содержимое виджета `Listbox` прокручивается, этот виджет должен взаимодействовать с полосой прокрутки, чтобы он мог обновлять положение ползунка. Виджет `Listbox` делает это, вызывая метод `set()` виджета `Scrollbar`. Это делается путем вызова метода `config()` виджета `Listbox` с передачей аргумента `yscrollcommand=scrollbar.set` (где `scrollbar` — это имя виджета `Scrollbar`).

Программа 13.21 демонстрирует пример создания виджета `Listbox` с функционирующей вертикальной полосой прокрутки. На рис. 13.42 показано окно, выводимое на экран программой.

Программа 13.21 (vertical_scrollbar.py)

```
1 # Эта программа демонстрирует виджет Listbox с вертикальной прокруткой.
2 import tkinter
3
4 class VerticalScrollbarExample:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать рамку для виджета Listbox и вертикальную прокрутку.
10        self.listbox_frame = tkinter.Frame(self.main_window)
11        self.listbox_frame.pack(padx=20, pady=20)
12
```

```

13     # Создать виджет Listbox в рамке listbox_frame.
14     self.listbox = tkinter.Listbox(
15         self.listbox_frame, height=6, width=0)
16     self.listbox.pack(side='left')
17
18     # Создать вертикальный виджет Scrollbar в рамке listbox_frame.
19     self.scrollbar = tkinter.Scrollbar(
20         self.listbox_frame, orient=tkinter.VERTICAL)
21     self.scrollbar.pack(side='right', fill=tkinter.Y)
22
23     # Сконфигурировать виджеты Scrollbar и Listbox для совместной работы.
24     self.scrollbar.config(command=self.listbox.yview)
25     self.listbox.config(yscrollcommand=self.scrollbar.set)
26
27     # Создать список названий месяцев.
28     months = ['Январь', 'Февраль', 'Март', 'Апрель',
29                'Май', 'Июнь', 'Июль', 'Август', 'Сентябрь',
30                'Октябрь', 'Ноябрь', 'Декабрь']
31
32     # Заполнить виджет Listbox данными.
33     for month in months:
34         self.listbox.insert(tkinter.END, month)
35
36     # Запустить главный цикл.
37     tkinter.mainloop()
38
39 # Создать экземпляр класса VerticalScrollbarExample.
40 if __name__ == '__main__':
41     scrollbar_example = VerticalScrollbarExample()

```

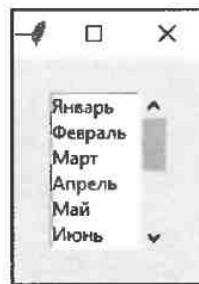


РИС. 13.42. Окно, выводимое на экран программой 13.19

Добавление только горизонтальной полосы прокрутки

Процедура добавления горизонтальной полосы прокрутки в виджет Listbox очень похожа на процедуру добавления вертикальной полосы прокрутки. Вот описание шагов:

1. Создать рамку для размещения виджета `Listbox` и полосы прокрутки.

Рекомендуется создавать рамку специально для виджета `Listbox` и его полосы прокрутки. Это облегчает их правильное расположение с помощью метода `pack()`.

2. Упаковать рамку.

Упаковать рамку с любым необходимым заполнением.

3. Создать виджет `Listbox` внутри рамки.

Создать виджет `Listbox` с нужной высотой и шириной. Обязательно назначить рамку, созданную на шаге 1, в качестве родительского виджета для виджета `Listbox`.

4. Упаковать виджет `Listbox` в верхней части рамки.

Это стандартное правило для горизонтальной полосы прокрутки виджета `Listbox`, которая отображается под виджетом (см. рис. 13.41). Поэтому, когда вы вызываете метод `pack()` виджета `Listbox`, следует передать аргумент `side='top'`, чтобы упаковать виджет в верхней части рамки.

5. Создать горизонтальную полосу прокрутки внутри рамки.

Для создания полосы прокрутки необходимо создать экземпляр класса `Scrollbar` модуля `tkinter`. При передаче аргументов методу `__init__` класса следует назначить созданную на шаге 1 рамку в качестве родительского виджета. Кроме того, следует передать аргумент `orient=tkinter.HORIZONTAL`.

6. Упаковать полосу прокрутки в нижней части рамки.

Как уже упоминалось ранее, это стандартное правило для горизонтальной полосы прокрутки виджета `Listbox`, которая отображается под этим виджетом (см. рис. 13.41). Поэтому, когда вы вызываете метод `pack()` виджета `Scrollbar`, следует передать аргумент `side='bottom'`, чтобы упаковать виджет в нижней части рамки. Кроме того, следует передать аргумент `fill=tkinter.x`, чтобы полоса прокрутки расширялась от левой стороны рамки до его правой стороны.

7. Сконфигурировать полосу прокрутки для вызова метода `xview()` виджета `Listbox` при перемещении ползунка полосы прокрутки.

Виджет `Listbox` имеет метод `xview()`, который заставляет содержимое списка прокручиваться по горизонтали. Для вызова метода `xview()` виджета `Listbox` в любое время при перемещении ползунка полосы прокрутки необходимо настроить виджет `Scrollbar`. Это делается путем вызова метода `config()` виджета `Scrollbar` с передачей аргумента `command=listbox.xview` (где `listbox` — это имя виджета `Listbox`).

8. Сконфигурировать виджет `Listbox` так, чтобы он вызывал метод `set()` полосы прокрутки при каждом обновлении виджета.

Всякий раз, когда содержимое виджета `Listbox` прокручивается, список должен взаимодействовать с полосой прокрутки, чтобы тот мог обновлять положение ползунка. Виджет `Listbox` делает это, вызывая метод `set()` виджета `Scrollbar`. Это делается путем вызова метода `config()` виджета `Listbox` с передачей аргумента `xscrollcommand=scrollbar.set` (где `scrollbar` — это имя виджета `Scrollbar`).

Программа 13.22 демонстрирует пример создания виджета `Listbox` с функционирующей горизонтальной полосой прокрутки. На рис. 13.43 показано окно, выводимое на экран программой.

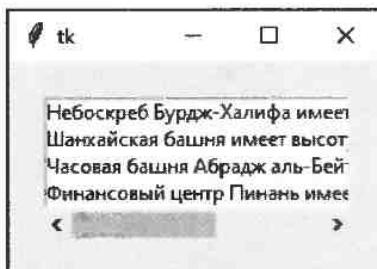


РИС. 13.43. Окно, выводимое на экран программой 13.20

Программа 13.22 (horizontal_scrollbar.py)

```

1 # Эта программа демонстрирует виджет Listbox с горизонтальной прокруткой.
2 import tkinter
3
4 class HorizontalScrollbarExample:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать рамку для виджета Listbox и вертикальную прокрутку.
10        self.listbox_frame = tkinter.Frame(self.main_window)
11        self.listbox_frame.pack(padx=20, pady=20)
12
13        # Создать виджет Listbox в рамке listbox_frame.
14        self.listbox = tkinter.Listbox(
15            self.listbox_frame, height=0, width=30)
16        self.listbox.pack(side='top')
17
18        # Создать горизонтальный виджет Scrollbar в рамке listbox_frame.
19        self.scrollbar = tkinter.Scrollbar(
20            self.listbox_frame, orient=tkinter.HORIZONTAL)
21        self.scrollbar.pack(side='bottom', fill=tkinter.X)
22
23        # Сконфигурировать виджеты Scrollbar и Listbox для совместной работы.
24        self.scrollbar.config(command=self.listbox.xview)
25        self.listbox.config(xscrollcommand=self.scrollbar.set)
26
27        # Создать список.
28        data = [
29            'Небоскреб Бурдж-Халифа имеет высоту 2717 футов.',
30            'Шанхайская башня имеет высоту 2073 фута.',
31            'Часовая башня Абрадж аль-Бейт имеет высоту 1971 фут.',
32            'Финансовый центр Пинань имеет высоту 1965 футов.']

```

```
34     # Заполнить виджет Listbox данными.
35     for element in data:
36         self.listbox.insert(tkinter.END, element)
37
38     # Запустить главный цикл.
39     tkinter.mainloop()
40
41 # Создать экземпляр класса HorizontalScrollbarExample.
42 if __name__ == '__main__':
43     scrollbar_example = HorizontalScrollbarExample()
```

Добавление вертикальной и горизонтальной полос прокрутки одновременно

При одновременном добавлении в виджет Listbox вертикальной и горизонтальной полос прокрутки рекомендуется использовать две вложенные рамки. Внутренняя рамка будет содержать виджет Listbox и вертикальную полосу прокрутки, а внешняя рамка — внутреннюю рамку и горизонтальную полосу прокрутки. Вот краткое изложение этой процедуры:

1. Создать внешнюю рамку, которая будет содержать внутреннюю рамку и горизонтальную полосу прокрутки.

Внешняя рамка будет содержать только внутреннюю рамку и горизонтальную полосу прокрутки.

2. Упаковать внешнюю рамку.

Упаковать внешнюю рамку с любым необходимым заполнением.

3. Создать внутреннюю рамку, которая будет содержать виджет Listbox и вертикальную полосу прокрутки.

Внутренняя рамка будет содержать только виджет Listbox и его вертикальную полосу прокрутки. Следует назначить внешнюю рамку, созданную на шаге 1, в качестве родительского виджета для внутренней рамки виджета.

4. Упаковать внутреннюю рамку.

Эта рамка будет вложена во внешнюю рамку, поэтому добавлять заполнение при упаковке внутренней рамки нет необходимости.

5. Создать виджет Listbox внутри внутренней рамки.

Следует создать виджет Listbox с нужными высотой и шириной и назначить внутреннюю рамку, созданную на шаге 3, в качестве родительского виджета для виджета Listbox.

6. Упаковать виджет Listbox в левую часть внутренней рамки.

Это стандартное правило для того, чтобы вертикальная полоса прокрутки отображалась в правой части виджета Listbox (см. рис. 13.41). Поэтому, когда вы вызываете метод pack() виджета Listbox, следует передать аргумент side='left', чтобы упаковать виджет в левую часть внутренней рамки.

7. Создать вертикальную полосу прокрутки внутри внутренней рамки.

Следует создать экземпляр класса Scrollbar модуля tkinter. При передаче аргументов в метод __init__ указанного класса следует назначить внутреннюю рамку, созданную на

шаге 3, в качестве родительского виджета. Кроме того, следует передать аргумент `orient=tkinter.VERTICAL`.

8. Упаковать вертикальную полосу прокрутки в правую часть внутренней рамки.

Как уже упоминалось ранее, это стандартное правило для вертикальной полосы прокрутки виджета `Listbox`, отображаемой в правой части виджета (см. рис. 13.41). Поэтому, когда вы вызываете метод `pack()` вертикальной полосы прокрутки, следует передать аргумент `side='right'`, чтобы упаковать виджет в правую часть внутренней рамки. Кроме того, следует передать аргумент `fill=tkinter.Y`, чтобы полоса прокрутки расширялась от верхней части рамки до нижней.

9. Создать горизонтальную полосу прокрутки внутри внешней рамки.

Следует создать экземпляр класса `Scrollbar` модуля `tkinter`. При передаче аргументов в метод `__init__` этого класса следует назначить внешнюю рамку, созданную на шаге 1, в качестве родительского виджета. Кроме того, следует передать аргумент `orient=tkinter.HORIZONTAL`.

10. Упаковать горизонтальную полосу прокрутки в нижнюю часть внешней рамки.

Как уже упоминалось ранее, это стандартное правило для горизонтальной полосы прокрутки виджета `Listbox`, которая отображается под списком (см. рис. 13.41). Поэтому, когда вы вызываете метод `pack()` горизонтальной полосы прокрутки, следует передать аргумент `side='bottom'`, чтобы упаковать виджет в нижней части внешней рамки. Кроме того, следует передать аргумент `fill=tkinter.X`, чтобы полоса прокрутки расширялась от левой стороны рамки до правой.

11. Сконфигурировать вертикальную полосу прокрутки для вызова метода `yview()` виджета `Listbox` при перемещении ползунка полосы прокрутки.

Виджет `Listbox` имеет метод `yview()`, который заставляет содержимое виджета прокручиваться по вертикали. Необходимо настроить виджет вертикальной полосы прокрутки `Scrollbar` для вызова метода `yview()` виджета `Listbox` в любое время при перемещении ползунка полосы прокрутки. Это делается путем вызова метода `config()` виджета вертикальной полосы прокрутки `Scrollbar` с передачей аргумента `command=listbox.yview` (где `listbox` — это имя виджета `Listbox`).

12. Сконфигурировать горизонтальную полосу прокрутки для вызова метода `xview()` виджета `Listbox` при перемещении ползунка полосы прокрутки.

Виджет `Listbox` имеет метод `xview()`, который заставляет содержимое виджета `Listbox` прокручиваться по горизонтали. Необходимо настроить виджет вертикальной полосы прокрутки `Scrollbar` для вызова метода `xview()` виджета `Listbox` в любое время при перемещении ползунка полосы прокрутки. Это делается путем вызова метода `config()` виджета горизонтальной полосы прокрутки `Scrollbar` с передачей аргумента `command=listbox.xview` (где `listbox` — это имя виджета `Listbox`).

13. Сконфигурировать виджет `Listbox` так, чтобы он вызывал метод `set()` каждого виджета `Scrollbar` при каждом обновлении списка.

Всякий раз, когда содержимое виджета `Listbox` прокручивается, оно должно взаимодействовать с его полосами прокрутки, чтобы они могли обновлять положение своих ползунков. Виджет `Listbox` делает это, вызывая метод `set()` каждого виджета `Scrollbar`. Вызовите метод `config()` виджета `Listbox`, передав следующие аргументы:

- `yscrollcommand=verticalscrollbar.set` (где `verticalscrollbar` — это имя виджета вертикальной полосы прокрутки `Scrollbar`);
- `xscrollcommand=horizontalscrollbar.set` (где `horizontalscrollbar` — это имя виджета горизонтальной полосы прокрутки `Scrollbar`).

В программе 13.23 демонстрируется пример создания виджета `Listbox` с вертикальной и с горизонтальной полосами прокрутки. На рис. 13.44 показано окно, выводимое на экран программой.

Программа 13.23 (vertical_horizontal_scrollbar.py)

```
1 # Эта программа демонстрирует виджет Listbox с вертикальной
2 # и горизонтальной полосами прокрутки.
3 import tkinter
4
5 class ScrollbarExample:
6     def __init__(self):
7         # Создать главное окно.
8         self.main_window = tkinter.Tk()
9
10        # Создать внешнюю рамку, которая будет содержать
11        # внутреннюю рамку и горизонтальную полосу прокрутки.
12        self.outer_frame = tkinter.Frame(self.main_window)
13        self.outer_frame.pack(padx=20, pady=20)
14
15        # Создать внутреннюю рамку для Listbox и вертикальную полосу прокрутки.
16        self.inner_frame = tkinter.Frame(self.outer_frame)
17        self.inner_frame.pack()
18
19        # Создать виджет Listbox в рамке inner_frame.
20        self.listbox = tkinter.Listbox(
21            self.inner_frame, height=5, width=30)
22        self.listbox.pack(side='left')
23
24        # Создать вертикальный виджет Scrollbar в рамке inner_frame.
25        self.v_scrollbar = tkinter.Scrollbar(
26            self.inner_frame, orient=tkinter.VERTICAL)
27        self.v_scrollbar.pack(side='right', fill=tkinter.Y)
28
29        # Создать горизонтальный виджет Scrollbar в рамке outer_frame.
30        self.h_scrollbar = tkinter.Scrollbar(
31            self.outer_frame, orient=tkinter.HORIZONTAL)
32        self.h_scrollbar.pack(side='bottom', fill=tkinter.X)
33
34        # Сконфигурировать виджеты Scrollbar и Listbox для совместной работы.
35        self.v_scrollbar.config(command=self.listbox.yview)
36        self.h_scrollbar.config(command=self.listbox.xview)
```

```

37     self.listbox.config(yscrollcommand=self.v_scrollbar.set,
38                         xscrollcommand=self.h_scrollbar.set)
39
40     # Создать список.
41     data = [
42         'Небоскреб Бурдж-Халифа имеет высоту 2717 футов.',
43         'Шанхайская башня имеет высоту 2073 фута.',
44         'Часовая башня Абрадж аль-Бейт имеет высоту 1971 фут.',
45         'Финансовый центр Пинань имеет высоту 1965 футов.',
46         'Здание Goldin Finance имеет высоту 1957 футов.',
47         'Башня Lotte World имеет высоту 1819 футов.',
48         'Всемирный торговый центр 1 имеет высоту 1776 футов.']
49
50     # Заполнить виджет Listbox данными.
51     for element in data:
52         self.listbox.insert(tkinter.END, element)
53
54     # Запустить главный цикл.
55     tkinter.mainloop()
56
57 # Создать экземпляр класса ScrollbarExample.
58 if __name__ == '__main__':
59     scrollbar_example = ScrollbarExample()

```

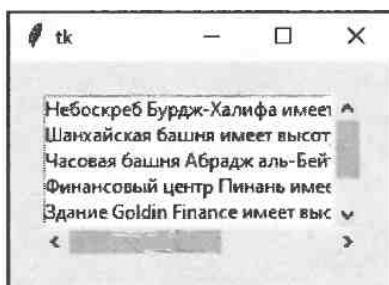


РИС. 13.44. Окно, выводимое на экран программой 13.23



Контрольная точка

13.23. Предположим, что в программе появляется следующая инструкция:

```
self.listbox = tkinter.Listbox(self.main_window)
```

Напишите код, чтобы добавить в список приведенные ниже элементы виджета Listbox:

- 'Январь' в индексной позиции 0;
- 'Февраль' в индексной позиции 1;
- 'Март' в индексной позиции 2.

13.24. Какова высота и ширина виджета Listbox по умолчанию?

13.25. Модифицируйте следующую ниже инструкцию, чтобы в виджете Listbox было 20 строк в высоту и 30 символов в ширину:

```
self.listbox = tkinter.Listbox(self.main_window)
```

13.26. Посмотрите на следующий код:

```
self.listbox = tkinter.Listbox(self.main_window)
self.listbox.insert(tkinter.END, 'Петр')
self.listbox.insert(tkinter.END, 'Павел')
self.listbox.insert(tkinter.END, 'Мария')
```

В каких индексных позициях хранятся элементы 'Петр', 'Павел' и 'Мария'?

13.27. Что возвращает метод `curselection()` виджета Listbox?

13.28. Как удалить элемент из виджета Listbox?

13.10 Рисование фигур с помощью виджета *Canvas*

Ключевые положения

Виджет *Canvas* предоставляет методы для рисования простых фигур, таких как линии, прямоугольники, овалы, многоугольники и т. д.

Виджет *Canvas* (Холст) — это незаполненная прямоугольная область, которая позволяет рисовать на своей поверхности простые двумерные фигуры. В этом разделе мы рассмотрим методы виджета *Canvas* для рисования отрезков прямой, прямоугольников, овалов, дуг, многоугольников и текста. Однако прежде чем заняться изучением методов рисования этих фигур, мы должны рассмотреть экранную систему координат. Экранная система координат виджета *Canvas* используется для указания позиции графических объектов.

Экранная система координат виджета *Canvas*

Изображения, выводимые на компьютерный экран, состоят из крошечных точек, которые называются *пикселями*. Экранная система координат используется для идентификации позиции каждого пикселя в окне приложения. Каждый пиксель имеет координаты *X* и *Y*. Координата *X* идентифицирует горизонтальную позицию пикселя, координата *Y* — ее вертикальную позицию. Координаты обычно пишутся в форме (*X*, *Y*).

В экранной системе координат виджета *Canvas* координаты пикселя в левом верхнем углу экрана равняются (0, 0). Это означает, что обе его координаты *X* и *Y* равняются 0. Координата *X* увеличивается слева направо, а координата *Y* — сверху вниз. В окне шириной 640 и высотой 480 пикселов координаты пикселя в правом нижнем углу окна равняются (639, 479). В том же окне координаты пикселя в центре окна равняются (319, 239). На рис. 13.45 показаны координаты разных пикселов в окне.

ПРИМЕЧАНИЕ

Экранная система координат виджета *Canvas* отличается от декартовой системы координат, которая используется в библиотеке *чертёжашей* графики. Вот эти отличия:

- в виджете Canvas точка (0, 0) находится в левом верхнем углу окна, в черепашьей графике точка (0, 0) находится в центре окна;
- в виджете Canvas координаты Y увеличиваются по мере перемещения вниз экрана, в черепашьей графике координаты Y уменьшаются по мере перемещения вниз экрана.

Виджет Canvas имеет многочисленные методы для рисования графических фигур на поверхности этого виджета. Мы рассмотрим методы:

- ◆ `create_line();`
- ◆ `create_rectangle();`
- ◆ `create_oval();`
- ◆ `create_arc();`
- ◆ `create_polygon();`
- ◆ `create_text();`

Прежде чем обсудить детали этих методов, рассмотрим программу 13.24. В ней используется виджет Canvas для нанесения прямых. На рис. 13.46 показано окно, которое программа выводит на экран.

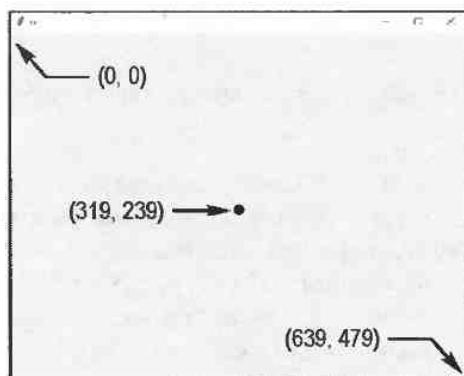


РИС. 13.45. Разные позиции пикселов в окне 640 на 480 пикселов

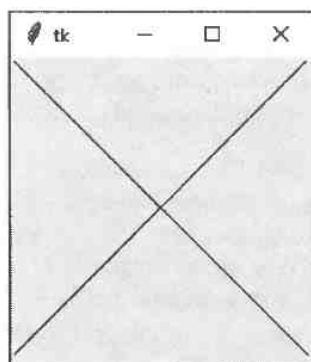


РИС. 13.46. Окно, выводимое на экран программой 13.24

Программа 13.24 (draw_line.py)

```

1 # Эта программа демонстрирует виджет Canvas.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Canvas.
10        self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
11

```

```

12     # Нарисовать две прямые.
13     self.canvas.create_line(0, 0, 199, 199)
14     self.canvas.create_line(199, 0, 0, 199)
15
16     # Упаковать холст.
17     self.canvas.pack()
18
19     # Запустить главный цикл.
20     tkinter.mainloop()
21
22 # Создать экземпляр класса MyGUI.
23 if __name__ == '__main__':
24     my_gui = MyGUI()

```

Рассмотрим эту программу. Стока 10 создает виджет `Canvas`. Первый аргумент внутри круглых скобок является ссылкой на `self.main_window`, т. е. родительский контейнер, в который мы добавляем этот виджет. Аргументы `width=200` и `height=200` задают размер виджета `Canvas`.

Строка 13 вызывает метод `create_line()` виджета `Canvas`, чтобы начертить прямую линию. Первый и второй аргументы являются координатами (X, Y) исходной точки прямой, третий и четвертый аргументы — координатами (X, Y) конечной точки прямой. Таким образом, эта инструкция чертит на виджете `Canvas` прямую из точки (0, 0) в точку (199, 199).

Строка 14 тоже вызывает метод `create_line()` виджета `Canvas`, чтобы начертить вторую прямую. Инструкция чертит на виджете `Canvas` прямую из точки (199, 0) в точку (0, 199).

Строка 17 вызывает метод `pack()` виджета `Canvas`, который делает этот виджет видимым. Стока 20 исполняет функцию `mainloop` модуля `tkinter`.

Рисование прямых: метод `create_line()`

Метод `create_line()` чертит на виджете `Canvas` отрезки прямой между двумя или несколькими точками. Вот общий формат вызова этого метода для рисования отрезка прямой между двумя точками:

`имя_холста.create_line(x1, y1, x2, y2, опции...)`

Аргументы `x1` и `y1` — это координаты (X, Y) исходной точки прямой. Аргументы `x2` и `y2` — координаты (X, Y) конечной точки прямой. В приведенном выше общем формате в `опции...` указывается несколько необязательных именованных аргументов, которые можно передавать в этот метод. Некоторые из них мы рассмотрим в табл. 13.5.

Вы видели примеры вызова метода `create_line()` в программе 13.24. Напомним, что строка 13 в этой программе чертит линию от (0, 0) до (199, 199):

`self.canvas.create_line(0, 0, 199, 199)`

В качестве аргументов можно передавать многочисленные наборы координат. Метод `create_line()` начертит отрезки прямой, соединяющие эти точки. Программа 13.25 демонстрирует, как это делается. Инструкция в строке 13 чертит отрезки, соединяющие точ-

ки (10, 10), (189, 10), (100, 189) и (10, 10). На рис. 13.47 представлено окно, которое данная программа выводит на экран.

Программа 13.25 (draw_multi_lines.py)

```

1 # Эта программа соединяет несколько точек отрезками прямой.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Canvas.
10        self.canvas = tkinter.Canvas(self.main_window,
11                                     width=200, height=200)
12        # Начертить отрезки прямой, соединяющие несколько точек.
13        self.canvas.create_line(10, 10, 189, 10, 100, 189, 10, 10)
14
15        # Упаковать холст.
16        self.canvas.pack()
17
18        # Запустить главный цикл.
19        tkinter.mainloop()
20
21 # Создать экземпляр класса MyGUI.
22 if __name__ == '__main__':
23     my_gui = MyGUI()

```

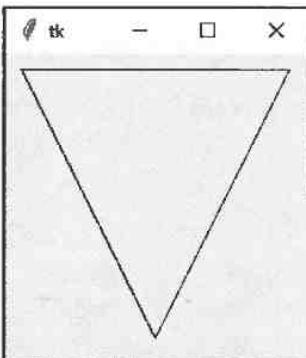


РИС. 13.47. Окно, выводимое программой 13.25

В качестве аргумента также можно передать список или кортеж, содержащий координаты. Например, в программе 13.25 можно было бы заменить строку 13 приведенным ниже фрагментом кода и получить те же самые результаты:

```

points = [10, 10, 189, 10, 100, 189, 10, 10]
self.canvas.create_line(points)

```

В метод `create_line()` можно передавать несколько необязательных именованных аргументов. В табл. 13.5 приводятся некоторые наиболее часто используемые из них.

Таблица 13.5. Несколько необязательных аргументов для метода `create_line()`

Аргумент	Описание
<code>arrow=значение</code>	По умолчанию прямые линии не имеют стреловидных указателей, но этот аргумент наносит стрелку на один или оба конца. Добавьте <code>arrow=tk.FIRST</code> для нанесения указателя в начале прямой, <code>arrow=tk.LAST</code> для нанесения указателя в конце прямой или <code>arrow=tk.BOTH</code> для нанесения указателей в обоих концах прямой
<code>dash=значение</code>	Этот аргумент превращает прямую в пунктирную линию. Его значением является кортеж из целых чисел, который задает шаблон. Первое целое число — количество рисуемых пикселов, второе целое число — количество пропускаемых пикселов и т. д. Например, аргумент <code>dash=(5, 2)</code> нарисует 5 пикселов, пропустит 2 пикселя и будет повторяться до тех пор, пока не будет достигнут конец отрезка прямой
<code>fill=значение</code>	Задает цвет прямой в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Наиболее часто используемые из них: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>fill</code> , то цвет прямой по умолчанию будет черным.)
<code>smooth=значение</code>	По умолчанию аргумент <code>smooth</code> равен <code>False</code> , в результате чего данный метод чертит прямые отрезки, соединяющие указанные точки. Если задать <code>smooth=True</code> , то будут нанесены изогнутые сплайны
<code>width=значение</code>	Задает ширину отрезка в пикселях. Например, аргумент <code>width=5</code> нанесет отрезок шириной 5 пикселов. По умолчанию отрезки прямой имеют ширину 1 пикселя

Рисование прямоугольников: метод `create_rectangle()`

Метод `create_rectangle()` чертит на виджете `Canvas` прямоугольник. Вот общий формат вызова этого метода:

`имя_холста.create_rectangle(x1, y1, x2, y2, опции...)`

Аргументы `x1` и `y1` — это координаты (X , Y) левого верхнего угла прямоугольника. Параметры `x2` и `y2` — координаты (X , Y) правого нижнего угла прямоугольника. В приведенном выше общем формате в `опциях...` указывается несколько необязательных именованных аргументов, которые можно передавать в этот метод. Некоторые из них мы рассмотрим в табл. 13.6.

Программа 13.26 демонстрирует метод `create_rectangle()` в строке 13. Левый верхний угол прямоугольника находится в координате (20, 20), его правый нижний угол — в координате (180, 180). На рис. 13.48 показано окно, которое данная программа выводит на экран.

Программа 13.26 (draw_square.py)

```
1 # Эта программа чертит прямоугольник на виджете Canvas.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
```

```

6      # Создать главное окно.
7      self.main_window = tkinter.Tk()
8
9      # Создать виджет Canvas.
10     self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
11
12     # Нарисовать прямоугольник.
13     self.canvas.create_rectangle(20, 20, 180, 180)
14
15     # Упаковать холст.
16     self.canvas.pack()
17
18     # Запустить главный цикл.
19     tkinter.mainloop()
20
21 # Создать экземпляр класса MyGUI.
22 if __name__ == '__main__':
23     my_gui = MyGUI()

```

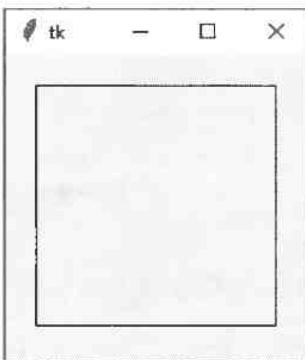


РИС. 13.48. Окно, выводимое на экран программой 13.26

В метод `create_rectangle()` можно передавать несколько необязательных именованных аргументов. В табл. 13.6 приведены некоторые наиболее часто используемые из них.

Таблица 13.6. Несколько необязательных аргументов для метода `create_rectangle()`

Аргумент	Описание
<code>dash=значение</code>	Этот аргумент делает контур прямоугольника пунктирным. Его значением является кортеж из целых чисел, которые задают шаблон. Первое целое число — количество рисуемых пикселов, второе целое число — количество пропускаемых пикселов и т. д. Например, аргумент <code>dash=(5, 2)</code> нарисует 5 пикселов, пропустит 2 пикселя и будет повторяться до тех пор, пока контур не замкнется
<code>fill=значение</code>	Задает цвет, которым прямоугольник может быть заполнен. Значением этого аргумента является имя цвета в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>fill</code> , то прямоугольник останется незаполненным.)

Таблица 13.6 (окончание)

Аргумент	Описание
outline=значение	Задает цвет контура прямоугольника в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент outline, то цвет контура прямоугольника по умолчанию будет черным.)
width=значение	Задает ширину контура прямоугольника в пикселях. Например, аргумент width=5 нанесет линию контура шириной 5 пикселов. По умолчанию линия контура прямоугольника имеет ширину 1 пикселя

Например, если изменить строку 13 в программе 13.26 следующим образом:

```
self.canvas.create_rectangle(20, 20, 180, 180, dash=(5, 2), width=3)
```

то программа начертит прямоугольник с пунктирным контуром и шириной линии 3 пикселя. Вывод программы в этом случае показан на рис. 13.49.

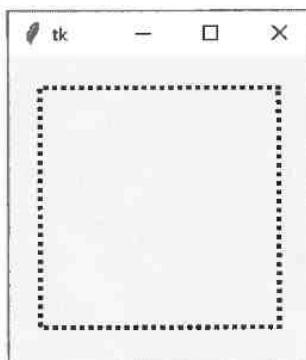


РИС. 13.49. Пунктирный контур с шириной линии 3 пикселя

Рисование овалов: метод `create_oval()`

Метод `create_oval()` чертит овал, или эллипс. Вот общий формат вызова этого метода:

```
имя_холста.create_oval(x1, y1, x2, y2, опции...)
```

Данный метод чертит овал, который строго укладывается в ограничивающий прямоугольник, заданный координатами, переданными в качестве аргументов. Координаты (x_1, y_1) — это координаты левого верхнего угла прямоугольника, (x_2, y_2) — координаты правого нижнего угла прямоугольника (рис. 13.50). В приведенном выше общем формате в *опциях...* указывается несколько необязательных именованных аргументов, которые можно передавать в этот метод. Некоторые из них мы рассмотрим в табл. 13.7.

Программа 13.27 демонстрирует метод `create_oval()` в строках 13 и 14. Первый овал, который чертится в строке 13, задан прямоугольником со своим левым верхним углом в координате $(20, 20)$ и своим правым нижним углом в координате $(70, 70)$. Второй овал, который чертится в строке 14, задан прямоугольником со своим левым верхним углом в координа-

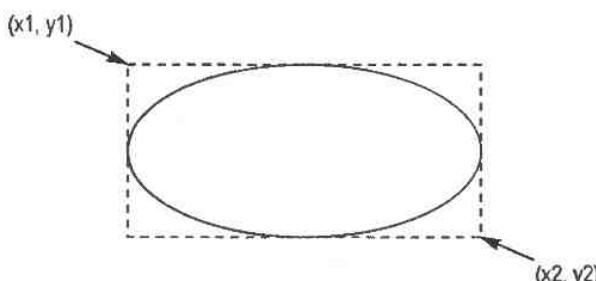


РИС. 13.50. Овал, ограничивающий четырехугольник

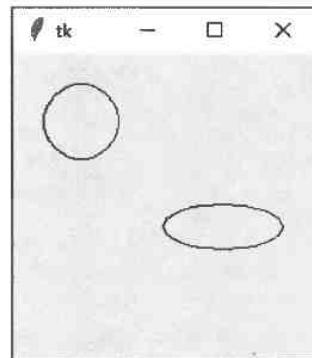


РИС. 13.51. Окно, выводимое на экран программой 13.27

те (100, 100) и своим правым нижним углом в координате (180, 130). На рис. 13.51 показано окно, которое данная программа выводит на экран.

Программа 13.27 (draw_ovals.py)

```

1 # Эта программа чертит два овала на виджете Canvas.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Canvas.
10        self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
11
12        # Нарисовать два овала.
13        self.canvas.create_oval(20, 20, 70, 70)
14        self.canvas.create_oval(100, 100, 180, 130)
15
16        # Упаковать холст.
17        self.canvas.pack()
18
19        # Запустить главный цикл.
20        tkinter.mainloop()
21
22 # Создать экземпляр класса MyGUI.
23 if __name__ == '__main__':
24     my_gui = MyGUI()

```

В метод `create_oval()` можно передавать несколько необязательных именованных аргументов. В табл. 13.7 приведены некоторые наиболее часто используемые.

Таблица 13.7. Несколько необязательных аргументов для метода `create_oval()`

Аргумент	Описание
<code>dash=значение</code>	Этот аргумент делает контур овала пунктирным. Его значением является кортеж из целых чисел, которые задают шаблон. Первое целое число — количество рисуемых пикселов, второе целое число — количество пропускаемых пикселов и т. д. Например, аргумент <code>dash=(5, 2)</code> нарисует 5 пикселов, пропустит 2 пикселя и будет повторяться до тех пор, пока контур не замкнется
<code>fill=значение</code>	Задает цвет, которым овал может быть заполнен. Значением этого аргумента является имя цвета в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>fill</code> , то овал останется незаполненным.)
<code>outline=значение</code>	Задает цвет контура овала в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>outline</code> , то цвет контура овала по умолчанию будет черным.)
<code>width=значение</code>	Задает ширину контура овала в пикселях. Например, аргумент <code>width=5</code> нанесет линии контура шириной 5 пикселов. По умолчанию линия контура овала имеет ширину 1 пикселя



СОВЕТ

Для того чтобы нарисовать круг, следует вызвать метод `create_oval()` и сделать все стороны ограничивающего прямоугольника одинаковой длины.

Рисование дуг: метод `create_arc()`

Метод `create_arc()` чертит дугу. Вот общий формат вызова этого метода:

`имя_холста.create_arc(x1, y1, x2, y2, start=угол, extent=ширина, опции...)`

Данный метод чертит дугу, которая является частью овала. Овал строго укладывается в ограничивающий прямоугольник, который задан координатами, переданными в качестве аргументов. Координаты $(x1, y1)$ — это координаты левого верхнего угла прямоугольника, а координаты $(x2, y2)$ — координаты правого нижнего угла прямоугольника. Аргумент `start=угол` задает исходный угол, аргумент `extent=ширина` задает в виде угла протяженность дуги против часовой стрелки. Например, аргумент `start=90` определяет, что дуга начинается в 90° , аргумент `extent=45` определяет, что дуга должна идти против часовой стрелки на 45° (рис. 13.52).

В приведенном выше общем формате в `опциях...` указывается несколько необязательных именованных аргументов, которые можно передавать в этот метод. Некоторые из них мы рассмотрим в табл. 13.8.

Программа 13.28 демонстрирует метод `create_arc()`. Дуга, нарисованная в строке 13, задается прямоугольником со своим левым верхним углом в координате $(10, 10)$ и своим правым нижним углом в координате $(190, 190)$. Дуга начинается в 45° и простирается на 30° . На рис. 13.53 показано окно, которое данная программа выводит на экран.

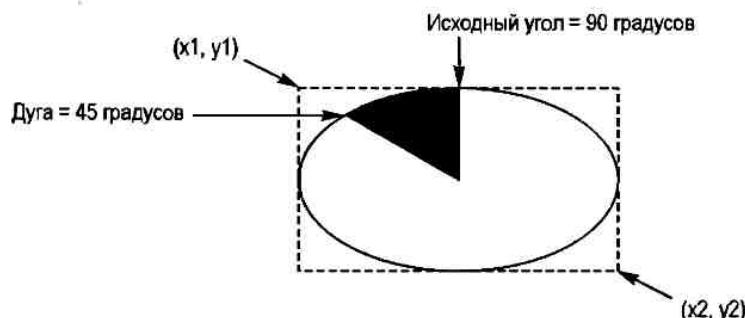


РИС. 13.52. Свойства дуги

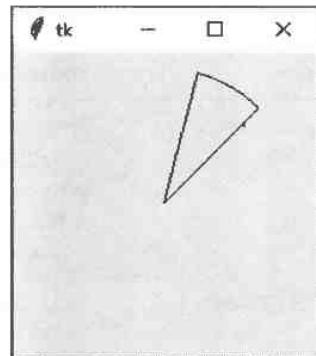


РИС. 13.53. Окно, выводимое на экран программой 13.28

Программа 13.28 (draw_arc.py)

```

1 # Эта программа чертит дугу на виджете Canvas.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Canvas.
10        self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
11
12        # Нарисовать дугу.
13        self.canvas.create_arc(10, 10, 190, 190, start=45, extent=30)
14
15        # Упаковать холст.
16        self.canvas.pack()
17
18        # Запустить главный цикл.
19        tkinter.mainloop()
20
21 # Создать экземпляр класса MyGUI.
22 if __name__ == '__main__':
23     my_gui = MyGUI()

```

В метод `create_arc()` можно передавать несколько необязательных именованных аргументов. В табл. 13.8 приведены некоторые наиболее часто используемые из них.

Один из трех стилей (табл. 13.9) рисуемой дуги задается аргументом `style=значение`. (По умолчанию используется тип `tkinter.PIESLICE` (сектор круга).) На рис. 13.54 показаны примеры каждого типа дуги.

Таблица 13.8. Несколько необязательных аргументов для метода `create_arc()`

Аргумент	Описание
<code>dash=значение</code>	Этот аргумент делает контур дуги пунктирным. Его значением является кортеж из целых чисел, которые задают шаблон. Первое целое число — количество рисуемых пикселов, второе целое число — количество пропускаемых пикселов и т. д. Например, аргумент <code>dash=(5, 2)</code> нарисует 5 пикселов, пропустит 2 пикселя и будет повторяться до тех пор, пока контур не закончится
<code>fill=значение</code>	Задает цвет, которым дуга может быть заполнена. Значением этого аргумента является имя цвета в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>fill</code> , то овал останется незаполненным.)
<code>outline=значение</code>	Задает цвет контура дуги в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>outline</code> , то цвет контура дуги по умолчанию будет черным.)
<code>style=значение</code>	Задает стиль дуги. Аргумент <code>style</code> может иметь одно из трех значений: <code>tkinter.PIESLICE</code> , <code>tkinter.ARC</code> или <code>tkinter.CHORD</code> (см. табл. 13.9)
<code>width=значение</code>	Задает ширину контура дуги в пикселях. Например, аргумент <code>width=5</code> нанесет линию контура шириной 5 пикселов. По умолчанию линия контура дуги имеет ширину 1 пикселя

Таблица 13.9. Типы дуг

Аргумент <code>style</code>	Описание
<code>style=tkinter.PIESLICE</code>	Этот тип дуги задан по умолчанию. Из обеих конечных точек до центральной точки дуги проводятся отрезки. В результате дуга будет сформирована как сектор круга
<code>style=tkinter.ARC</code>	Отрезки, соединяющие конечные точки, отсутствуют. Изображается только дуга
<code>style=tkinter.CHORD</code>	Из одной конечной точки дуги в другую ее конечную точку проводится отрезок прямой (хорда)



РИС. 13.54. Типы дуг

В программе 13.29 приведен соответствующий пример, в котором дуги используются для построения круговой диаграммы. Вывод программы показан на рис. 13.55.

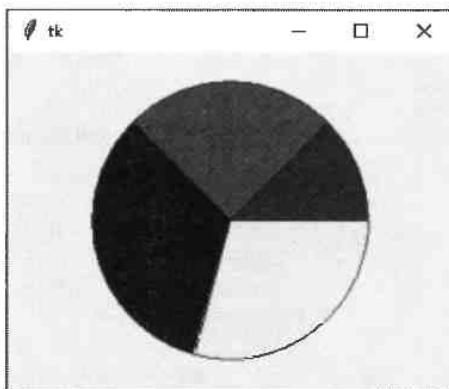


РИС. 13.55. Окно, выводимое на экран программой 13.29

Программа 13.29 (draw_piechart.py)

```
32                         extent=self._PIE1_WIDTH,
33                         fill='red')
34
35     # Начертить сектор 2.
36     self.canvas.create_arc(self._X1, self._Y1, self._X2,
37                             self._Y2, start=self._PIE2_START,
38                             extent=self._PIE2_WIDTH,
39                             fill='green')
40
41     # Начертить сектор 3.
42     self.canvas.create_arc(self._X1, self._Y1, self._X2,
43                             self._Y2, start=self._PIE3_START,
44                             extent=self._PIE3_WIDTH,
45                             fill='black')
46
47     # Начертить сектор 4.
48     self.canvas.create_arc(self._X1, self._Y1, self._X2,
49                             self._Y2, start=self._PIE4_START,
50                             extent=self._PIE4_WIDTH,
51                             fill='yellow')
52
53     # Упаковать холст.
54     self.canvas.pack()
55
56     # Запустить главный цикл.
57     tkinter.mainloop()
58
59 # Создать экземпляр класса MyGUI.
60 if __name__ == '__main__':
61     my_gui = MyGUI()
```

Рассмотрим метод `__init__()` класса `MyGUI` в программе 13.29.

- ◆ Строки 6 и 7 задают атрибуты ширины и высоты виджета `Canvas`.
- ◆ Строки 8–11 задают атрибуты координат левого верхнего и правого нижнего углов ограничивающего прямоугольника, которые каждая дуга делит между собой.
- ◆ Строки 12–19 задают атрибуты каждого исходного угла и протяженности сектора круга.
- ◆ Стока 22 создает главное окно, а строки 25–27 — виджет `Canvas`.
- ◆ Строки 30–33 создают первый сектор круга, назначая ему красный цвет заливки.
- ◆ Строки 36–39 создают второй сектор круга, назначая ему зеленый цвет.
- ◆ Строки 42–45 создают третий сектор круга, назначая ему черный цвет заливки.
- ◆ Строки 48–51 создают четвертый сектор круга, назначая ему желтый цвет заливки.
- ◆ Стока 54 упаковывает виджет `Canvas`, делая видимым его содержимое, а строка 57 запускает функцию `mainloop` модуля `tkinter`.

Рисование многоугольников: метод *create_polygon()*

Метод `create_polygon()` чертит замкнутый многоугольник на виджете `Canvas`. Многоугольник строится из многочисленных соединенных отрезков. Точка, где два отрезка соединяются, называется *вершиной*. Вот общий формат вызова метода рисования многоугольника:

имя холста.create polygon(x1, y1, x2, y2, опции...)

Аргументы $x1$ и $y1$ — это координаты (X , Y) первой вершины, $x2$ и $y2$ — координаты (X , Y) второй вершины и т. д. Данный метод автоматически замыкает многоугольник, проведя отрезок из последней вершины в первую вершину. В приведенном выше общем формате в [опциях...](#) указывается несколько необязательных именованных аргументов, которые можно передавать в этот метод. Некоторые из них мы рассмотрим в табл. 13.10.

Программа 13.30 демонстрирует метод `create_polygon()`. Инструкция в строках 13 и 14 чертит многоугольник с восемью вершинами. Первая вершина находится в координате (60, 20), вторая вершина — в координате (100, 20) и т. д. (рис. 13.56). На рис. 13.57 показано окно, которое данная программа выводит на экран.

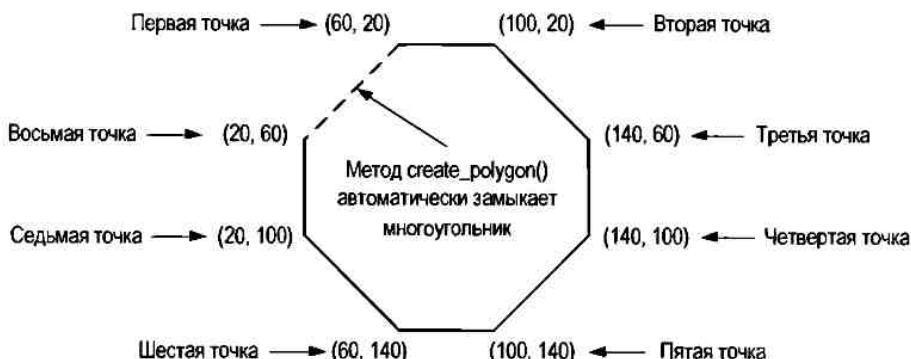


Рис. 13.56. Точки каждой вершины в многоугольнике

Программа 13.30 (draw_polygon.py)

```

16     # Упаковать холст.
17     self.canvas.pack()
18
19     # Запустить главный цикл.
20     tkinter.mainloop()
21
22 # Создать экземпляр класса MyGUI.
23 if __name__ == '__main__':
24     my_gui = MyGUI()

```

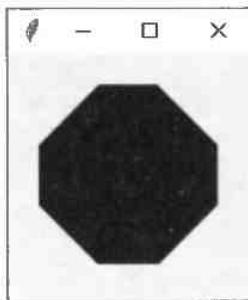


РИС. 13.57. Окно, выводимое на экран программой 13.30

В метод `create_polygon()` можно передавать несколько необязательных именованных аргументов. В табл. 13.10 приведены некоторые наиболее часто используемые из них.

Таблица 13.10. Некоторые необязательные аргументы для метода `create_polygon()`

Аргумент	Описание
<code>dash=значение</code>	Этот аргумент делает контур многоугольника пунктирным. Его значением является кортеж из целых чисел, которые задают шаблон. Первое целое число — количество рисуемых пикселов, второе целое число — количество пропускаемых пикселов и т. д. Например, аргумент <code>dash=(5, 2)</code> нарисует 5 пикселов, пропустит 2 пикселя и будет повторяться до тех пор, пока контур не замкнется
<code>fill=значение</code>	Задает цвет, которым многоугольник может быть заполнен. Значением этого аргумента является имя цвета в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>fill</code> , то многоугольник будет заполнен черным цветом.)
<code>outline=значение</code>	Задает цвет контура многоугольника в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приводится их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>outline</code> , то цвет контура дуги по умолчанию будет черным.)
<code>smooth=значение</code>	По умолчанию аргумент <code>smooth</code> равен <code>False</code> , в результате чего данный метод чертит прямые отрезки, соединяющие указанные точки. Если задать <code>smooth=True</code> , то будут нанесены изогнутые сплайны

Таблица 13.10 (окончание)

Аргумент	Описание
width=значение	Задает ширину контура многоугольника в пикселях. Например, аргумент width=5 нанесет линию контура шириной 5 пикселов. По умолчанию линия контура многоугольника имеет ширину 1 пиксель

Рисование текста: метод `create_text()`

Для нанесения текста на виджете `Canvas` используется метод `create_text()`. Вот общий формат вызова этого метода:

`имя_холста.create_text(x, y, text=текст, опции...)`

Аргументы `x` и `y` — это координаты (X, Y) точки вставки текста, аргумент `text=текст` задает наносимый текст. По умолчанию текст центрируется горизонтально и вертикально вокруг точки вставки. В приведенном выше общем формате в `опциях...` указывается несколько необязательных именованных аргументов, которые можно передавать в этот метод. Некоторые из них мы рассмотрим в табл. 13.11.

Программа 13.31 демонстрирует метод `create_text()`. Инструкция в строке 13 выводит текст 'Привет, мир!' в центре окна в координатах (100, 100). На рис. 13.58 показано это окно.

Программа 13.31 (draw_text.py)

```

1 # Эта программа наносит текст на виджет Canvas.
2 import tkinter
3
4 class MyGUI:
5     def __init__(self):
6         # Создать главное окно.
7         self.main_window = tkinter.Tk()
8
9         # Создать виджет Canvas.
10        self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
11
12        # Показать текст в центре окна.
13        self.canvas.create_text(100, 100, text='Привет, мир!')
14
15        # Упаковать холст.
16        self.canvas.pack()
17
18        # Запустить главный цикл.
19        tkinter.mainloop()
20
21 # Создать экземпляр класса MyGUI.
22 if __name__ == '__main__':
23     my_gui = MyGUI()

```

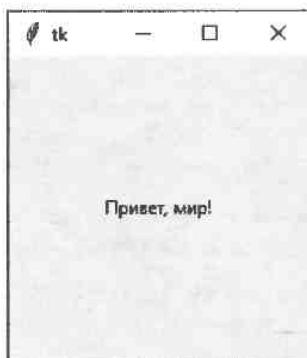


РИС. 13.58. Окно, выводимое на экран программой 13.31

В метод `create_text()` можно передавать несколько необязательных именованных аргументов. В табл. 13.11 приведены некоторые наиболее часто используемые из них.

Таблица 13.11. Несколько необязательных аргументов для метода `create_text()`

Аргумент	Описание
<code>anchor=значение</code>	Этот аргумент задает расположение текста относительно его точки вставки. По умолчанию аргумент привязки <code>anchor</code> равен <code>tkinter.CENTER</code> , что позволяет центрировать текст вертикально и горизонтально вокруг точки вставки. Можно задать любое из нескольких значений, перечисленных в табл. 13.12
<code>fill=значение</code>	Задает цвет, которым текст может быть нарисован. Значением этого аргумента является имя цвета в виде строкового значения. Можно использовать самые разные предопределенные имена цветов, и в <i>приложении 4</i> приведен их полный список. Вот наиболее часто используемые: 'red', 'green', 'blue', 'yellow' и 'cyan'. (Если опустить аргумент <code>fill</code> , то текст будет заполнен черным цветом.)
<code>font=значение</code>	Для того чтобы изменить стандартный шрифт, следует создать объект <code>tkinter.font.Font</code> и передать его как значение аргумента <code>font</code> (применение шрифтов см. в следующем разделе)
<code>justify=значение</code>	Если выводится несколько строк текста, то этот аргумент задает выравнивание строк. Он может принимать значения: <code>tk.LEFT</code> , <code>tk.CENTER</code> или <code>tk.RIGHT</code> . По умолчанию используется значение <code>tk.LEFT</code>

Текст позиционируется относительно своей точки вставки девятью разными способами. Для изменения позиционирования текста используется аргумент `anchor=значение`. Разные значения данного аргумента перечислены в табл. 13.12. По умолчанию используется значение `tkinter.CENTER`.

Таблица 13.12. Значения привязки

Аргумент <code>anchor</code>	Описание
<code>anchor=tkinter.CENTER</code>	Позиционирует текст, центрированным вертикально и горизонтально вокруг точки вставки. Этот вариант позиционирования используется по умолчанию

Таблица 13.12 (окончание)

Аргумент <code>anchor</code>	Описание
<code>anchor=tkinter.NW</code>	Позиционирует текст таким образом, что точка вставки находится в левом верхнем углу текста (на северо-западе)
<code>anchor=tkinter.N</code>	Позиционирует текст таким образом, что точка вставки центрируется вдоль верхнего края текста (на севере)
<code>anchor=tkinter.NE</code>	Позиционирует текст таким образом, что точка вставки находится в правом верхнем углу текста (на северо-востоке)
<code>anchor=tkinter.W</code>	Позиционирует текст таким образом, что точка вставки находится с левого края текста в середине (на западе)
<code>anchor=tkinter.E</code>	Позиционирует текст таким образом, что точка вставки находится с правого края текста в середине (на востоке)
<code>anchor=tkinter.SW</code>	Позиционирует текст таким образом, что точка вставки находится в левом нижнем углу текста (на юго-западе)
<code>anchor=tkinter.S</code>	Позиционирует текст таким образом, что точка вставки центрируется вдоль нижнего края текста (на юге)
<code>anchor=tkinter.SE</code>	Позиционирует текст таким образом, что точка вставки находится в правом нижнем углу текста (на юго-востоке)

На рис. 13.59 показаны различные позиции привязки. В каждой строке текста имеется точка, которая представляет позицию точки вставки.

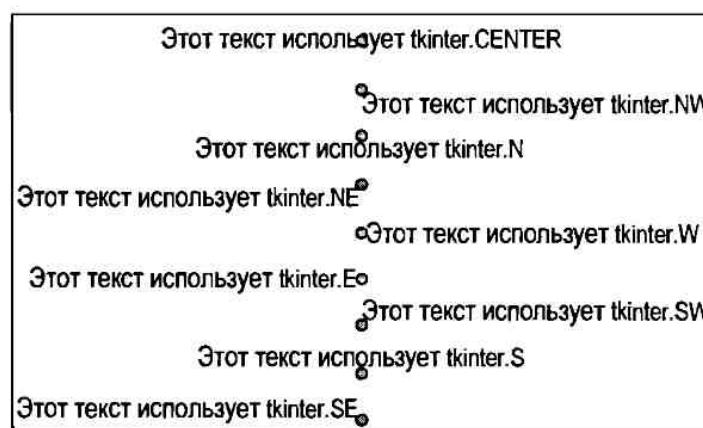


РИС. 13.59. Результаты различных значений привязки

Настройка шрифта

Используемый в методе `create_text()` шрифт задается путем создания объекта `Font` и его передачи в качестве аргумента `font=`. Класс `Font` находится в модуле `tkinter.font`, поэтому в программу необходимо включить приведенную ниже инструкцию `import`:

```
import tkinter.font
```

Вот пример создания объекта `Font`, который задает шрифт `Helvetica` размером 12 пунктов:

```
myfont = tkinter.font.Font(family='Helvetica', size='12')
```

При конструировании объекта `Font` можно передавать значения для любого из именованных аргументов, приведенных в табл. 13.13.

Таблица 13.13. Именованные аргументы для класса `Font`

Аргумент	Описание
<code>family=значение</code>	Этот аргумент является строковым значением, которым задается имя семейства шрифтов, в частности 'Arial', 'Courier', 'Helvetica', 'Times New Roman' и т. д.
<code>size=значение</code>	Этот аргумент является целым числом, которым задается размер шрифта в точках
<code>weight=значение</code>	Этот аргумент задает толщину шрифта. Допустимыми строковыми значениями являются 'bold' и 'normal'
<code>slant=значение</code>	Этот аргумент задает наклон шрифта. Для того чтобы шрифт выглядел наклонным, следует задать значение 'italic'. Для того чтобы шрифт выглядел прямым, следует указать значение 'roman'
<code>underline=значение</code>	Для того чтобы текст выглядел подчеркнутым, следует задать значение 1. В противном случае следует указать значение 0
<code>overstrike=значение</code>	Для того чтобы текст выглядел перечеркнутым, следует задать значение 1. В противном случае следует указать значение 0

Доступные имена семейств шрифтов разнятся в зависимости от используемой операционной системы. Для того чтобы получить список установленных вами семейств шрифтов, в оболочке Python следует ввести приведенные ниже инструкции:

```
>>> import tkinter
>>> import tkinter.font
>>> tkinter.Tk()
<tkinter.Tk object .>
>>> tkinter.font.families()
```

В программе 13.32 приведен пример вывода текста жирным шрифтом `Helvetica` размером 18 пунктов. На рис. 13.60 показано окно, которое данная программа выводит на экран.

Программа 13.32 (font_demo.py)

```
1 # Эта программа наносит текст на виджет Canvas.
2 import tkinter
3 import tkinter.font
4
5 class MyGUI:
6     def __init__(self):
7         # Создать главное окно.
8         self.main_window = tkinter.Tk()
9
```

```

10     # Создать виджет Canvas.
11     self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
12
13     # Создать объект Font.
14     myfont = tkinter.font.Font(family='Helvetica', size=18, weight='bold')
15
16     # Показать немного текста.
17     self.canvas.create_text(100, 100, text='Привет, мир!', font=myfont)
18
19     # Упаковать холст.
20     self.canvas.pack()
21
22     # Запустить главный цикл.
23     tkinter.mainloop()
24
25 # Создать экземпляр класса MyGUI.
26 if __name__ == '__main__':
27     my_gui = MyGUI()

```

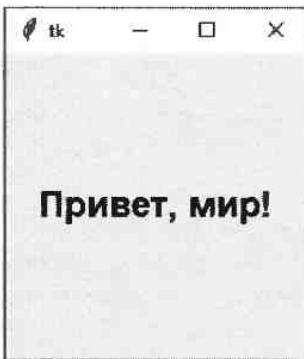


РИС. 13.60. Окно, выводимое на экран программой 13.32



Контрольная точка

- 13.29. Каковы координаты пикселя в левом верхнем углу окна в экранной системе координат виджета `Canvas`?
- 13.30. Каковы координаты пикселя в правом нижнем углу, если пользоваться экранной системой координат виджета `Canvas` с окном шириной 640 и высотой 480 пикселов?
- 13.31. Чем отличается экранная система координат виджета `Canvas` от декартовой системы координат, используемой в библиотеке черепашьей графики?
- 13.32. Какие методы виджета `Canvas` следует использовать для начертания каждого из приведенных ниже типов фигур?
 - а) круг;
 - б) квадрат;
 - в) прямоугольник;

- г) замкнутая шестигранная фигура;
- д) эллипс;
- е) дуга.

Вопросы для повторения

Множественный выбор

1. _____ является частью компьютера, с которой взаимодействует пользователь.
 - а) центральный процессор;
 - б) пользовательский интерфейс;
 - в) система управления;
 - г) система интерактивности.
2. Прежде чем GUI стали популярными, чаще всего использовался _____.
 - а) интерфейс командной строки;
 - б) интерфейс удаленного терминала;
 - в) сенсорный интерфейс;
 - г) событийно-управляемый интерфейс.
3. _____ — это небольшое окно, которое выводит на экран информацию и позволяет пользователю выполнять действия.
 - а) меню;
 - б) окно подтверждения;
 - в) стартовый экран;
 - г) диалоговое окно.
4. Эти типы программ являются событийно-управляемыми.
 - а) консольная программа;
 - б) текст-ориентированная программа;
 - в) программа с GUI;
 - г) процедурная программа.
5. Элемент, который появляется в графическом интерфейсе пользователя программы, называется _____.
 - а) гаджетом;
 - б) виджетом;
 - в) инструментом;
 - г) объектом, свернутым в значок на рабочем столе.
6. Этот модуль используется в Python для создания программ с GUI.
 - а) GUI;
 - б) PythonGui;

- в) tkinter;
г) tgui.
7. Этот виджет представляет собой область, в которой выводится одна строка текста.
- а) Label;
б) Entry;
в) TextLine;
г) Canvas.
8. Этот виджет представляет собой область, в которой пользователь может ввести одну строку, набираемую с клавиатуры.
- а) Label;
б) Entry;
в) TextLine;
г) Input.
9. Этот виджет представляет собой контейнер, который может содержать другие виджеты.
- а) Grouper;
б) Composer;
в) Fence;
г) Frame.
10. Этот метод размещает виджет в своей соответствующей позиции и делает его видимым при выводе главного окна на экран.
- а) pack;
б) arrange;
в) position;
г) show.
11. _____ является функцией или методом, которые вызываются, когда происходит определенное событие.
- а) функция обратного вызова;
б) автоматическая функция;
в) функция запуска;
г) исключение.
12. Функция showinfo находится в модуле _____.
- а) tkinter;
б) tkinfo;
в) sys;
г) tkinter.messagebox.

13. Этот метод следует вызывать для закрытия программы с GUI.
- метод `destroy` корневого виджета;
 - метод `cancel` любого виджета;
 - функцию `sys.shutdown`;
 - метод `Tk.shutdown`.
14. Метод _____ следует вызывать для извлечения данных из виджета `Entry`.
- `gata_entry`;
 - `data`;
 - `get`;
 - `retrieve`.
15. Объект этого типа может быть привязан к виджету `Label`, и любые данные, хранящиеся в этом объекте, будут выводиться в элементе `Label`.
- `StringVar`;
 - `LabelVar`;
 - `LabelValue`;
 - `DisplayVar`.
16. Если в контейнере имеется группа таких элементов, то только один из них может быть выбран в любой момент времени.
- `Checkbutton`;
 - `Radiobutton`;
 - `Mutualbutton`;
 - `Button`.
17. Виджет _____ предоставляет методы для рисования простых двумерных фигур.
- `Shape`;
 - `Draw`;
 - `Palette`;
 - `Canvas`.

Истина или ложь

- Язык Python имеет встроенные ключевые слова для создания программ с GUI.
- Каждый виджет имеет метод `quit()`, который можно вызывать с целью закрытия программы.
- Данные, извлекаемые из виджета `Entry`, всегда имеют тип `int`.
- Среди всех виджетов `Radiobutton`, находящихся в том же самом контейнере, автоматически создается взаимоисключающая связь.

5. Среди всех виджетов `Checkbutton`, находящихся в том же самом контейнере, автоматически создается взаимоисключающая связь.

Короткий ответ

1. Что определяет порядок, в котором все происходит, когда программа выполняется в текст-ориентированной среде, такой как интерфейс командной строки?
2. Что делает метод `pack()` виджета?
3. Что делает функция `mainloop` модуля `tkinter`?
4. Каким образом будут расположены виджеты в их родительском виджете, если создать эти виджеты и вызвать их методы `pack()` без аргументов?
5. Каким образом указать, что виджет должен быть расположен в максимальной левой позиции в своем родительском виджете?
6. Как извлечь данные из виджета `Entry`?
7. Каким образом используется объект `StringVar` для обновления содержимого виджета `Label`?
8. Каким образом используется объект `IntVar` для определения, какой именно виджет `Radiobutton` был выбран в группе виджетов `Radiobutton`?
9. Каким образом используется объект `IntVar` для определения, был выбран виджет `Checkbutton` или нет?

Алгоритмический тренажер

1. Напишите инструкцию, которая создает виджет `Label`. Его родительским виджетом должен быть виджет `self.main_window`, и он должен содержать текст 'Программировать – это круто! '.
2. Допустим, что `self.label1` и `self.label2` ссылаются на два виджета `Label`. Напишите фрагмент кода, который упаковывает эти два виджета таким образом, чтобы они были расположены в своем родительском виджете максимально слева.
3. Напишите инструкцию, которая создает виджет `Frame`. Его родительским виджетом должен быть виджет `self.main_window`.
4. Напишите инструкцию, которая выводит на экран информационное диалоговое окно с заголовком "Программа приостановлена" и сообщением "Нажмите OK, когда будете готовы продолжить".
5. Напишите инструкцию, которая создает виджет `Button`. Его родительским виджетом должен быть `self.button_frame`, его текст должен содержать строковый литерал 'Вычислить', а его функцией обратного вызова должен быть метод `self.calculate()`.
6. Напишите инструкцию, которая создает виджет `Button`, закрывающий программу при его нажатии. Его родительским виджетом должен быть виджет `self.button_frame`, и он должен содержать текст 'Выход'.
7. Допустим, что переменная `data_entry` ссылается на виджет `Entry`. Напишите инструкцию, которая извлекает значение из этого виджета, приводит его к типу `int` и присваивает его переменной с именем `var`.

8. Допустим, что в программе приведенная ниже инструкция создает виджет `Canvas` и присваивает его переменной `self.canvas`:

```
self.canvas = tkinter.Canvas(self.main_window, width=200, height=200)
```

Напишите инструкции, которые делают следующее:

- чертят синюю прямую из левого верхнего угла виджета `Canvas` в его правый нижний угол, прямая должна быть шириной 3 пикселя;
- чертят прямоугольник с красным контуром и черным заполнением, углы прямоугольника должны располагаться на холсте в приведенных ниже позициях:
 - левый верхний: (50, 50);
 - правый верхний: (100, 50);
 - левый нижний: (50, 100);
 - правый нижний: (100, 100);
- чертят зеленый круг, центральная точка круга должна быть в координатах (100, 100), а ее радиус должен равняться 50;
- чертят заполненную синим цветом дугу, заданную ограничивающим прямоугольником, чей левый верхний угол находится в координатах (20, 20), правый нижний угол — в координатах (180, 180). Дуга должна начинаться в 0° и простираться на 90° .

Упражнения по программированию

1. **ФИО и адрес.** Напишите программу с GUI, которая при нажатии кнопки выводит на экран ваше полное имя и адрес. При запуске программы ее окно должно выглядеть так, как на эскизе с левой стороны рис. 13.61. Когда пользователь нажимает кнопку **Показать инфо**, программа должна вывести на экран ваше имя и адрес, как показано на эскизе справа.

 Видеозапись "Задача с ФИО и адресом" (Name and Address Problem)

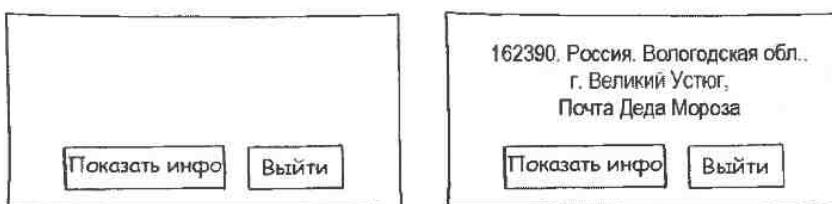


РИС. 13.61. Программа "ФИО и адрес"

2. **Переводчик с латинского.** Взгляните на приведенный в табл. 13.14 список латинских слов и их значений.

Таблица 13.14

Латинский	Русский
sinister	Левый
dexter	Правый
medium	Центральный

Напишите программу с GUI, которая переводит латинские слова на русский язык. Окно должно иметь три кнопки, по одной для каждого латинского слова. Когда пользователь нажимает кнопку, программа должна выводить на экран русский перевод в виджет Label.

3. **Калькулятор миль на галлон бензина.** Напишите программу с GUI, которая вычисляет экономичность автомобиля. Окно программы должно содержать виджеты Entry, которые позволяют пользователю вводить объем бензина в галлонах, заправленного в автомобиль, и число миль, которые он может пройти с полным баком. При нажатии кнопки **Вычислить MPG** программа должна вывести на экран число миль, которые автомобиль может пройти в расчете на галлон бензина. Для вычисления показателя числа миль на галлон примените приведенную ниже формулу:

$$\text{Показатель миль на галлоны} = \frac{\text{мили}}{\text{галлоны}}.$$

4. **Из шкалы Цельсия в шкалу Фаренгейта.** Напишите программу с GUI, которая преобразует показания температуры по шкале Цельсия в температуру по шкале Фаренгейта. Пользователь должен иметь возможность вводить температуру по шкале Цельсия, нажимать кнопку и затем получать эквивалентную температуру по шкале Фаренгейта. Для выполнения этого преобразования примените приведенную ниже формулу:

$$F = \frac{9}{5}C + 32,$$

где F — это температура по Фаренгейту; C — температура по шкале Цельсия.

5. **Налог на недвижимость.** Территориальный округ собирает налоги на недвижимое имущество, опираясь на оценочную стоимость имущества, которая составляет 60% фактической стоимости недвижимого имущества. Если акр земли оценивается в \$10 000, то его оценочная стоимость составляет \$6000. Налог на имущество в таком случае составит \$0.75 для каждого \$100 оценочной стоимости. Налог на акр, оцененный в \$6000, составит \$45.00. Напишите программу с GUI, которая выводит на экран оценочную стоимость и налог на недвижимое имущество при вводе пользователем фактической стоимости недвижимого имущества.

6. **Авторемонтная фирма "Автоцех".** Авторемонтная фирма "Автоцех" предлагает услуги по регламентному техобслуживанию:

- замена масла — 500.00 руб.;
- смазочные работы — 300.00 руб.;
- промывка радиатора — 700.00 руб.;
- замена жидкости в трансмиссии — 1000.00 руб.;
- осмотр — 800.00 руб.;
- замена глушителя выхлопа — 1300.00 руб.;
- перестановка шин — 1300.00 руб.

Напишите программу с GUI с использованием флаговых кнопок, которые позволяют пользователю выбирать любые из этих видов услуг. При нажатии пользователем кнопки **Показать затраты** должна быть выведена общая стоимость услуг.

7. **Междугородные звонки.** Провайдер междугородних звонков взимает плату за телефонные вызовы в соответствии с приведенными в табл. 13.15 тарифами.

Таблица 13.15

Категория тарифа	Тариф в минуту, руб.
Дневное время (с 6:00 до 17:59)	10
Вечернее время (с 18:00 до 23:59)	12
Непиковый период (с полуночи до 5:59)	5

Напишите приложение с GUI, которое позволяет пользователю выбирать категорию уровня (из набора радиокнопок) и вводить в виджет Entry продолжительность вызова в минутах. Информационное диалоговое окно должно выводить на экран стоимость вызова.

8. **Рисунок старого дома.** Примените виджет Canvas, с которым вы познакомились в этой главе, чтобы нарисовать дом. Рисунок дома должен содержать по меньшей мере два окна и дверь. Можно добавить и другие объекты, такие как небо, солнце и даже облака.
9. **Возраст дерева.** Подсчет годичных колец дерева дает довольно точное представление о его возрасте. Каждое годичное кольцо образуется за один год. Примените виджет Canvas, чтобы показать на рисунке, как могли бы выглядеть годичные кольца 5-летнего дерева. Затем, используя метод `create_text()`, пронумеруйте каждое годичное кольцо, начиная с центра и далее продолжая наружу, указывая возраст в годах, связанный с этим кольцом.
10. **Голливудская звезда.** Создайте собственную звезду на Аллее славы в Голливуде. Напишите программу, которая выводит на экран звезду, похожую на приведенную на рис. 13.62, с вашим именем в середине.



РИС. 13.62. Голливудская звезда

11. **Контур транспортного средства.** Используя геометрические фигуры, создавать которые вы научились в этой главе, начертите контур транспортного средства по своему выбору (автомобиль, грузовик, самолет и т. д.).
12. **Солнечная система.** Примените виджет Canvas для создания рисунка всех планет Солнечной системы. Сначала нарисуйте Солнце, затем остальные планеты в соответствии с расстоянием от него (Меркурий, Венеру, Землю, Марс, Юпитер, Сатурн, Уран, Нептун и карликовую планету Плутон). Возле каждой планеты поместите надпись, используя метод `create_text()`.

14.1

Системы управления базами данных

Ключевые положения

Система управления базами данных (СУБД) — это программное обеспечение, которое управляет крупными коллекциями данных.

Если приложению необходимо хранить только небольшой объем данных, то традиционные файлы с этой задачей справляются хорошо. Однако эти файлы становятся непрактичными, когда необходимо хранить и обрабатывать большой объем данных. Во многих компаниях в файлах хранятся миллионы элементов данных. Когда традиционный файл содержит такое количество данных, простые операции, например поиск, вставка и удаление, становятся громоздкими и неэффективными.

При разработке приложений, работающих с большими объемами данных, большинство разработчиков предпочитают вместо традиционных файлов использовать систему управления базами данных. *Система управления базами данных* (СУБД) — это программное обеспечение, специально разработанное для организованного и эффективного хранения, извлечения и управления большими объемами данных. На языке Python или другом языке программирования можно написать приложение, которое будет использовать СУБД для управления данными. Вместо того чтобы извлекать данные или манипулировать ими напрямую, приложение может отправлять инструкции в СУБД. А та, в свою очередь, выполняет эти инструкции и отправляет результаты обратно в приложение (рис. 14.1).

Хотя рис. 14.1 упрощен, он иллюстрирует многоуровневую архитектуру приложения, работающего с СУБД. Самый верхний уровень программного обеспечения, который в данном случае написан на Python, взаимодействует с пользователем. Он также отправляет инструкции на следующий уровень — СУБД. СУБД работает непосредственно с данными и отправляет результаты операций обратно в приложение.

Предположим, компания хранит все записи о своей продукции в базе данных. У компании есть приложение на Python, которое позволяет пользователю искать информацию о любом изделии, вводя его идентификационный номер. Приложение Python инструктирует СУБД, что нужно извлечь запись для изделия с указанным идентификатором (ID). СУБД извлекает запись изделия и отправляет данные обратно в приложение Python. Приложение Python отображает данные пользователю.

Преимущество такого многоуровневого подхода к разработке программного обеспечения заключается в том, что Python-программисту не нужно беспокоиться о том, как данные хранятся на диске, и алгоритмах, которые используются для хранения и извлечения данных. Программисту следует знать только, как взаимодействовать с СУБД. СУБД манипулирует фактическим чтением, записью и поиском данных.



РИС. 14.1. Приложение на Python, взаимодействующее с СУБД, которая манипулирует данными

SQL

Аббревиатура SQL расшифровывается как Structured Query Language, т. е. язык структурированных запросов. Указанный язык является стандартным для работы с СУБД. Первоначально он был разработан компанией IBM в 1970-х годах. С тех пор SQL был принят большинством поставщиков программного обеспечения для баз данных в качестве предпочтительного языка для взаимодействия с СУБД.

SQL состоит из нескольких ключевых слов, которые используются для конструирования инструкций. Инструкции SQL передаются в СУБД и являются для СУБД командами по выполнению операций с ее данными. Когда приложение Python взаимодействует с СУБД, оно должно создавать инструкции SQL в виде строк, а затем использовать библиотечный метод для передачи этих строк в СУБД. В этой главе вы узнаете, как создавать простые инструкции SQL, а затем передавать их в СУБД с помощью библиотечного метода.

ПРИМЕЧАНИЕ

Хотя SQL и является языком, он не предназначен для написания приложений. Он используется только в качестве стандартного средства взаимодействия с СУБД. Вам по-прежнему будет нужен общий язык программирования, такой как Python, чтобы написать приложение для обычного пользователя.

SQLite

В настоящее время используется целый ряд СУБД, и Python может взаимодействовать со многими из них. Несколько наиболее популярных среди них — это MySQL, Oracle, Microsoft SQL Server, PostgreSQL и SQLite. В этой книге мы опираемся на СУБД SQLite, потому что она проста в использовании и инсталлируется в системе автоматически при установке Python. В целях использования СУБД SQLite вместе с Python необходимо импортировать модуль `sqlite3` с помощью следующей инструкции:

```
import sqlite3
```



Контрольная точка

- 14.1. Что такое система управления базами данных (СУБД)?
- 14.2. Почему большинство компаний используют СУБД для хранения своих данных вместо создания собственных текстовых файлов для хранения данных?
- 14.3. Почему программисту при разработке приложения на Python, использующего СУБД для хранения и обработки данных, не нужно знать конкретные детали о физической структуре данных?
- 14.4. Что такое SQL?
- 14.5. Как программа Python отправляет инструкции SQL в СУБД?
- 14.6. Какая инструкция `import` требуется для того, чтобы использовать SQLite в среде Python?

14.2 Таблицы, строки и столбцы

Ключевые положения

Данные, хранящиеся в базе данных, организованы в таблицы, строки и столбцы.

СУБД хранит данные в *базе данных*. Данные, хранящиеся в базе данных, организованы в одну или несколько таблиц. Каждая *таблица* содержит набор связанных данных. Данные, хранящиеся в таблице, упорядочиваются в строки и столбцы. *Строка* — это полный набор данных об одном элементе. Данные, хранящиеся в строке, делятся на столбцы. Каждый *столбец* содержит отдельный фрагмент данных об элементе. Например, предположим, что мы разрабатываем приложение по ведению списка контактов и хотим хранить список имен и телефонных номеров в базе данных.

Первоначально мы храним следующий список:

Кэти Аллен	555-1234
Джилл Аммонс	555-5678
Кевин Браун	555-9012
Элиза Гарсия	555-3456
Джефф Дженкинс	555-7890
Лео Киллиан	555-1122
Марсия Потемкин	555-3344
Келси Роуз	555-5566

Подумайте о том, как выглядели бы эти данные, если бы мы хранили их в виде строк и столбцов в электронной таблице. Мы бы поместили имена в один столбец, а номера телефонов — в другой. Таким образом, каждая строка будет содержать данные об одном человеке. На рис. 14.2 выделена третья строка, содержащая имя и номер телефона Кевина Брауна.

Когда мы создаем таблицу базы данных для хранения этой информации, мы организуем ее аналогичным образом. Мы даем таблице имя, например `Contacts` (Контакты). В таблице мы создаем столбец для имен и столбец для телефонных номеров. Каждый столбец в таблице должен иметь имя, поэтому мы можем назвать наши столбцы соответственно `Name` (Имя) и `Phone` (Телефон).



Name	Phone
Кэти Аллен	555-1234
Джилл Аммонс	555-5678
Кевин Браун	555-9012
Элиза Гарсия	555-3456
Джефф Дженкинс	555-7890
Лео Киллиан	555-1122
Марсия Потемкин	555-3344
Келси Роуз	555-5566

РИС. 14.2. Имена и номера телефонов, хранящиеся в таблице

Типы данных в столбцах

При создании таблицы базы данных необходимо указать тип данных для столбцов. Однако типы данных, которые вы можете выбрать, не являются типами данных Python. Это типы данных, предоставляемые СУБД. В этой книге мы используем SQLite, поэтому выберем один из типов данных, предлагаемых этой СУБД. В табл. 14.1 перечислены типы данных SQLite и указан тип данных Python, с которым каждый из них в целом совместим.

Таблица 14.1. Типы данных SQLite

Тип данных SQLite	Описание	Соответствующий тип данных Python
NULL	Неизвестное значение	None
INTEGER	Целое число	int
REAL	Вещественное число	float
TEXT	Строковое значение	str
BLOB	Двоичный большой объект	Может быть любым объектом

Вот краткое описание каждого из этих типов данных.

- ◆ NULL — этот тип данных может использоваться, если значение неизвестно или отсутствует. Когда Python читает значение столбца NULL в память, это значение конвертируется в значение None.
- ◆ INTEGER — этот тип данных содержит целочисленное значение со знаком. Когда Python читает значение INTEGER столбца в память, это значение конвертируется в значение типа int.
- ◆ REAL — этот тип данных содержит действительное, или вещественное, число. Когда Python читает значение REAL столбца в память, это значение конвертируется в значение типа float.
- ◆ TEXT — этот тип данных содержит строковое значение. Когда Python читает значение TEXT столбца в память, это значение конвертируется в значение типа str.

- ◆ BLOB — этот тип данных содержит объект любого типа, например массив или изображение. Когда Python читает значение столбца BLOB в память, это значение конвертируется в объект bytes, который представляет собой немутируемую последовательность байтов.

Первичные ключи

Таблицы базы данных обычно имеют *первичный ключ*, представляющий собой столбец, который можно использовать для идентификации той или иной строки в таблице. Столбец, назначенный в качестве первичного ключа, должен содержать уникальное значение для каждой строки. Вот несколько примеров.

- ◆ В таблице хранятся данные о сотрудниках, а в одном из столбцов содержатся идентификационные номера сотрудников. Поскольку идентификационный номер каждого сотрудника уникален, этот столбец можно использовать в качестве первичного ключа.
- ◆ В таблице хранятся данные об изделиях, а в одном из столбцов содержится серийный номер изделия. Поскольку каждое изделие имеет уникальный серийный номер, этот столбец можно использовать в качестве первичного ключа.
- ◆ В таблице хранятся данные счетов-фактур, а в одном из столбцов содержатся их номера. Каждый счет-фактура имеет уникальный номер, поэтому этот столбец можно использовать в качестве первичного ключа.

Вы можете создать таблицу базы данных без первичного ключа. Однако большинство разработчиков согласны с тем, что, за редким исключением, таблицы базы данных всегда должны иметь первичный ключ. Позже в этой главе мы обсудим первичные ключи более подробно, и вы увидите, как их можно использовать для установления связей между несколькими таблицами.

Идентификационные столбцы

Иногда данные, которые вы хотите хранить в таблице, не содержат уникальных элементов, которые можно использовать в качестве первичного ключа. Например, в таблице контактов `Contacts`, которую мы описали ранее, ни столбец `Name`, ни столбец `Phone` не содержат уникальных данных. Два человека могут иметь одно и то же имя, поэтому возможно, что имя появится в столбце `Name` несколько раз. Кроме того, несколько человек могут совместно использовать один и тот же номер телефона, поэтому вполне вероятно, что номер телефона может появиться в столбце `Phone` несколько раз. Следовательно, вы не можете использовать ни столбец `Name`, ни столбец `Phone` в качестве первичного ключа.

В таком случае необходимо создать идентификационный столбец специально для использования в качестве первичного ключа. *Идентификационный столбец* — это столбец, содержащий уникальные значения, созданные СУБД. Идентификационные столбцы обычно содержат целые числа, которые сохраняются *автоматически*. Это означает, что всякий раз, когда в таблицу добавляется новая строка, СУБД автоматически назначает идентификационному столбцу целое число, которое на 1 больше наибольшего значения, хранящегося в данный момент в идентификационном столбце. Естественно, идентификационный столбец будет содержать последовательность значений 1, 2, 3 и т. д.

Например, при разработке таблицы контактов, которую мы обсуждали ранее, мы могли бы создать столбец `INTEGER` с именем `ContactID` и назначить этот столбец в качестве идентифи-

кационного столбца. Как следствие, СУБД будет присваивать столбцу `ContactID` уникальное целочисленное значение для каждой строки.

Затем мы могли бы назначить столбец `ContactID` в качестве первичного ключа таблицы. На рис. 14.3 показан пример таблицы контактов `Contacts` после того, как мы создали ее и ввели в нее данные.

ContactID	Name	Phone
1	Кэти Аллен	555-1234
2	Джилл Амmons	555-5678
3	Кевин Браун	555-9012
4	Элиза Гарсия	555-3456
5	Джефф Дженкинс	555-7890
6	Лео Киплиан	555-1122
7	Марсия Потемкин	555-3344
8	Келси Роуз	555-5566

РИС. 14.3. Таблица контактов с введенными данными



ПРИМЕЧАНИЕ

При добавлении строки, имеющей идентификационный столбец, с помощью SQLite для идентификационного столбца можно указывать значение в явной форме, если это значение является уникальным. Если для идентификационного столбца указать значение, которое уже используется в другой строке, то СУБД выдаст исключение. Если вы не укажете значение для идентификационного столбца, то СУБД сгенерирует уникальное значение и присвоит его идентификационному столбцу.

Разрешение использовать значения `null`

Если столбец не содержит данных, считается, что он имеет значение `NULL`. Иногда можно оставлять столбец пустым. Однако некоторые столбцы, такие как первичные ключи, должны содержать значение. При разработке таблицы можно применять *ограничение*, или правило, которое не позволяет столбцу иметь значение `NULL`. Если конкретному столбцу не разрешено иметь значение `NULL`, то всякий раз при добавлении строки данных в таблицу СУБД будет требовать, чтобы для этого столбца было указано значение. Если оставить столбец пустым, это приведет к ошибке.



Контрольная точка

14.7. Опишите каждый приведенный ниже термин:

- а) база данных;
- б) таблица;
- в) строка;
- г) столбец.

14.8. Сопоставьте тип данных SQLite с совместимым типом данных Python.

- | | |
|------------|------------------------------|
| 1) NULL | а) float |
| 2) INTEGER | б) может быть любым объектом |
| 3) REAL | в) None |
| 4) TEXT | г) int |
| 5) BLOB | д) str |

14.9. Каково назначение первичного ключа?

14.10. Что такое идентификационный столбец?

14.3

Открытие и закрытие соединения с базой данных с помощью SQLite

Ключевые положения

Прежде чем вы сможете работать с базой данных, вы должны к ней подсоединиться. Когда вы закончите работу с базой данных, вы должны закрыть соединение.



Видеозапись "Открытие и закрытие соединения с базой данных"
(*Opening and Closing a Database Connection*)

Типичный процесс использования базы данных SQLite можно обобщить следующим псевдокодом:

Подсоединиться к базе данных
Получить курсор для базы данных
Выполнить операции с базой данных
Зафиксировать изменения в базе данных
Закрыть соединение с базой данных

Давайте рассмотрим каждый шаг в псевдокоде подробнее.

- ◆ **Подключиться к базе данных.** База данных SQLite хранится в файле на системном диске. На этом шаге устанавливается соединение между программой и конкретным файлом базы данных. Если файл базы данных не существует, он будет создан.
- ◆ **Получить курсор для базы данных.** Курсор — это объект, который может получать доступ к данным в базе данных и манипулировать ими.
- ◆ **Выполнить операции с базой данных.** Имея курсор, вы можете получать доступ к данным в базе данных и изменять их по мере необходимости. Вы можете использовать курсор для извлечения данных, вставки новых данных, обновления существующих данных и удаления данных.
- ◆ **Зафиксировать изменения в базе данных.** Когда вы вносите в базу данных изменения, эти изменения фактически не сохраняются в базе данных до тех пор, пока вы их не зафиксируете. После выполнения любых операций, изменяющих содержимое таблицы, следует зафиксировать эти изменения в базе данных.
- ◆ **Закрыть соединение с базой данных.** Когда вы закончите использовать базу данных, вы должны закрыть соединение.

Для подключения к базе данных мы вызываем функцию `connect` модуля `sqlite3`. В следующих ниже интерактивных сессиях показан пример:

```
>>> import sqlite3
>>> conn = sqlite3.connect('contacts.db')
```

Первая инструкция импортирует модуль `sqlite3`. Затем вторая инструкция вызывает функцию `connect` модуля, передавая в качестве аргумента имя нужного файла базы данных `contacts.db`. Функция `connect` откроет соединение с файлом базы данных. Если файл не существует, функция создаст его и установит с ним соединение. Указанная функция возвращает объект `Connection`, на который нам нужно будет ссылаться позже, поэтому он присваивается переменной `conn`.

ПРИМЕЧАНИЕ

Функция `connect` создает файл пустой базы данных, которая не содержит таблиц.

Затем мы вызываем метод `cursor()` объекта `Connection`, чтобы получить курсор для базы данных:

```
>>> cur = conn.cursor()
```

Метод `cursor` возвращает объект `Cursor`, который имеет возможность доступа и изменения базы данных. Мы назначаем объект `Cursor` переменной `cur`. Объект `Cursor` будет использоваться для выполнения операций с таблицами базы данных, таких как извлечение строк, вставка строк, изменение строк и удаление строк.

После завершения работы с базой данных следует вызвать метод `commit()` объекта `Connection`, чтобы сохранить все изменения, внесенные в базу данных. Вот пример:

```
>>> conn.commit()
```

Последний шаг состоит в том, чтобы закрыть соединение с базой данных с помощью метода `close()` объекта `Connection`:

```
>>> conn.close()
```

Программа 14.1 демонстрирует исходный код Python, который выполняет эти шаги.

Программа 14.1 (sqlite_skeleton.py)

```
1 import sqlite3
2
3 def main():
4     conn = sqlite3.connect('contacts.db')
5     cur = conn.cursor()
6
7     # Здесь вставить код для выполнения операций с базой данных.
8
9     conn.commit()
10    conn.close()
11
12 # Вызвать главную функцию.
13 if __name__ == '__main__':
14     main()
```

Указание месторасположения базы данных на диске

Когда в качестве аргумента функции `connect` вы передаете имя файла, не содержащее путь, СУБД предполагает, что месторасположение файла совпадает с расположением программы. Например, предположим, что программа находится в следующей папке на компьютере с ОС Windows:

```
C:\Users\Имя_пользователя\Documents\Python
```

Если программа запущена и выполняет следующую ниже инструкцию, то файл `contacts.db` создается в той же папке:

```
sqlite3.connect('contacts.db')
```

Если вы хотите открыть соединение с файлом базы данных в другом месторасположении, то вы можете указать путь, а также имя файла в аргументе, который вы передаете в функцию `connect`. Если вы указываете путь в строковом литерале (в особенности на компьютере с ОС Windows), то следует предварить строку буквой `r`. Вот пример:

```
sqlite3.connect(r'C:\Users\Имя_пользователя\temp\contacts.db')
```

Напомним из главы 6, что префикс `r` указывает на то, что строка является сырым строковым значением. Это приводит к тому, что интерпретатор Python будет читать символы обратной косой черты буквально, как обратные косые черты. Без префикса `r` интерпретатор предложил бы, что символы обратной косой черты являются частью экранирующих последовательностей, и произошла бы ошибка.

Передача инструкций языка SQL в СУБД

Ранее мы упоминали, что вы выполняете операции с базой данных, создавая инструкции SQL в виде строк, а затем используя библиотечный метод для передачи этих строк в СУБД. С помощью SQLite вы используете метод `execute()` объекта `Cursor` для передачи инструкции языка SQL в СУБД. Вот общий формат вызова метода `execute()`:

```
cur.execute(Строка_SQL)
```

В общем формате `cur` — это имя объекта `Cursor`, а `Строка_SQL` — строковый литерал либо строковая переменная, содержащая инструкцию SQL. Метод `execute()` отправляет строку в СУБД SQLite, которая, в свою очередь, выполняет ее в базе данных.



Контрольная точка

- 14.11. Что такое курсор?
- 14.12. Когда вы получаете курсор для базы данных: до или после подключения к базе данных?
- 14.13. Почему важно фиксировать любые изменения, вносимые в базу данных?
- 14.14. Какую функцию/метод следует вызывать для подключения к базе данных SQLite?
- 14.15. Что происходит при подключении к несуществующей базе данных?
- 14.16. Какой тип объекта используют для доступа и изменения данных в базе данных?
- 14.17. Какой метод вызывают, чтобы получить курсор для базы данных SQLite?
- 14.18. Какой метод вызывают для фиксации изменений в базе данных SQLite?

14.19. Какой метод вызывают, чтобы закрыть соединение с базой данных SQLite?

14.20. Какой метод вызывают для отправки инструкции SQL в СУБД SQLite?

14.4

Создание и удаление таблиц

Ключевые положения

Для создания таблицы в базе данных используется SQL-инструкция CREATE TABLE. Для удаления таблицы применяется SQL-инструкция DROP TABLE.



Видеозапись "Создание таблицы" (Creating a Table)

Создание таблицы

После создания новой базы данных с помощью функции connect модуля `sqlite3` необходимо добавить в базу данных одну или несколько таблиц. Для этого используется SQL-инструкция CREATE TABLE. Вот ее общий формат:

```
CREATE TABLE ИмяТаблицы (ИмяСтолбца1 ТипДанных1, ИмяСтолбца2 ТипДанных2, ...)
```

Здесь *ИмяТаблицы* — это имя создаваемой таблицы; *ИмяСтолбца1* — это имя первого столбца, *ТипДанных* — тип данных SQL для первого столбца; *ИмяСтолбца2* — это имя второго столбца, *ТипДанных2* — тип данных SQL для второго столбца. Эта последовательность повторяется для всех столбцов таблицы. Например, посмотрите на следующую инструкцию SQL:

```
CREATE TABLE Inventory (ItemName TEXT, Price REAL)
```

Эта инструкция создает таблицу *Inventory* (Инструменты). Таблица состоит из двух столбцов: *ItemName* и *Price*. Данные столбца *ItemName* имеют тип `TEXT`, а данные столбца *Price* — тип `REAL`. Однако не хватает одной вещи: первичного ключа. Ранее мы упоминали, что таблицы базы данных обычно имеют первичный ключ, т. е. столбец, содержащий уникальное значение для каждой строки. Первичный ключ позволяет идентифицировать каждую строку в таблице. Можно назначить столбец в качестве первичного ключа, указав ограничение `PRIMARY KEY` после типа данных столбца. Вот общий формат:

```
CREATE TABLE ИмяТаблицы (ИмяСтолбца1 ТипДанных1 PRIMARY KEY,  
                           ИмяСтолбца2 ТипДанных2,  
                           ...)
```

При назначении первичного ключа также рекомендуется использовать ограничение `NOT NULL`. Оно указывает на то, что столбец нельзя оставлять пустым. Вот общий формат:

```
CREATE TABLE ИмяТаблицы (ИмяСтолбца1 ТипДанных1 PRIMARY KEY NOT NULL,  
                           ИмяСтолбца2 ТипДанных2,  
                           ...)
```

При назначении столбца в качестве первичного ключа необходимо обеспечить, чтобы никакие две строки в таблице не могли иметь одинакового значения в этом столбце. Возможна ситуация, что в нашей таблице *Inventory* два элемента могут иметь одно и то же имя. Например, в магазине бытовой техники может быть несколько предметов с названием "отвертка". Кроме того, вполне вероятно, что несколько предметов могут иметь одинаковую цену.

Следовательно, мы не можем использовать ни столбец `ItemName`, ни столбец `Price` в качестве первичного ключа.

Поскольку мы не можем назначить их первичными ключами, добавим в нашу таблицу еще один столбец и будем использовать его в качестве первичного ключа. Давайте добавим столбец с типом `INTEGER` и именем `ItemID`. Вот инструкция SQL для создания таблицы:

```
CREATE TABLE Inventory (ItemID INTEGER PRIMARY KEY NOT NULL,
                       ItemName TEXT,
                       Price REAL)
```

В этой инструкции столбец `ItemID` имеет тип `INTEGER` и указан в качестве первичного ключа. В SQLite любой столбец, обозначенный как `INTEGER` и `PRIMARY KEY`, также станет автоматически сохраняемым идентификаторным столбцом. Это означает, что если при добавлении строки в таблицу значение для столбца `ItemID` явно не указано, то СУБД автоматически сгенерирует для столбца уникальное целочисленное значение. Это значение будет на 1 больше текущего наибольшего значения в идентификатором столбце.



ПРИМЕЧАНИЕ

Важность наличия первичного ключа в таблице базы данных, возможно, пока еще не очевидна. Позже, когда вы начнете создавать связи между несколькими таблицами, вы увидите, что первичные ключи очень важны.



СОВЕТ

Инструкции SQL имеют свободную форму, а значит, символы табуляции, новой строки и пробелы между ключевыми словами игнорируются. Например, инструкция

```
CREATE TABLE Inventory (ItemName TEXT, Price REAL)
```

работает так же, как и:

```
CREATE TABLE Inventory
(
    ItemName TEXT,
    Price REAL
)
```

Кроме того, ключевые слова языка SQL и имена таблиц не чувствительны к регистру. Приведенная выше инструкция может быть написана следующим образом:

```
create table inventory (ItemName text, Price real)
```

В этой книге мы следуем правилу написания ключевых слов SQL прописными буквами, поскольку это визуально отличает инструкции SQL от кода Python.

В целях исполнения нашей SQL-инструкции в Python мы должны передать в метод `execute()` объекта `Cursor` инструкцию в виде строкового значения. Один из способов — присвоить это значение переменной, а затем передать переменную методу `execute()`. В следующем примере предположим, что `cur` является объектом `Cursor`:

```
sql = '''CREATE TABLE Inventory (ItemID INTEGER PRIMARY KEY NOT NULL,
                                 ItemName TEXT,
                                 Price REAL)'''
```

```
cur.execute(sql)
```

Еще один подход состоит в том, чтобы просто передать в метод `execute()` инструкцию SQL в виде строкового литерала. Вот пример:

```
cur.execute('' 'CREATE TABLE Inventory (ItemID INTEGER PRIMARY KEY NOT NULL,
                                         ItemName TEXT,
                                         Price REAL)'''')
```

Обратите внимание, что в обоих приведенных выше примерах мы заключили инструкцию SQL в тройные кавычки. Это связано с тем, что оператор SQL слишком длинный, чтобы поместиться в одной строке кода. Вспомните из главы 2, что строковый литерал в тройных кавычках может охватывать несколько строк кода. В программном коде Python обычно проще написать длинную инструкцию SQL в виде строкового литерала с тройными кавычками, чем писать ее в виде нескольких сцепленных строк.

В программе 14.2 приведен полный исходный код, который подсоединяется к базе данных с именем `inventory.db` и создает таблицу `Inventory`. Программа не выводит на экран никаких результатов.

Программа 14.2 (add_table.py)

```
1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('inventory.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Добавить таблицу Inventory.
11    cur.execute('' 'CREATE TABLE Inventory (ItemID INTEGER PRIMARY KEY NOT NULL,
12                                         ItemName TEXT,
13                                         Price REAL)'''')
14
15    # Зафиксировать изменения.
16    conn.commit()
17
18    # Закрыть соединение.
19    conn.close()
20
21 # Вызвать главную функцию.
22 if __name__ == '__main__':
23     main()
```

Создание нескольких таблиц

Обычно базы данных содержат несколько таблиц. Например, база данных компании может содержать одну таблицу для хранения данных о клиентах, а другую таблицу для хранения данных о сотрудниках.

В программе 14.3 показан пример, в котором создаются две такие таблицы в базе данных. Инструкция в строках 11–13 программы создает таблицу *Customers*, содержащую три столбца: *CustomerID* (ID клиента), *Name* (Имя) и *Email* (Электронная почта). Инструкция в строках 16–18 создает таблицу *Employees* (Служащие), содержащую три столбца: *EmployeeID* (ID служащего), *Name* (Имя) и *Position* (Должность). (Программа не выводит никакого результата на экран.)

Программа 14.3 (multiple_tables.py)

```

1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('company.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Добавить таблицу Customers.
11    cur.execute(''CREATE TABLE Customers (CustomerID INTEGER PRIMARY KEY NOT NULL,
12                                         Name TEXT,
13                                         Email TEXT)'''')
14
15    # Добавить таблицу Employees.
16    cur.execute(''CREATE TABLE Employees (EmployeeID INTEGER PRIMARY KEY NOT NULL,
17                                         Name TEXT,
18                                         Position TEXT)'''')
19
20    # Зафиксировать изменения.
21    conn.commit()
22
23    # Закрыть соединение.
24    conn.close()
25
26 # Вызвать главную функцию.
27 if __name__ == '__main__':
28     main()

```

Создание таблицы, только если она еще не существует

При попытке создать уже существующую таблицу возникнет ошибка. Ради предотвращения ошибки вы можете использовать вот такой формат инструкции CREATE TABLE:

```
CREATE TABLE IF NOT EXISTS ИмяТаблицы (ИмяСтолбца1 ТипДанных1,
                                            ИмяСтолбца2 ТипДанных2, ...)
```

При использовании инструкции CREATE TABLE с выражением IF NOT EXISTS указанная таблица будет создана только в том случае, если она еще не существует. В противном случае инструкция не станет ее создать, и ошибка будет устранена. Вот пример:

```
CREATE TABLE IF NOT EXISTS Inventory (ItemId INTEGER PRIMARY KEY NOT NULL,
                                      ItemName TEXT,
                                      Price REAL)
```

Удаление таблицы

Если вам нужно удалить таблицу, то для этого вы можете воспользоваться SQL-инструкцией `DROP TABLE`. Вот ее общий формат:

```
DROP TABLE ИмяТаблицы
```

Здесь `ИмяТаблицы` — это имя удаляемой таблицы. После выполнения указанной инструкции таблица и все содержащиеся в ней данные будут удалены. Например, если предположить, что `cur` является объектом `Cursor`, то вот пример удаления таблицы `Temp`:

```
>>> cur.execute('DROP TABLE Temp')
```

При попытке удалить уже существующую таблицу возникнет ошибка. В целях предотвращения ошибки вы можете использовать вот такой формат инструкции `DROP TABLE`:

```
DROP TABLE IF EXISTS ИмяТаблицы
```

При использовании инструкции `DROP TABLE` с выражением `IF EXISTS` указанная таблица будет удалена только в том случае, если она существует. Если таблицы нет, инструкция не будет пытаться удалить ее, и ошибка будет устранена.

Пусть `cur` является объектом `Cursor`, тогда вот пример:

```
>>> cur.execute('DROP TABLE IF EXISTS Temp')
```



Контрольная точка

- 14.21. Напишите инструкцию SQL для создания таблицы с именем `Book`. В таблице `Book` должны быть столбцы, содержащие название издателя, имя автора, число страниц и 10-символьный код ISBN.
- 14.22. Напишите инструкцию удаления таблицы `Book`, созданной в контрольном упражнении 14.21.



14.5 Добавление данных в таблицу

Ключевые положения

Инструкция `INSERT` на языке SQL используется для вставки новой строки в таблицу.



Видеозапись "Добавление данных в таблицу" (Adding Data to a table)

Создав файл базы данных и одну или нескольких таблиц в базе данных, можно добавлять строки в таблицы. В SQL инструкция `INSERT` используется для вставки строки в таблицу базы данных. Вот общий формат:

```
INSERT INTO ИмяТаблицы (ИмяСтолбца1, ИмяСтолбца2, ...)
VALUES (Значение1, Значение2, ...)
```

Здесь *ИмяСтолбца1*, *ИмяСтолбца2*, ... — это список имен столбцов; *Значение1*, *Значение2*, ... — список соответствующих значений. В новой строке *Значение1* появится в столбце, указанном в *ИмяСтолбца1*, *Значение2* появится в столбце, указанном в *ИмяСтолбца2*, и т. д.

Предположим, что *cur* является объектом *Cursor* для базы данных *inventory.db*. Вот пример, в котором вставляется строка в таблицу *Inventory*:

```
cur.execute(''INSERT INTO Inventory (ItemID, ItemName, Price)
VALUES (1, "Отвертка", 4.99)'''')
```

Эта инструкция создаст новую строку, содержащую следующие значения столбцов:

```
ItemID: 1
ItemName: Отвертка
Price: 4.99
```

Поскольку столбец *ItemID* одновременно имеет тип *INTEGER* и является первичным ключом (*PRIMARY KEY*), СУБД автоматически сгенерирует для него уникальное целочисленное значение, если мы его не предоставим. Если мы хотим использовать автоматически сгенерированное значение, мы можем просто опустить столбец *ItemID* из инструкции *INSERT*, как показано ниже:

```
cur.execute(''INSERT INTO Inventory (ItemName, Price)
VALUES ("Отвертка", 4.99)'''')
```

Эта инструкция создаст новую строку с автоматически сгенерированным целочисленным значением, присвоенным столбцу *ItemID*, слово "Отвертка" будет назначено столбцу *ItemName*, а цена 4.99 — столбцу *Price*.

Стоит отметить, что приведенная выше инструкция SQL является строковым литералом, а внутри него находится другой строковый литерал — "Отвертка". Инструкция SQL заключена в тройные кавычки, а внутреннее строковое значение заключено в двойные кавычки (рис. 14.4). При желании вы можете использовать тройные кавычки для обрамления инструкции SQL и одинарные кавычки для обрамления внутреннего строкового значения, как показано в следующем примере:

```
cur.execute(''INSERT INTO Inventory (ItemName, Price)
VALUES ('Отвертка', 4.99)'''')
```

Важно помнить, что кавычки, которые вы используете во внутреннем строковом значении, должны отличаться от кавычек, которые вы используете во внешнем строковом значении.



РИС. 14.4. Кавычки, обрамляющие внешний и внутренний строковые литералы

В программе 14.4 представлен пример добавления строк в таблицу `Inventory` в базе данных `inventory.db`. Предполагается, что база данных `inventory.db` уже создана и таблица `Inventory` добавлена в базу данных. Перед запуском программы 14.4 убедитесь, что вы уже добавили таблицу, выполнив программу `add_table.py` (см. программу 14.3).

Программа 14.4 (insert_rows.py)

```
1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('inventory.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Добавить строку в таблицу Inventory.
11    cur.execute(''': INSERT INTO Inventory (ItemName, Price)
12                  VALUES ("Отвертка", 4.99)''')
13
14    # Добавить еще одну строку в таблицу Inventory.
15    cur.execute(''': INSERT INTO Inventory (ItemName, Price)
16                  VALUES ("Молоток", 12.99)''')
17
18    # Добавить еще одну строку в таблицу Inventory.
19    cur.execute(''': INSERT INTO Inventory (ItemName, Price)
20                  VALUES ("Плоскогубцы", 14.99)''')
21
22    # Зафиксировать изменения.
23    conn.commit()
24
25    # Закрыть соединение.
26    conn.close()
27
28 # Вызвать главную функцию.
29 if __name__ == '__main__':
30     main()
```

Давайте рассмотрим программу подробнее. Стока 5 подключается к базе данных, а строка 8 получает объект `Cursor`. Инструкция в строках 11–12 программы вставляет новую строку в таблицу `Inventory` со следующими данными:

`ItemID: (автоматически сгенерированное значение)`

`ItemName: Отвертка`

`Price: 4.99`

Инструкция в строках 15–16 программы вставляет еще одну новую строку в таблицу `Inventory` со следующими данными:

ItemID: *(автоматически сгенерированное значение)*

ItemName: Молоток

Price: 12.99

Инструкция в строках 19–20 программы вставляет еще одну новую строку в таблицу *Inventory* со следующими данными:

ItemID: *(автоматически сгенерированное значение)*

ItemName: Плоскогубцы

Price: 14.99

Строка 23 фиксирует изменения в базе данных, строка 26 закрывает соединение с базой данных. На рис. 14.5 показано содержимое таблицы *Inventory* после выполнения программы. Обратите внимание, что столбец *ItemID* содержит автоматически сгенерированные значения 1, 2 и 3.

ItemID	ItemName	Price
1	Отвертка	4.99
2	Молоток	12.99
3	Плоскогубцы	14.99

РИС. 14.5. Содержимое таблицы *Inventory*

Вставка нескольких строк с помощью одной инструкции *INSERT*

В программе 14.4 используются три отдельные инструкции *INSERT* для вставки трех строк в таблицу *Inventory*. В качестве альтернативы вы можете вставить несколько строк в таблицу с помощью всего одной инструкции *INSERT*. Вот пример инструкции, которая вставляет три строки в таблицу *Inventory*:

```
cur.execute('''INSERT INTO Inventory (ItemName, Price)
    VALUES ("Отвертка", 4.99),
            ("Молоток", 12.99),
            ("Плоскогубцы", 14.99)'''')
```

Как видно из примера, ключевое слово *VALUES* достаточно записать один раз с последующими несколькими наборами значений, разделенными запятыми.

Вставка нулевых данных

Иногда может не оказаться всех данных для строки, которую вы вставляете в таблицу. Например, предположим, что вы хотите добавить новую позицию в таблицу *Inventory*, но у вас еще нет цены этого товара. В такой ситуации вы можете использовать значение *NULL* для столбца *Price*, понимая, что позже вы обновите строку с правильной ценой. Вот пример:

```
cur.execute('''INSERT INTO Inventory (ItemName, Price)
    VALUES ("Электродрель", NULL)'''')
```

Вы также можете неявно присвоить NULL столбцу, просто оставив столбец вне инструкции `INSERT`. Вот пример:

```
cur.execute(''INSERT INTO Inventory (ItemName)
VALUES ("Электродрель")'')
```

Значение `NULL` должно использоваться только в качестве местозаполнителя для неизвестных данных. Если вы попытаетесь использовать `NULL` в вычислительной операции, это, скорее всего, приведет к исключению или неправильному результату. По этой причине при назначении `NULL` столбцу следует соблюдать осторожность. Для того чтобы столбцу никогда не присваивалось значение `NULL`, при создании таблицы нужно использовать ограничение `NOT NULL`. Вот пример:

```
CREATE TABLE Inventory (ItemID INTEGER PRIMARY KEY NOT NULL,
ItemName TEXT NOT NULL,
Price REAL NOT NULL)
```

В этом примере мы применили ограничение `NOT NULL` ко всем трем столбцам таблицы. Первый столбец, `ItemID`, является целочисленным первичным ключом `INTEGER PRIMARY KEY`. Если мы присвоим `NULL` целочисленному первичному ключу, то СУБД создаст значение для столбца автоматически. Если мы попытаемся назначить `NULL` столбцам `ItemName` или `Price`, СУБД выдаст исключение.

Вставка значений переменных

Вам часто нужно будет вставлять значения переменных в столбцы таблицы базы данных. Например, вам может потребоваться написать программу, которая получает значения от пользователя, а затем вставляет эти значения в строку. Для выполнения этой работы SQLite позволяет написать инструкцию SQL, в которой вопросительные знаки появляются в качестве местозаполнителей для значений. Например, взгляните на следующий строковый литерал, содержащий инструкцию `INSERT`:

```
'''INSERT INTO Inventory (ItemName, Price) VALUES (?, ?)'''
```

Обратите внимание, что выражение `VALUES` вместо фактических значений содержит вопросительные знаки. Когда мы вызываем метод `execute()`, мы передаем этот строковый литерал в качестве первого аргумента, а кортеж переменных — в качестве второго аргумента. Значения переменных будут вставлены вместо вопросительных знаков перед передачей инструкции SQL в СУБД. Этот тип инструкции SQL называется *параметризованным запросом*. Вот общий формат передачи параметризованного запроса в метод `execute()`:

```
cur.execute(СтрокаСSQL_с_Местозаполнителями, (Переменная1, Переменная2, ...))
```

Когда эта инструкция исполняется, значение `Переменной1` будет вставлено вместо первого вопросительного знака, значение `Переменной2` — вместо второго вопросительного знака и т. д. В приведенном ниже фрагменте кода показано применение этого технического приема для добавления строки в таблицу `Inventory` в базе данных `inventory.db`. Предположим, что `cur` — это объект `Cursor`.

```
1 item_name = "Гаечный ключ"
2 price = 16.99
3 cur.execute(''INSERT INTO Inventory (ItemName, Price)
VALUES (?, ?)'',
4 (item_name, price))
```

В этом фрагменте кода строка 1 присваивает "Гаечный ключ" переменной `item_name`, а строка 2 присваивает 16.99 переменной `price`. Обратите внимание на вопросительные знаки, которые появляются в выражении `VALUES` в строке 4. При исполнении метода `execute()` значение переменной `item_name` займет место первого вопросительного знака, а значение переменной `price` — место второго вопросительного знака. В результате в таблицу будет вставлена следующая строка:

ItemID: (автоматически сгенерированное значение)

ItemName: Гаечный ключ

Price: 16.99

В программе 14.5 приведен полный исходный код, который получает от пользователя названия позиций и цены, а затем сохраняет входные данные в таблице `Inventory`.

Программа 14.5 (insert_variables.py)

```

1 import sqlite3
2
3 def main():
4     # Переменная управления циклом.
5     again = 'д'
6
7     # Подсоединиться к базе данных.
8     conn = sqlite3.connect('inventory.db')
9
10    # Получить курсор.
11    cur = conn.cursor()
12
13    while again == 'д':
14        # Получить название и цену позиции.
15        item_name = input('Название: ')
16        price = float(input('Цена: '))
17
18        # Добавить позицию в таблицу Inventory.
19        cur.execute(''': INSERT INTO Inventory (ItemName, Price)
20                    VALUES (?, ?)'',
21                    (item_name, price))
22
23        # Добавить еще?
24        again = input('Добавить еще одну позицию? (д/н): ')
25
26    # Зафиксировать изменения.
27    conn.commit()
28
29    # Закрыть соединение.
30    conn.close()
31

```

```

32 # Вызвать главную функцию.
33 if __name__ == '__main__':
34     main()

```

Вывод программы (ввод выделен жирным шрифтом)

```

Название: Пила Enter
Цена: 24.99 Enter
Добавить еще одну позицию? (д/н): д Enter
Название: Дрель Enter
Цена: 89.99 Enter
Добавить еще одну позицию? (д/н): д Enter
Название: Рулетка Enter
Цена: 8.99 Enter
Добавить еще одну позицию? (д/н): и Enter

```

Значение Python `None` и значение SQLite `NULL` эквивалентны. Если переменная имеет значение `None` и это значение переменной вы вставляете в столбец, столбцу будет присвоено значение `NULL`. По этой причине следует соблюдать осторожность, чтобы непреднамеренно не присвоить `NULL` столбцу при вставке значения переменной в инструкции `INSERT`. Вот пример:

```

1 item_name = "Гаечный ключ"
2 price = None
3 cur.execute(''INSERT INTO Inventory (ItemName, Price)
4             VALUES (?, ?)'',
5             (item_name, price))

```

Этот код вставит в таблицу `Inventory` следующую строку:

```

ItemID: (автоматически сгенерированное значение)
ItemName: Гаечный ключ
Price: NULL

```

Следите за атаками SQL-инъекций

По соображениям безопасности **никогда** не используйте конкатенацию строк для вставки введенных пользователем данных непосредственно в инструкцию SQL. Вот пример:

```

# ВНИМАНИЕ! НЕ пишите такой код!
name = input('Введите название позиции: ')
price = float(input('Введите цену: '))
cur.execute('INSERT INTO Inventory (ItemName, Price) ' +
           'VALUES ("' + name + '", ' + str(price) + ')')

```

Кроме того, вы никогда не должны вставлять вводимые пользователем данные в инструкцию SQL с использованием местозаполнителей `f`-строки, как показано ниже:

```

# ВНИМАНИЕ! НЕ пишите такой код!
name = input('Введите название позиции: ')
price = float(input('Введите цену: '))

```

```
cur.execute(f'INSERT INTO Inventory (ItemName, Price) ' +
            f'VALUES ("{{name}}", {{price}}")')
```

Эти приемы работы делают вашу программу уязвимой для атаки, которая называется *SQL-инъекцией*. Инъекция SQL может произойти, когда приложение запрашивает у пользователя входные данные, и вместо ввода ожидаемых данных пользователь вводит фрагмент искусно разработанного кода SQL. Когда этот фрагмент SQL-кода вставляется в инструкцию SQL вашей программы, он изменяет характер исполнения инструкции и потенциально выполняет вредоносное действие в вашей базе данных. Вместо конкатенации строк или местозаполнителей f-строк для сборки инструкций SQL следует использовать параметризованные запросы, как было показано ранее в этой главе. Большинство СУБД выполняют параметризованные запросы таким образом, чтобы исключить возможность инъекции SQL.

Существуют и другие технические приемы предотвращения инъекции SQL. Например, перед вставкой входных данных пользователя в инструкцию SQL программа может проверить эти данные, чтобы убедиться, что они не содержат операторов или других символов, которые могут указывать на то, что пользователь ввел код SQL. Если ввод выглядит подозрительно, то он отклоняется.

Эта глава предназначена для введения в программирование баз данных на языке Python, поэтому мы не будем подробно останавливаться на предотвращении атаки с использованием инъекций SQL. Однако при написании кода в рабочей среде вы обязаны обеспечить безопасность своих программ. Инъекция SQL — это один из наиболее распространенных способов взлома веб-сайтов хакерами, поэтому следует изучить эту тему подробнее позже.



Контрольная точка

14.23. Напишите инструкцию SQL для вставки следующих данных в таблицу `Inventory` базы данных `inventory.db`:

```
ItemID: 10
ItemName: "Циркулярная пила"
Price: 199.99
```

14.24. Напишите инструкцию SQL для вставки следующих данных в таблицу `Inventory` базы данных `inventory.db`:

```
ItemID: автоматически сгенерированный
ItemName: "Зубило"
Price: 8.99
```

14.25. Предположим, что `cur` является объектом `Cursor`, переменная `name_input` ссылается на строковое значение, а переменная `price_input` ссылается на значение с плавающей точкой. Напишите инструкцию, в которой используется параметризованный запрос SQL для добавления строки в таблицу `Inventory` базы данных `inventory.db`. В этом запросе значение переменной `name_input` вставляется в столбец `ItemName`, а значение переменной `price_input` — в столбец `Price`.

14.6 Запрос данных с помощью инструкции SQL *SELECT*

Ключевые положения

Инструкция *SELECT* используется в SQL для извлечения данных из базы данных.

Образец базы данных

В этом разделе мы используем инструкцию *SELECT* для извлечения строк из таблицы. В наших примерах мы будем работать с образцом базы данных, которая входит в исходный код этой книги. База данных содержит данные от вымышленной компании, которая продает шоколадные изделия для гурманов. База данных называется *chocolate.db* и содержит таблицу *Products* со следующими столбцами:

- ◆ *ProductID* (ID изделия) — INTEGER PRIMARY KEY NOT NULL;
- ◆ *Description* (описание) — TEXT;
- ◆ *UnitCost* (стоимость единицы) — REAL;
- ◆ *RetailPrice* (розничная цена) — REAL;
- ◆ *UnitsOnHand* (единиц в наличии) — INTEGER.

Таблица *Products* содержит строки данных, приведенные в табл. 14.2.

Таблица 14.2. Таблица *Products* в базе данных *chocolate.db*

<i>ProductID</i>	<i>Description</i>	<i>UnitCost</i>	<i>RetailPrice</i>	<i>UnitsOnHand</i>
1	Плитка темного шоколада	2.99	5.99	197
2	Плитка средняя темного шоколада	2.99	5.99	406
3	Плитка молочного шоколада	2.99	5.99	266
4	Шоколадные трюфели	5.99	11.99	398
5	Плитка шоколада с карамелью	3.99	6.99	272
6	Плитка шоколада с малиной	3.99	6.99	363
7	Плитка шоколада с кешью	4.99	9.99	325
8	Смесь горячего шоколада	5.99	12.99	222
9	Стружка из полусладкого шоколада	1.99	3.99	163
10	Стружка из белого шоколада	1.99	3.99	293

Инструкция *SELECT*

Как следует из названия, инструкция *SELECT* позволяет выбирать те или иные строки в таблице. Мы начнем с очень простой формы этой инструкции:

SELECT Столбцы FROM Таблица



Видеозапись "Инструкция *SELECT*" (The *SELECT* Statement)

Здесь *Столбцы* — это одно или несколько имен столбцов; *Таблица* — имя таблицы. Когда эта инструкция исполняется, она извлекает указанные столбцы из каждой строки указанной таблицы. Вот пример инструкции SELECT, которую мы могли бы исполнить для базы данных chocolate.db:

```
SELECT Description FROM Products
```

Эта инструкция извлекает столбец Description для каждой строки из таблицы Products.

Вот еще один пример:

```
SELECT Description, RetailPrice FROM Products
```

Эта инструкция извлекает столбцы Description и RetailPrice для каждой строки из таблицы Products.

В Python использование инструкции SELECT с СУБД SQLite представляет собой двухшаговый процесс:

1. Выполнить инструкцию SELECT. Сначала вы передаете инструкцию SELECT в виде строкового литерала в метод execute() объекта Cursor. СУБД извлекает результаты инструкции SELECT, но не возвращает эти результаты в вашу программу.
2. Получить результаты. После исполнения инструкции SELECT нужно вызвать метод fetchall() или метод fetchone() для получения результатов (fetchall и fetchone являются методами объекта Cursor).

Метод fetchall() объекта Cursor возвращает результаты инструкции SELECT в виде списка кортежей. В целях ознакомления с тем, как это работает, посмотрите на приведенный ниже интерактивный сеанс:

```
1 >>> import sqlite3
2 >>> conn = sqlite3.connect('chocolate.db')
3 >>> cur = conn.cursor()
4 >>> cur.execute('SELECT Description, RetailPrice FROM Products')
5 <sqlite3.Cursor object at 0x0000024E5E0FFE30>
6 >>> cur.fetchall()
7 [ ('Плитка темного шоколада', 5.99),
  ('Плитка средняя темного шоколада', 5.99),
  ('Плитка молочного шоколада', 5.99),
  ('Шоколадные трюфели', 11.99),
  ('Плитка шоколада с карамелью', 6.99),
  ('Плитка шоколада с малиной', 6.99),
  ('Плитка шоколада с кешью', 9.99),
  ('Смесь горячего шоколада', 12.99),
  ('Стружка из полусладкого шоколада', 3.99),
  ('Стружка из белого шоколада', 3.99)]
```

Инструкция SELECT в строке 4 извлекает столбцы Description и RetailPrice из каждой строки таблицы Products. Обратите внимание, что при вызове метода fetchall() в строке 6 указанный метод возвращает все результаты инструкции SELECT в виде списка кортежей. Каждый элемент списка является кортежем, и каждый кортеж имеет два элемента: описание и розничную цену. Если бы мы извлекали эти данные в программе, то мы, вероятно, захотели бы прокрутить список в цикле и распечатать каждый элемент кортежа в более удобном для чтения формате. В программе 14.6 показан пример.

Программа 14.6 (get_descriptions_prices.py)

```

1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Получить описания и розничные цены.
11    cur.execute('SELECT Description, RetailPrice FROM Products')
12
13    # Извлечь результаты инструкции SELECT.
14    results = cur.fetchall()
15
16    # Перебрать строки и показать результаты.
17    for row in results:
18        print(f'{row[0]:35} {row[1]:5}')
19
20    # Закрыть соединение с базой данных.
21    conn.close()
22
23 # Вызвать главную функцию.
24 if __name__ == '__main__':
25     main()

```

Вывод программы

Плитка темного шоколада	5.99
Плитка средняя темного шоколада	5.99
Плитка молочного шоколада	5.99
Шоколадные трюфели	11.99
Плитка шоколада с карамелью	6.99
Плитка шоколада с малиной	6.99
Плитка шоколада с кешью	9.99
Смесь горячего шоколада	12.99
Стружка из полусладкого шоколада	3.99
Стружка из белого шоколада	3.99

В этой программе строка 11 выполняет инструкцию SELECT, а строка 14 вызывает метод `fetchall()` для получения результатов инструкции SELECT. Результаты возвращаются в виде списка и присваиваются переменной `results`. Каждый элемент списка представляет собой кортеж, содержащий столбцы `Description` и `RetailPrice` из строки таблицы.

Цикл `for` в строках 17–18 программы перебирает список `results`. По мере выполнения цикла переменная строки будет ссылаться на кортеж из списка. Функция `print` в строке 18 использует `f`-строку для вывода на экран двух элементов кортежа, `row[0]` и `row[1]`. Элемент

в `row[0]` выводится в поле шириной 35 пробелов, а элемент в `row[1]` выводится в поле шириной 5 пробелов.

Вы видели, как метод `fetchall()` возвращает список, содержащий все строки, получаемые в результате исполнения инструкции `SELECT`. В качестве альтернативы можно применять метод `fetchone()`, который возвращает только одну строку в качестве кортежа при каждом его вызове. Метод `fetchone()` можно использовать для перебора результатов инструкции `SELECT` без извлечения всего списка. После исполнения инструкции `SELECT` первый вызов метода `fetchone()` возвращает первую строку в результатах, второй вызов метода `fetchone()` возвращает вторую строку в результатах и т. д. Если строк больше не осталось, метод `fetchone()` возвращает значение `None`. Взгляните на следующий интерактивный сеанс. Предположим, что `cur` является объектом `Cursor` для базы данных `chocolate.db`:

```
>>> cur.execute('SELECT Description, RetailPrice FROM Products') [Enter]
<sqlite3.Cursor object at 0x0084FBA0>
>>> cur.fetchone() [Enter]
("Плитка темного шоколада", 5.99) [Enter]
```

В приведенном выше сеансе инструкция `SELECT` извлекает столбцы `Description` и `RetailPrice` всех строк таблицы. Однако метод `fetchone()` вернул только первую строку результатов. Если мы снова вызовем метод `fetchone()`, то он вернет вторую строку, и т. д. Когда больше строк не останется, метод `fetchone()` вернет значение `None`. Программа 14.7 это демонстрирует.

Программа 14.7 (fetchone_demo.py)

```
1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Выбрать все столбцы из каждой строки таблицы Products.
11    cur.execute('SELECT Description, RetailPrice FROM Products')
12
13    # Извлечь первую строку результатов.
14    row = cur.fetchone()
15
16    while row != None:
17        # Показать строку.
18        print(f'{row[0]:35} {row[1]:5}')
19
20        # Извлечь следующую строку.
21        row = cur.fetchone()
22
```

```

23     # Закрыть соединение с базой данных.
24     conn.close()
25
26 # Вызвать главную функцию.
27 if __name__ == '__main__':
28     main()

```

Вывод программы

Плитка темного шоколада	5.99
Плитка средняя темного шоколада	5.99
Плитка молочного шоколада	5.99
Шоколадные трюфели	11.99
Плитка шоколада с карамелью	6.99
Плитка шоколада с малиной	6.99
Плитка шоколада с кешью	9.99
Смесь горячего шоколада	12.99
Стружка из полусладкого шоколада	3.99
Стружка из белого шоколада	3.99

Выбор всех столбцов в таблице

Если вы хотите использовать инструкцию SELECT для извлечения каждого столбца таблицы, можете использовать символ * вместо перечисления имен столбцов. Вот пример:

```
SELECT * FROM Products
```

Эта инструкция извлекает каждый столбец для каждой строки из таблицы Products. Предположим, что мы подсоединились к базе данных chocolate.db, а cur — это объект Cursor. Следующий интерактивный сеанс демонстрирует пример:

```

>>> cur.execute('SELECT * FROM Products') [Enter]
<sqlite3.Cursor object at 0x03AB4520>
>>> cur.fetchall()
[(1, 'Плитка темного шоколада', 2.99, 5.99, 197),
 (2, 'Плитка средняя темного шоколада', 2.99, 5.99, 406),
 (3, 'Плитка молочного шоколада', 2.99, 5.99, 266),
 (4, 'Шоколадные трюфели', 5.99, 11.99, 398),
 (5, 'Плитка шоколада с карамелью', 3.99, 6.99, 272),
 (6, 'Плитка шоколада с малиной', 3.99, 6.99, 363),
 (7, 'Плитка шоколада с кешью', 4.99, 9.99, 325),
 (8, 'Смесь горячего шоколада', 5.99, 12.99, 222),
 (9, 'Стружка из полусладкого шоколада', 1.99, 3.99, 163),
 (10, 'Стружка из белого шоколада', 1.99, 3.99, 293)]

```

Программа 14.8 обеспечивает вывод результатов инструкции SELECT в более удобном для чтения формате.

Программа 14.8 (get_all_columns.py)

```

1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Выбрать все столбцы из каждой строки таблицы Products.
11    cur.execute('SELECT * FROM Products')
12
13    # Излечь результаты инструкции SELECT.
14    results = cur.fetchall()
15
16    # Показать результаты.
17    for row in results:
18        print(f'{row[0]:2} {row[1]:35} {row[2]:5} {row[3]:6} {row[4]:5}')
19
20    # Закрыть соединение с базой данных.
21    conn.close()
22
23 # Вызвать главную функцию.
24 if __name__ == '__main__':
25     main()

```

Вывод программы

1 Плитка темного шоколада	2.99	5.99	197
2 Плитка средняя темного шоколада	2.99	5.99	406
3 Плитка молочного шоколада	2.99	5.99	266
4 Шоколадные трюфели	5.99	11.99	398
5 Плитка шоколада с карамелью	3.99	6.99	272
6 Плитка шоколада с малиной	3.99	6.99	363
7 Плитка шоколада с кешью	4.99	9.99	325
8 Смесь горячего шоколада	5.99	12.99	222
9 Стружка из полусладкого шоколада	1.99	3.99	163
10 Стружка из белого шоколада	1.99	3.99	293

В этой программе строка 11 исполняет инструкцию SELECT, а строка 14 вызывает метод `fetchall()` для получения результатов инструкции SELECT. Результаты возвращаются в виде списка и присваиваются переменной `results`. В таблице пять столбцов, поэтому каждый элемент списка представляет собой кортеж, состоящий из пяти элементов.

Цикл `for` в строках 17–18 перебирает список `results`. По мере повторения цикла переменная `row` будет ссылаться на кортеж из списка. Функция `print` в строке 18 использует `f`-строку для вывода на экран элементов кортежа в столбцах.

Указание критериев поиска с помощью выражения *WHERE*

Иногда может потребоваться получить все строки из таблицы. Например, если вам нужен список всех товарных позиций из таблицы `Products`, инструкция `SELECT * FROM Products` предоставит его вам. Однако обычно требуется сузить список до нескольких выбранных строк таблицы. Вот тут-то и вступает в игру выражение `WHERE`. Его можно использовать с инструкцией `SELECT` для указания критериев поиска. Если использовать выражение `WHERE`, то в результирующем наборе будут возвращены только те строки, которые удовлетворяют критерию поиска. Общий формат инструкции `SELECT` с выражением `WHERE` таков:

```
SELECT Столбцы FROM Таблица WHERE Критерий
```

Здесь `Критерий` — это условное выражение. Вот пример инструкции `SELECT`, в которой используется выражение `WHERE`:

```
SELECT * FROM Products WHERE RetailPrice > 10.00
```

В первой части инструкции `SELECT * FROM Products` указывается, что мы хотим получить все столбцы. Выражение `WHERE` указывает на то, что нам нужны только те строки, в которых содержимое столбца `RetailPrice` превышает 10.00. Приведенный ниже интерактивный сеанс это демонстрирует. Предположим, что мы подсоединились к базе данных `chocolate.db`, и `cur` — это объект `Cursor`.

```
>>> cur.execute('SELECT * FROM Products WHERE RetailPrice > 10.00') [Enter]
<sqlite3.Cursor object at 0x03AB4520>
>>> cur.fetchall()
[(4, 'Шоколадные трюфели', 5.99, 11.99, 398),
 (8, 'Смесь горячего шоколада', 5.99, 12.99, 222)]
```

В приведенном ниже примере показано, каким образом можно получить только столбец `Description` для всех изделий, розничная цена которых превышает 10.00:

```
>>> cur.execute('SELECT Description FROM Products WHERE RetailPrice > 10.00') [Enter]
<объект sqlite3.Cursor в 0x03AB4520>
>>> cur.fetchall()
[('Шоколадные трюфели',), ('Смесь горячего шоколада',)]
```

SQLite поддерживает реляционные операторы, перечисленные в табл. 14.3. Например, следующая ниже инструкция выбирает все строки, в которых данные в столбце `UnitsOnHand` меньше 100:

```
SELECT * FROM Products WHERE UnitsOnHand < 100
```

Следующая инструкция выбирает все строки, в которых значение в столбце `UnitCost` равно 2.99:

```
SELECT * FROM Products WHERE UnitCost == 2.99
```

Приведенная ниже инструкция выбирает строку, в которой столбец описания равен 'Смесь горячего шоколада':

```
SELECT * FROM Products WHERE Description == 'Смесь горячего шоколада'
```

Обратите внимание, что в табл. 14.3 SQLite предоставляет два оператора "равно" и два оператора "не равно". Рекомендуется использовать `==` для равных сравнений равенства и `!=` для сравнения неравенства, поскольку это те же операторы, которые используются в Python.

Таблица 14.3. Реляционные операторы языка SQL

Оператор	Описание
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
=	Равно
!=	Не равно
<>	Не равно

Программа 14.9 позволяет пользователю ввести минимальную цену, а затем в таблице Products выполняет поиск строк, в которых столбец RetailPrice больше или равен указанной цене.

Программа 14.9 (product_min_price.py)

```

1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Получить от пользователя минимальную цену.
11    min_price = float(input('Введите минимальную цену: '))
12
13    # Отправить инструкцию SELECT в СУБД.
14    cur.execute(''':SELECT Description, RetailPrice FROM Products
15                WHERE RetailPrice >= ?''',
16                (min_price,))
17
18    # Извлечь результаты инструкции SELECT.
19    results = cur.fetchall()
20
21    if len(results) > 0:
22        # Показать результаты.
23        print('Вот результаты:')
24        print()
25        print('Описание'.rjust(20), 'Цена')
26        print('-----')

```

```

27     for row in results:
28         print(f'{row[0]:35} {row[1]:>5}')
29     else:
30         print('Ни одно изделие не найдено.')
31
32     # Закрыть соединение с базой данных.
33     conn.close()
34
35 # Вызвать главную функцию.
36 if __name__ == '__main__':
37     main()

```

Вывод программы (ввод выделен жирным шрифтом)

Введите минимальную цену: **5.99**

Вот результаты:

Описание	Цена
Плитка темного шоколада	5.99
Плитка средняя темного шоколада	5.99
Плитка молочного шоколада	5.99
Шоколадные трюфели	11.99
Плитка шоколада с карамелью	6.99
Плитка шоколада с малиной	6.99
Плитка шоколада с кешью	9.99
Смесь горячего шоколада	12.99

Логические операторы языка SQL: *AND*, *OR* и *NOT*

Логические операторы *AND* и *OR* можно использовать для указания нескольких критериев поиска в выражении *WHERE*. Например, посмотрите на следующую инструкцию:

```

SELECT * FROM Products
WHERE UnitCost > 3.00 AND UnitsOnHand < 100

```

Оператор *AND* требует истинности обоих критериев поиска, чтобы строка таблицы была квалифицирована как совпадающая. Единственные строки, которые будут возвращены из этой инструкции, — те, в которых столбец *UnitCost* больше 3.00, а столбец *UnitsOnHand* меньше 100.

Вот пример, в котором используется оператор *OR*:

```

SELECT * FROM Products
WHERE RetailPrice > 10.00 OR UnitsOnHand < 50

```

Оператор *OR* требует истинности любого критерия поиска, чтобы строка таблицы была квалифицирована как совпадающая. Эта инструкция выполняет поиск строк, в которых столбец *RetailPrice* больше 10.00 или столбец *UnitsOnHand* меньше 50.

Оператор *NOT* изменяет истинность своего операнда. Если оператор применяется к выражению, которое является истинным, то он возвращает ложь. Если оператор применяется к вы-

ражению, которое является ложным, он возвращает истину. Вот пример, в котором используется оператор NOT:

```
SELECT * FROM Products
WHERE NOT RetailPrice > 5.00
```

Эта инструкция выполняет поиск строк, в которых столбец `RetailPrice` не превышает 5.00.

Вот еще один пример:

```
SELECT * FROM Products
WHERE NOT (RetailPrice > 5.00 AND RetailPrice < 10.00)
```

Эта инструкция выполняет поиск строк, в которых столбец `RetailPrice` не превышает 5.00 и не менее 10.00.

Сравнение строковых значений в инструкции `SELECT`

Сравнение строковых значений в SQL чувствительно к регистру. Если вы выполните следующую ниже инструкцию для базы данных `chocolate.db`, то не получите никаких результатов:

```
SELECT * FROM Products
WHERE Description == "плитка молочного шоколада"
```

Однако вы можете использовать функцию `lower()` для конвертирования строкового значения в нижний регистр. Вот пример:

```
SELECT * FROM Products
WHERE lower(Description) == "плитка молочного шоколада"
```

Эта инструкция вернет строку таблицы, в которой столбец `Description` имеет значение "Плитка молочного шоколада". SQLite также предоставляет функцию `upper()`, которая конвертирует свой аргумент в верхний регистр.

Использование оператора `LIKE`

Иногда поиск точного строкового значения не дает желаемых результатов. Например, предположим, что нам нужен список всех плиток шоколада из таблицы `Products`. Следующая инструкция работать не будет. Догадываетесь, почему?

```
SELECT * FROM Products WHERE Description == "Плитка"
```

Эта инструкция будет искать строки таблицы, в которых столбец `Description` равен строковому значению "Плитка". К сожалению, он ничего не найдет. Если вы снова взглянете на табл. 14.1, то увидите, что ни в одной из строк таблицы `Products` нет столбца `Description`, равного "Плитка". Однако есть несколько строк, в которых слово "Плитка" действительно появляется в столбце `Description`. Например, в одной строке вы найдете "Плитка темного шоколада", в другой строке — "Плитка молочного шоколада". В еще одной строке вы обнаружите "Плитка шоколада с карамелью". В дополнение к слову "Плитка" каждая из этих строк содержит другие символы.

Для того чтобы найти все плитки шоколада, нам нужно отыскать строки, в которых "Плитка" появляется в качестве подстроки в столбце `Description`. Вы можете выполнить такой поиск с помощью оператора `LIKE`. Вот пример того, как его использовать:

```
SELECT * FROM Products WHERE Description LIKE "%Плитка%"
```

За оператором LIKE следует строковое значение, содержащее *символьный шаблон*. В этом примере символьным шаблоном является "%Плитка%". Символ % используется в качестве *подстановочного знака*. Он представляет любую последовательность из нуля или более символов. Шаблон "%Плитка%" указывает строковое значение, которое содержит "Плитка" с любыми символами перед ним или после него. Следующий интерактивный сеанс это демонстрирует:

```
>>> cur.execute(''':SELECT * FROM Products
  WHERE Description LIKE "%Bar%"'') [Enter]
<sqlite3.Cursor object at 0x035D4520>
>>> cur.fetchall()
[(1, 'Плитка темного шоколада', 2.99, 5.99, 197),
 (2, 'Плитка средняя темного шоколада', 2.99, 5.99, 406),
 (3, 'Плитка молочного шоколада', 2.99, 5.99, 266),
 (5, 'Плитка шоколада с карамелью', 3.99, 6.99, 272),
 (6, 'Плитка шоколада с малиной', 3.99, 6.99, 363),
 (7, 'Плитка шоколада с кешью', 4.99, 9.99, 325)]
```

Следующий интерактивный сеанс показывает еще один пример. В этом сеансе мы ищем все строки, в которых столбец Description содержит строковое значение "Стружка". Инструкция SELECT возвращает только столбец Description соответствующих строк таблицы:

```
>>> cur.execute(''':SELECT Description FROM Products
  WHERE Description LIKE "%Стружка%"'') [Enter]
<sqlite3.Cursor object at 0x035D4520>
>>> cur.fetchall()
[('Стружка из полусладкого шоколада',), ('Стружка из белого шоколада',)]
```

Вы также можете использовать подстановочный знак % для поиска строковых значений, которые начинаются с указанной подстроки или оканчиваются указанной подстрокой. Например, инструкция SELECT в следующем ниже интерактивном сеансе выполняет поиск строковых значений, в которых столбец Description имеет подстроку "шоколада":

```
>>> cur.execute(''':SELECT Description FROM Products
  WHERE Description LIKE "%шоколада%"'') [Enter]
<sqlite3.Cursor object at 0x035D4520>
>>> cur.fetchall()
[('Плитка темного шоколада',),
 ('Плитка средняя темного шоколада',),
 ('Плитка молочного шоколада',),
 ('Смесь горячего шоколада',),
 ('Стружка из полусладкого шоколада',),
 ('Стружка из белого шоколада',)]
```

Инструкция SELECT в следующем ниже интерактивном сеансе выполняет поиск строк, в которых столбец Description начинается с подстроки "Смесь":

```
>>> cur.execute(''':SELECT Description FROM Products
  WHERE Description LIKE "%Смесь%"'') [Enter]
<sqlite3.Cursor object at 0x035D4520>
>>> cur.fetchall()
[('Смесь горячего шоколада',)]
```

Вы можете объединить оператор NOT и оператор LIKE для поиска строк, которые не соответствуют некоторому шаблону. Например, предположим, что вам нужны описания всех изделий, которые не содержат слова "Плитка". Следующий ниже интерактивный сеанс это демонстрирует:

```
>>> cur.execute('''SELECT Description FROM Products
   WHERE Description NOT LIKE "%Плитка%"''') 
<sqlite3.Cursor object at 0x035D4520>
>>> cur.fetchall()
[("Шоколадные трюфели",),
 ("Смесь горячего шоколада",),
 ("Стружка из полусладкого шоколада",),
 ("Стружка из белого шоколада",)]
```

Сортировка результатов запроса *SELECT*

Если вы хотите отсортировать результаты запроса *SELECT*, можете использовать выражение *ORDER BY*. Вот пример:

```
SELECT * FROM Products ORDER BY RetailPrice
```

Эта инструкция создаст список всех строк таблицы *Products*, упорядоченных по столбцу розничных цен. Список будет отсортирован в порядке возрастания значений в столбце *RetailPrice*, т. е. первыми появятся изделия с самой низкой ценой. Программа 14.10 это демонстрирует.

Программа 14.10 (sorted_by_retailprice.py)

```
1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Выбрать все столбцы из каждой строки таблицы Products.
11    cur.execute('''SELECT Description, RetailPrice FROM Products
12                           ORDER BY RetailPrice''')
13
14    # Извлечь результаты инструкции SELECT.
15    results = cur.fetchall()
16
17    # Показать результаты.
18    for row in results:
19        print(f'{row[0]:35} {row[1]:5}')
20
21    # Закрыть соединение с базой данных.
22    conn.close()
```

```

23
24 # Вызывать главную функцию.
25 if __name__ == '__main__':
26     main()

```

Вывод программы

Стружка из полусладкого шоколада	3.99
Стружка из белого шоколада	3.99
Плитка темного шоколада	5.99
Плитка средняя темного шоколада	5.99
Плитка молочного шоколада	5.99
Плитка шоколада с карамелью	6.99
Плитка шоколада с малиной	6.99
Плитка шоколада с кешью	9.99
Шоколадные трюфели	11.99
Смесь горячего шоколада	12.99

Вот инструкция SELECT, в которой используется выражение WHERE и выражение ORDER BY:

```

SELECT * FROM Products
WHERE RetailPrice > 9.00
ORDER BY RetailPrice

```

Эта инструкция создаст список всех строк таблицы Products, где столбец RetailPrice содержит значение, превышающее 9.00, в порядке возрастания по цене.

Если мы хотим, чтобы список был отсортирован в порядке убывания (от наибольшего значения до наименьшего), можем использовать оператор DESC, как показано ниже:

```

SELECT * FROM Products
WHERE RetailPrice > 9.00
ORDER BY RetailPrice DESC

```

Агрегатные функции

В языке SQL *агрегатная функция* выполняет вычисление на наборе значений из таблицы базы данных и возвращает одно значение. Например, функция AVG вычисляет среднее значение столбца, содержащего числовые данные. Вот пример инструкции SELECT с использованием функции AVG:

```
SELECT AVG(RetailPrice) FROM Products
```

Эта инструкция создает одно значение: среднее всех значений в столбце RetailPrice. Поскольку мы не использовали выражение WHERE, при расчете она перебирает все строки таблицы Products. Вот пример, который вычисляет среднюю цену всех изделий, имеющих описание, содержащее слово "Плитка":

```
SELECT AVG(Price) FROM Products WHERE Description LIKE "%Плитка%"
```

Еще одной агрегатной функцией является SUM, которая вычисляет сумму по столбцу, содержащему числовые значения. Следующая ниже инструкция вычисляет сумму значений в столбце UnitsOnHand:

```
SELECT SUM(UnitsOnHand) FROM Products
```

Функции MIN и MAX определяют минимальное и максимальное значения, находящиеся в столбце, содержащем числовые данные. Следующая ниже инструкция сообщает нам минимальное значение в столбце RetailPrice:

```
SELECT MIN(RetailPrice) FROM Products
```

Следующая ниже инструкция сообщает нам максимальное значение в столбце RetailPrice:

```
SELECT MAX(RetailPrice) FROM Products
```

Функция COUNT может использоваться для определения числа строк в таблице:

```
SELECT COUNT(*) FROM Products
```

Символ * просто указывает на то, что вы хотите подсчитать все строки таблицы. Вот еще один пример, который сообщает нам о числе изделий, цена которых превышает 9.95:

```
SELECT COUNT(*) FROM Products WHERE RetailPrice > 9.95
```

В коде Python наиболее простым способом получения значения, возвращаемого из агрегатной функции, является метод `fetchone()` объекта `Cursor`, как показано в приведенном ниже интерактивном сеансе:

```
>>> cur.execute('SELECT SUM(UnitsOnHand) FROM Products')
<sqlite3.Cursor object at 0x0030FD20>
>>> cur.fetchone()
(2905,)
```

Когда запрос возвращает только одно значение, метод `fetchone()` возвращает кортеж с одним элементом, имеющим индекс 0. Если вы хотите работать со значением, а не с кортежем, вы должны прочитать значение из кортежа. Например, в следующем ниже интерактивном сеансе мы назначаем кортеж, возвращаемый методом `fetchone()`, переменной `results` (строка 4), а затем назначаем элемент 0 кортежа `results` переменной `total` (строка 5):

```
1 >>> cur.execute('SELECT SUM(UnitsOnHand) FROM Products')
2 <sqlite3.Cursor object at 0x0030FD20>
3
4 >>> results = cur.fetchone()
5 >>> total = results[0]
```

Показанный в интерактивном сеансе программный код элементарен, но мы можем упростить его еще больше. Поскольку метод `fetchone()` возвращает кортеж, мы можем применить оператор `[0]` непосредственно к выражению, вызывающему этот метод:

```
1 >>> cur.execute('SELECT SUM(UnitsOnHand) FROM Products')
2 <sqlite3.Cursor object at 0x0030FD20>
3
4 >>> total = cur.fetchone()[0]
```

В строке 4 оператор `[0]` применяется к кортежу, возвращаемому методом `fetchone()`. Таким образом, значение, которое находится в элементе 0 кортежа, присваивается переменной `total`.

Программа 14.11 демонстрирует пример того, как можно использовать функции MIN, MAX и AVG для поиска минимальной, максимальной и средней цены в таблице `Products`.

Программа 14.11 (products_math.py)

```
1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Получить самую низкую цену.
11    cur.execute('SELECT MIN(RetailPrice) FROM Products')
12    lowest = cur.fetchone()[0]
13
14    # Получить самую высокую цену.
15    cur.execute('SELECT MAX(RetailPrice) FROM Products')
16    highest = cur.fetchone()[0]
17
18    # Получить среднюю цену.
19    cur.execute('SELECT AVG(RetailPrice) FROM Products')
20    average = cur.fetchone()[0]
21
22    # Показать результаты.
23    print(f'Минимальная цена: ${lowest:.2f}')
24    print(f'Максимальная цена: ${highest:.2f}')
25    print(f'Средняя цена: ${average:.2f}')
26
27    # Закрыть соединение с базой данных.
28    conn.close()
29
30 # Вызвать главную функцию.
31 if __name__ == '__main__':
32     main()
```

Вывод программы

```
Минимальная цена: $3.99
Максимальная цена: $12.99
Средняя цена: $7.49
```

**Контрольная точка**

14.26. Посмотрите на следующую ниже инструкцию SQL.

```
SELECT Id FROM Account
```

- а) Как называется таблица, из которой эта инструкция извлекает данные?
- б) Как называется извлекаемый столбец?

14.27. Предположим, что в базе данных есть таблица `Inventory` со следующими столбцами:

Имя столбца	Тип
<code>ProductName</code>	<code>TEXT</code>
<code>QtyOnHand</code>	<code>INTEGER</code>
<code>Cost</code>	<code>REAL</code>

- Напишите инструкцию `SELECT`, которая вернет все столбцы из каждой строки таблицы `Inventory`.
- Напишите инструкцию `SELECT`, которая вернет столбец `ProductName` из каждой строки таблицы `Inventory`.
- Напишите инструкцию `SELECT`, которая вернет столбец `ProductName` и столбец `QtyOnHand` из каждой строки таблицы `Inventory`.
- Напишите инструкцию `SELECT`, которая вернет столбец `ProductName` только из строк, где стоимость меньше 17.00.
- Напишите инструкцию `SELECT`, которая вернет все столбцы из строк, где `ProductName` заканчивается на "ZZ".

14.28. Каково назначение оператора `LIKE`?

14.29. Каково назначение символа `%` в символьном шаблоне, используемом оператором `LIKE`?

14.30. Как можно отсортировать результаты инструкции `SELECT` по некоторому столбцу?

14.31. В чем разница между методом `fetchall()` класса `Cursor` и методом `fetchone()`?

14.7

Обновление и удаление существующих строк

Ключевые положения

Инструкция `UPDATE` используется в SQL для изменения значения существующей строки таблицы. Инструкция `DELETE` применяется для удаления строк из таблицы.

Обновление строк

В языке SQL инструкция `UPDATE` предназначена для изменения содержимого существующей строки таблицы.



Видеозапись "Обновление строк таблицы" (Updating Rows)

Используя нашу базу данных `chocolate.db` в качестве примера, предположим, что цена шоколадных трюфелей меняется. Для того чтобы обновить цену в базе данных, мы можем применить инструкцию `UPDATE` для изменения значения в столбце `RetailPrice` для этой конкретной строки в таблице `Products`. Вот общий формат инструкции `UPDATE`:

```
UPDATE Таблица
SET Столбец = Значение
WHERE Критерий
```

Здесь *Таблица* — это имя таблицы; *Столбец* — имя столбца; *Значение* — значение для хранения в столбце; *Критерий* — условное выражение. Вот инструкция UPDATE, в которой цена шоколадных трюфелей изменяется на 13.99:

```
UPDATE Products
SET RetailPrice = 13.99
WHERE Description == "Шоколадные трюфели"
```

И еще один пример:

```
UPDATE Products
SET Description = "Плитка полусладкого шоколада"
WHERE ProductID == 2
```

Эта инструкция отыскивает в таблице строку, в которой ProductID равен 2, и устанавливает столбец Description равным "Плитка полусладкого шоколада".

Имеется возможность обновлять более одной строки. Например, предположим, что мы хотим изменить цену каждой плитки шоколада на 8.99. Для этого нам нужна лишь инструкция UPDATE, которая отыскивает все строки, в которых столбец Description заканчивается на "шоколада", и меняет столбец RetailPrice этих строк на 8.99. Ниже приведена эта инструкция:

```
UPDATE Products
SET RetailPrice = 8.99
WHERE Description LIKE "%шоколада"
```



ВНИМАНИЕ!

Будьте осторожны и не пропустите блок WHERE с условным выражением при использовании инструкции UPDATE. Вы можете изменить содержимое всех строк таблицы! Например, посмотрите на следующую инструкцию:

```
UPDATE Products
SET RetailPrice = 4.95
```

Поскольку в этой инструкции нет выражения WHERE, он изменит значение в столбце RetailPrice для каждой строки в таблице Products на 4.95!

Всякий раз, когда вы выполняете инструкцию SQL, которая изменяет содержимое базы данных, вызывайте метод `commit()` объекта `Connection`, чтобы зафиксировать изменения. Программа 14.12 демонстрирует обновление строки таблицы `Products`. Пользователь вводит существующий ID изделия, а программа выводит на экран описание этого изделия и различную цену. Затем пользователь вводит для указанного изделия новую розничную цену, и программа обновляет строку таблицы новой ценой.

Программа 14.12 (product_price_updater.py)

```
1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
```

```

7  # Получить курсор.
8  cur = conn.cursor()
9
10 # Получить от пользователя ProductID.
11 pid = int(input('Введите ID изделия: '))
12
13 # Получить текущую цену для этого изделия.
14 cur.execute(''':SELECT Description, RetailPrice FROM Products
15             WHERE ProductID == ?''', (pid,))
16 results = cur.fetchone()
17
18 # Если ID изделия найден, то продолжить...
19 if results != None:
20     # Напечатать текущую цену.
21     print(f'Текущая цена для {results[0]}: '
22           f'${results[1]:.2f}')
23
24     # Получить новую цену.
25     new_price = float(input('Введите новую цену: '))
26
27     # Обновить цену в таблице Products.
28     cur.execute(''':UPDATE Products
29                 SET RetailPrice = ?
30                 WHERE ProductID == ?''',
31             (new_price, pid))
32
33     # Зафиксировать изменения.
34     conn.commit()
35     print('Цена была изменена.')
36 else:
37     # Сообщение об ошибке.
38     print(f'ID изделия {pid} не найден.')
39
40 # Закрыть соединение с базой данных.
41 conn.close()
42
43 # Вызвать главную функцию.
44 if __name__ == '__main__':
45     main()

```

Вывод программы (ввод выделен жирным шрифтом)

Введите ID изделия: 2 **Enter**

Текущая цена для Плитка средняя темного шоколада: \$5.00

Введите новую цену: **7.99** **Enter**

Цена была изменена.

Давайте рассмотрим эту программу подробнее. В главной функции строка 5 программы получает соединение с базой данных, а строка 8 — объект Cursor. Стока 11 запрашивает у пользователя ID изделия, который присваивается переменной pid.

Инструкция в строках 14 и 15 выполняет SQL-запрос, который отыскивает строку таблицы, в которой столбец ProductID равен переменной pid. Если такая строка найдена, то извлекаются столбцы Description и RetailPrice. Инструкция в строке 16 вызывает метод fetchone() для извлечения результатов запроса в виде кортежа и присваивает его переменной results. (Если SQL-запрос не найдет соответствующей строки, метод fetchone() вернет None.)

Если переменная results не равна None, то инструкция if в строке 19 исполнит фрагмент кода, который появляется в строках 20–35 программы. В строках 21–22 печатаются описание изделия и текущая розничная цена. Стока 25 запрашивает у пользователя новую цену, которая назначается переменной new_price. Инструкция в строках 28–31 выполняет команду SQL, которая обновляет столбец RetailPrice изделия значением переменной new_price. В строке 34 изменения фиксируются в базе данных, а в строке 35 выводится сообщение, подтверждающее, что цена была изменена.

Блок else в строке 36 выполняется, если пользователь ввел ID изделия, который не был найден в таблице Products. Стока 38 просто выводит на экран сообщение об этом.

Наконец, строка 41 закрывает соединение с базой данных.

Обновление нескольких столбцов

В целях обновления значений нескольких столбцов следует использовать приведенный ниже общий формат UPDATE языка SQL:

```
UPDATE Таблица
SET Столбец1 = Значение1,
    Столбец2 = Значение2,
    ...
WHERE Критерий
```

Приведем пример:

```
UPDATE Products
SET RetailPrice = 8.99,
    UnitsOnHand = 100
WHERE Description LIKE "%шоколада"
```

Эта инструкция изменяет значения в столбце RetailPrice на 8.99, а в столбце UnitsOnHand на 100 в каждой строке, где столбец Description содержит подстроку "%шоколада".

Определение числа обновленных строк

Объект Cursor имеет публичный атрибут данных rowcount, который содержит число строк, измененных последней выполненной инструкцией SQL. После подачи инструкции UPDATE вы можете прочитать значение атрибута rowcount, чтобыяснить число обновленных строк таблицы. Следующий интерактивный сеанс это демонстрирует. Предположим, что мы подсоединены к базе данных chocolate.db и cur является объектом Cursor:

```

1 >>> cur.execute(''UPDATE Products
2 . . .           SET UnitsOnHand = 0
3 . . .           WHERE Description LIKE "%шоколада%"'')
4 <sqlite3.Cursor object at 0x035432A0>
5 >>> conn.commit()
6 >>> print(cur.rowcount)
7 6
8 >>>

```

В этом сеансе инструкция UPDATE в строках 1–3 изменяет значение в столбце UnitsOnHand на 0 для каждой строки, в которой столбец Description содержит подстроку "шоколада". Распечатав значение атрибута cur.rowcount в строке 6, мы увидим, что было обновлено 6 строк.

Удаление строк с помощью инструкции *DELETE*

В SQL существует инструкция DELETE для удаления одной или нескольких строк из таблицы. Общий формат инструкции DELETE:

`DELETE FROM Таблица WHERE Критерий`

Здесь *Таблица* — это имя таблицы; *Критерий* — условное выражение. Вот инструкция DELETE, которая удалит строку, в которой ProductID равен 10:

`DELETE FROM Products WHERE ProductID == 10`

Эта инструкция отыскивает строку в таблице Products, в столбце ProductID которой имеется значение 10, и удаляет эту строку.

С помощью инструкции DELETE можно удалять несколько строк. Например, посмотрите на следующую инструкцию:

`DELETE FROM Products WHERE Description LIKE "%шоколада%"`

Эта инструкция удалит все строки в таблице Products, в которой находится столбец Description с подстрокой "шоколада". Если снова посмотрите на табл. 14.2, то увидите, что будут удалены шесть строк.



ВНИМАНИЕ!

Будьте осторожны и не пропустите блок WHERE с условным выражением при использовании инструкции DELETE. Вы можете удалить все строки таблицы! Например, посмотрите на следующую ниже инструкцию:

`DELETE FROM Products`

Поскольку в этой инструкции нет выражения WHERE, она удалит все строки таблицы Products!

Программа 14.13 демонстрирует удаление строки в таблице Products. Пользователь вводит существующий ID изделия, а программа выводит на экран описание этого изделия и запрос на подтверждение удаления строки таблицы. Если пользователь соглашается с удалением, программа удаляет строку таблицы.

Программа 14.13 (product_deleter.py)

```

1 import sqlite3
2
3 def main():
4     # Подсоединиться к базе данных.
5     conn = sqlite3.connect('chocolate.db')
6
7     # Получить курсор.
8     cur = conn.cursor()
9
10    # Получить от пользователя ID изделия.
11    pid = int(input('Введите ID изделия для его удаления: '))
12
13    # Получить описание этого изделия.
14    cur.execute(''':SELECT Description FROM Products
15                  WHERE ProductID == ?''', (pid,))
16    results = cur.fetchone()
17
18    # Если ID изделия найден, то продолжить...
19    if results != None:
20        # Подтвердить желание удалить изделие.
21        sure = input(f'Вы уверены, что хотите удалить '
22                    f'{results[0]}? (д/н): ')
23
24        # Если да, то удалить изделие.
25        if sure.lower() == 'д':
26            cur.execute(''':DELETE FROM Products
27                          WHERE ProductID == ?''',
28            (pid,))
29
30            # Зафиксировать изменения.
31            conn.commit()
32            print('Изделие было удалено.')
33        else:
34            # Сообщение об ошибке.
35            print(f'ID изделия {pid} не найден.')
36
37    # Закрыть соединение с базой данных.
38    conn.close()
39
40 # Вызвать главную функцию.
41 if __name__ == '__main__':
42     main()

```

Вывод программы (ввод выделен полужирным шрифтом)

Введите ID изделия для его удаления: 10

Вы уверены, что хотите удалить Стружка из белого шоколада? (д/н): д

Изделие было удалено.

Давайте рассмотрим эту программу подробнее. В главной функции строка 5 программы получает соединение с базой данных, а строка 8 — объект `Cursor`. Стока 11 запрашивает у пользователя ID изделия, который присваивается переменной `pid`.

Инструкция в строках 14 и 15 выполняет SQL-запрос, который отыскивает строку таблицы, в которой столбец `ProductID` равен переменной `pid`. Если такая строка найдена, то извлекается столбец `Description`. Инструкция в строке 16 вызывает метод `fetchone()` для извлечения результатов запроса в виде кортежа и присваивает его переменной `results`. (Если SQL-запрос не найдет соответствующей строки, то метод `fetchone()` вернет `None`.)

Если переменная `results` не равна `None`, то инструкция `if` в строке 19 исполнит фрагмент кода, который появляется в строках 20–32. В строках 21–22 выводится подсказка, включающая описание продукта и запрашивающая у пользователя подтвердить свое желание удалить строку. Инструкция `if` в строке 25 определяет, ввел ли пользователь `d` или `D`. Если это так, то инструкция в строках 26–28 программы удаляет строку из таблицы `Products`. В строке 31 программы фиксируются изменения в базе данных, а в строке 32 выводится сообщение, подтверждающее удаление продукта.

Блок в строке 33 выполняется, если пользователь ввел ID изделия, который не был найден в таблице `Products`. В строке 35 он просто выводит на экран сообщение об этом. Стока 38 закрывает соединение с базой данных.

Определение числа удаленных строк

После исполнения инструкции `DELETE` вы можете прочитать значение атрибута `rowcount` объекта `Cursor`, чтобы выяснить количество удаленных строк. Следующий интерактивный сеанс это демонстрирует. Предположим, что мы подсоединены к базе данных `chocolate.db` и `cur` является объектом `Cursor`:

```
1 >>> cur.execute('''DELETE FROM Products
2 ...          WHERE Description LIKE "%Стружка%"''')
3 <sqlite3.Cursor object at 0x035432A0>
4 >>> conn.commit()
5 >>> print(cur.rowcount)
6 2
7 >>>
```

В этом сеансе инструкция `DELETE` в строках 1–2 удаляет все строки таблицы, в которых столбец `Description` содержит подстроку "Стружка". Распечатав значение атрибута `cur.rowcount` в строке 5, мы увидим, что было удалено 2 строки.



Контрольная точка

- 14.32. Напишите инструкцию SQL, которая поменяет цену всех изделий с шоколадной стружкой в таблице `Products` базы данных `chocolate.db` на 4.99.
- 14.33. Напишите инструкцию SQL, которая удалит все строки в таблице `Products` базы данных `chocolate.db`, стоимость единицы которых превышает 4.00.

14.8 Подробнее о первичных ключах

Ключевые положения

Первичный ключ однозначно идентифицирует каждую строку в таблице. В SQLite каждая таблица содержит целочисленный (INTEGER) идентификационный столбец с именем RowID. Если вы создадите столбец, который в таблице является целочисленным первичным ключом (INTEGER PRIMARY KEY), то этот столбец станет псевдонимом столбца RowID. Составной ключ — это ключ, созданный путем совмещения двух или более существующих столбцов.

В реальном мире нам нужны способы установления личности людей и идентичности вещей. Например, если вы являетесь учеником, то в вашей школе, вам, вероятно, присвоили идентификационный номер ученика. Ваш идентификационный номер является уникальным. Ни у одного другого ученика в школе нет такого же номера, как у вас. В любой момент, когда школе необходимо сохранить о вас некоторые данные, например оценку на одном из уроков, количество учебных часов, которые были отведены для изучения предмета, и т. д., ваш идентификационный номер сохраняется вместе с этими данными. Есть много других реальных примеров уникальных чисел, которые используются для идентификации людей и вещей. Водителям присваиваются номера водительских прав. Индивидуальные номера назначаются сотрудникам. Серийные номера присваиваются изделиям и продуктам.

В таблице базы данных важно иметь способ идентифицировать каждую строку в таблице. Для этих целей служит *первичный ключ*. В самой простой форме первичный ключ — это столбец, содержащий уникальное значение для каждой строки таблицы. Ранее в этой главе вы видели примеры использования базы данных chocolate.db. В этой базе данных таблица Products содержит целочисленный (INTEGER) столбец ProductID, который используется в качестве первичного ключа. Напомним, что столбец ProductID также является идентификационным столбцом. Это означает, что всякий раз, когда в таблицу добавляется новая строка, если для столбца ProductID не указано значение, то СУБД генерирует значение для этого столбца автоматически.

Вот некоторые общие правила, которые следует помнить о первичных ключах.

- ◆ Первичные ключи должны содержать значение. Они не могут быть пустыми, т. е. иметь NULL.
- ◆ Значение первичного ключа каждой строки должно быть уникальным. Никакие две строки в таблице не могут иметь один и тот же первичный ключ.
- ◆ Таблица может иметь только один первичный ключ. Однако несколько столбцов могут быть объединены для создания составного ключа. Далее мы обсудим составные ключи.

Столбец RowID в SQLite

В SQLite при создании таблица автоматически будет иметь целочисленный столбец с именем RowID. СУБД SQLite использует столбец RowID в своих внутренних алгоритмах для доступа к данным таблицы. Столбец RowID наращивается автоматически, т. е. автоинкрементируется. Всякий раз, когда в таблицу добавляется новая строка, столбцу RowID присваивается целочисленное значение, которое на 1 больше наибольшего значения, хранящегося в данный момент в столбце RowID.

Если вы исполните инструкцию `SELECT`, такую как `SELECT * FROM ИмяТаблицы`, то вы не увидите содержимое столбца `RowID` среди результатов. Тем не менее вы можете в явной форме выполнять поиск по столбцу `RowID` с помощью такой инструкции, как `SELECT RowID FROM ИмяТаблицы`. Например, предположим, что в следующем интерактивном сеансе мы подсоединились к базе данных `chocolate.db` и `cur` является объектом `Cursor` для таблицы `Products`:

```
>>> cur.execute('SELECT RowID FROM Products')
<sqlite3.Cursor object at 0x0030FD20>
>>> cur.fetchall()
[(1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,), (10,)]
```

При добавлении строки в таблицу с помощью инструкции `INSERT` можно в явной форме указать значение для столбца `RowID`, если это значение является уникальным. Если вы укажете значение для столбца `RowID`, которое уже используется в другой строке, то СУБД выдаст исключение.

Вы также можете изменить существующее значение столбца `RowID` с помощью инструкции `UPDATE`, если новое значение является уникальным. Если вы попытаетесь изменить значение столбца `RowID` на значение, которое уже используется в другой строке, то СУБД выдаст исключение.

SQLite неявно запрещает присваивать `NULL` столбцу `RowID`. Если вы попытаетесь присвоить `NULL` столбцу `RowID`, то СУБД просто назначит этому столбцу автоинкрементированное целое число.

Целочисленные первичные ключи в SQLite

Когда вы назначаете в SQLite столбец в качестве целочисленного первичного ключа (`INTEGER PRIMARY KEY`), этот столбец становится псевдонимом для столбца `RowID`. Всякий раз, когда вы работаете с целочисленным первичным ключом таблицы, вы на самом деле работаете со столбцом `RowID`. Например, предположим, что в следующем интерактивном сеансе мы подсоединились к базе данных `chocolate.db`, а `cur` является объектом `Cursor` для таблицы `Products`:

```
>>> cur.execute('SELECT ProductID FROM Products')
<sqlite3.Cursor object at 0x0030FD20>
>>> cur.fetchall()
[(1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,), (10,)]
```

Этот сеанс получает значения столбца `ProductID`, которые равны 1, 2, 3, ..., 10. Напомним, что столбец `ProductID` является целочисленным первичным ключом. Если мы запросим значения в столбце `RowID`, то увидим те же значения, что и в следующем интерактивном сеансе:

```
>>> cur.execute('SELECT RowID FROM Products')
<sqlite3.Cursor object at 0x0030FD20>
>>> cur.fetchall()
[(1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,), (10,)]
```

Поскольку целочисленный первичный ключ — это просто псевдоним столбца `RowID`, все описанные ранее характеристики столбца `RowID` применяются к целочисленным первичным ключам.

Первичные ключи, отличные от целочисленных

До сих пор в этой главе мы использовали в наших примерах только целочисленные первичные ключи. Однако вы можете назначить для таблицы любой тип столбца в качестве первичного ключа. Например, предположим, что вы создаете базу данных для хранения данных о сотрудниках, и каждый сотрудник имеет уникальный идентификатор, состоящий из букв и цифр. Поскольку этот ID каждого сотрудника является уникальным, его можно использовать в качестве первичного ключа, как показано в следующей инструкции SQL:

```
CREATE TABLE Employees (EmployeeID TEXT PRIMARY KEY NOT NULL,  
                      Name TEXT,  
                      PayRate REAL)
```

В этом примере столбец EmployeeID объявляется как TEXT и как первичный ключ. Мы также применили ограничение NOT NULL к столбцу, чтобы предотвратить присвоение пустых значений.

При использовании нецелочисленного первичного ключа необходимо обязательно присваивать столбцу первичного ключа уникальное ненулевое (не-NULL) значение при каждом добавлении строки в таблицу. В противном случае произойдет ошибка.

ПРИМЕЧАНИЕ

 Важно использовать ограничение NOT NULL с первичными ключами, потому что без него SQLite позволит вам хранить в первичном ключе пустые значения. Единственное исключение — это ситуация, в которой столбец объявляется как целочисленный первичный ключ (INTEGER PRIMARY KEY). В любое время, когда вы назначаете NULL целочисленному первичному ключу, СУБД будет назначать столбцу автоинкрементированное целое число. Это связано с тем, что целочисленный первичный ключ является просто псевдонимом для столбца RowID.

Составные ключи

Первичные ключи должны содержать уникальные данные для каждой строки таблицы. Однако иногда в таблице нет столбцов, содержащих уникальные данные. В этом случае ни один из существующих столбцов не может использоваться в качестве первичного ключа. Например, предположим, что мы разрабатываем таблицу базы данных, содержащую следующие данные об учебных занятиях в кампусе колледжа:

- ◆ номер учебной аудитории — номера в каждом здании пронумерованы, например, 101, 102 и т. д.;
- ◆ название здания — каждое здание в кампусе имеет такое название, как "Машиностроение", "Биология" или "Коммерция";
- ◆ вместимость — в каждой аудитории максимальное число мест, например 20, 50 или 100.

Мы решаем, что таблица базы данных будет называться Classrooms, и в ней будут следующие столбцы:

- ◆ RoomNumber (Номер учебной аудитории) — INTEGER;
- ◆ Building (Здание) — TEXT;
- ◆ Seats (Места) — INTEGER.

При разработке таблицы базы данных мы понимаем, что ни один из этих элементов не может быть использован в качестве первичного ключа. Номера учебных аудиторий не уни-

кальны, потому что в каждом здании есть номера, которые пронумерованы 101, 102, 103 и т. д. Как следствие, столбец RoomNumber будет содержать несколько строк с одинаковым значением. Кроме того, назания зданий нельзя использовать, потому что в каждом здании есть несколько учебных аудиторий, и в результате у нас будет несколько строк с одинаковым значением в столбце Building. Столбец Seats также не уникален, поскольку несколько учебных аудиторий имеют одинаковую вместимость, поэтому несколько строк будут иметь одинаковое значение в столбце Seats.

Для создания первичного ключа у нас есть два варианта. Первый вариант — добавить идентификационный столбец, т. е. целочисленный первичный ключ. Второй вариант — создать *составной ключ*, представляющий собой два или более столбцов, которые объединяются для создания уникального значения. В нашей таблице Classrooms мы могли бы объединить столбец RoomNumber и столбец Building, чтобы создать уникальное значение. Хотя есть несколько комнат, которые пронумерованы 101, есть только одна аудитория 101 для биологии и только одна аудитория 101 для машиностроения.

В SQL мы создаем составной ключ, используя следующий общий формат инструкции CREATE TABLE:

```
CREATE TABLE ИмяТаблицы (ИмяСтолбца1 ТипДанных1,
                           ИмяСтолбца2 ТипДанных2,
                           ...
                           PRIMARY KEY(ИмяСтолбца1, ИмяСтолбца2, ...))
```

Обратите внимание, что после списка объявлений столбцов следует табличное ограничение PRIMARY KEY, за которым указан список имен столбцов в круглых скобках. Столбцы, перечисленные в скобках, объединяются, создавая первичный ключ. Вот пример того, как можно создать таблицу Classrooms с используемыми в качестве составного ключа столбцами RoomNumber и Building:

```
CREATE TABLE Classrooms (RoomNumber INTEGER NOT NULL,
                           Building TEXT NOT NULL,
                           Seats INTEGER,
                           PRIMARY KEY(RoomNumber, Building))
```

Обратите внимание, что мы применили ограничение NOT NULL как к столбцу roomNumber, так и к столбцу Building. Это важно, потому что первичный ключ не может быть NULL.



Контрольная точка

- 14.34. Допустимо ли, чтобы первичный ключ был NULL?
- 14.35. Может ли первичный ключ двух или более строк в таблице иметь одно и то же значение?
- 14.36. Каков тип данных столбца RowID?
- 14.37. Предположим, что столбец RowID таблицы содержит следующие значения: 1, 2, 3, 7 и 99. Если вы добавляете новую строку в таблицу без явного присвоения значения в столбце RowID новой строки, какое значение СУБД автоматически присвоит в этом столбце?
- 14.38. Какова связь в SQLite между столбцом INTEGER PRIMARY KEY и столбцом RowID?
- 14.39. Что такое составной ключ?

14.9

Обработка исключений базы данных

Ключевые положения

Модуль `sqlite3` определяет собственные исключения, которые вызываются при возникновении ошибки базы данных.

Модуль `sqlite3` определяет исключение с именем `Error`, которое вызывается при каждом возникновении ошибки базы данных. В целях изящной обработки этих ошибок вы должны создавать свой код для работы с базой данных внутри инструкций `try/except`. В следующем общем формате показан один из способов использования инструкции `try/except` для обработки ошибок базы данных:

```
1 conn = None
2 try:
3     conn = sqlite3.connect(ИмяБазыДанных)
4     cur = conn.Cursor()
5
6     # Здесь выполнить операции базы данных.
7
8 except sqlite3.Error:
9
10    # Ответить на исключение базы данных.
11
12 except Exception:
13
14    # Ответить на общее исключение.
15
16 finally:
17    if conn != None:
18        conn.close()
```

В этом общем формате мы начинаем с присвоения значения `None` переменной `conn` в строке 1. Затем мы выполняем все операции с базой данных внутри блока `try` (строки 3–7). Если какой-либо программный код внутри блока `try` вызовет исключение базы данных, то программа перейдет к блоку `except` в строке 8. Если какой-либо программный код в блоке `try` вызовет общее исключение не из базы данных, то программа перейдет к блоку `except` в строке 12.

Программный код, который появляется в блоке `finally`, всегда исполняется, независимо от того, возникает ли исключение. Именно здесь мы закрываем соединение с базой данных. Инструкция `if` в строке 17 проверяет значение переменной `conn`. Если переменная `conn` не равна `None`, то мы знаем, что соединение с базой данных было успешно открыто (в строке 3), и можем выполнить инструкцию в строке 18, чтобы закрыть соединение. Однако если переменная `conn` все еще имеет значение `None`, то мы знаем, что соединение с базой данных так и не было открыто, поэтому нет причин его закрывать. В программе 14.14 представлен пример.

Программа 14.14 (exception_handling.py)

```

1 import sqlite3
2
3 def main():
4     # Переменная управления циклом.
5     again = 'д'
6
7     while (again == 'д'):
8         # Получить ID товара, название и цену.
9         item_id = int(input('ID товара: '))
10        item_name = input('Название товара: ')
11        price = float(input('Цена: '))
12
13        # Добавить товар в базу данных.
14        add_item(item_id, item_name, price)
15
16        # Добавить еще одну?
17        again = input('Добавить еще одну позицию? (д/н): ')
18
19 # Функция add_item добавляет позицию в базу данных.
20 def add_item(item_id, name, price):
21     # Инициализировать переменную соединения.
22     conn = None
23
24     try:
25         # Подсоединиться к базе данных.
26         conn = sqlite3.connect('inventory.db')
27
28         # Получить курсор.
29         cur = conn.cursor()
30
31         # Добавить позицию в таблицу Inventory.
32         cur.execute('''INSERT INTO Inventory (ItemID, ItemName, Price)
33                         VALUES (?, ?, ?)''',
34                         (item_id, name, price))
35
36         # Зафиксировать изменения.
37         conn.commit()
38
39     except sqlite3.Error as err:
40         print(err)
41
42     finally:
43         # Закрыть соединение.
44         if conn != None:
45             conn.close()
46

```

```
47 # Вызвать главную функцию.
48 if __name__ == '__main__':
49     main()
```

Вывод программы (ввод выделен жирным шрифтом)

```
ID товара: 1 Enter
Название товара: Отбойный молоток Enter
Цена: 299.99 Enter
UNIQUE constraint failed: Inventory.ItemID
Добавить еще одну позицию? (д/н): и Enter
```

Эта программа позволяет пользователю добавлять новую товарную позицию в таблицу `Inventory` базы данных `inventory.db`. Напомним, что столбец `ItemID` является целочисленным первичным ключом. В демонстрационном выводе программы пользователь пытается добавить строку с уже существующим `ItemID`, что приводит к возникновению исключения из программного кода в строках 27–29. Вместо аварийного отказа программы мы обрабатываем исключение с помощью инструкции `try/except`.



Контрольная точка

14.40. Какое исключение вы должны обрабатывать, чтобы изящно реагировать на ошибки базы данных SQLite?

14.10 Операции CRUD

Ключевые положения

Четырьмя базовыми операциями приложения базы данных являются создание, чтение, обновление и удаление.

Аббревиатура CRUD образована из английских слов `Create`, `Read`, `Update` и `Delete`, обозначающих операции создания, чтения, обновления и удаления. Это четыре базовые операции, выполняемые приложением базы данных. Вот краткое описание каждой операции:

- ◆ **создание** — это процесс создания нового набора данных в базе данных. Он осуществляется с помощью SQL-инструкции `INSERT`;
- ◆ **чтение** — это процесс чтения существующего набора данных из базы данных. Он осуществляется с помощью SQL-инструкции `SELECT`;
- ◆ **обновление** — это процесс изменения или обновления существующего набора данных в базе данных. Он осуществляется с помощью SQL-инструкции `UPDATE`;
- ◆ **удаление** — это процесс удаления набора данных из базы данных. Он осуществляется с помощью SQL-инструкции `DELETE`.

В ЦЕНТРЕ ВНИМАНИЯ

Приложение CRUD по ведению учета инвентаря

Давайте рассмотрим пример приложения CRUD. Программа 14.15 содержит исходный код, выполняющий CRUD-операции с базой данных `inventory.db`, которую вы видели ранее.

в этой главе. В базе данных есть таблица `Inventory`, которая содержит следующие столбцы и данные:

ItemID	ItemName	Price
1	Отвертка	4.99
2	Молоток	12.99
3	Плоскогубцы	14.99
4	Пила	24.99
5	Дрель	89.99
6	Рулетка	8.99

Столбец `ItemID` имеет тип `INTEGER` и является целочисленным первичным ключом (`INTEGER PRIMARY KEY`), столбец `ItemName` имеет тип `TEXT`, столбец `Price` имеет тип `REAL`.

Эта программа работает под управлением меню, т. е. она выводит на экран список вариантов, из которых пользователь может выбирать. При запуске программы на экран выводится следующее меню:

----- Меню ведения учета инструментов -----
 1. Создать новую позицию
 2. Прочитать позицию
 3. Обновить позицию
 4. Удалить позицию
 5. Выйти из программы

Введите свой вариант действия:

Пользователь может ввести целое число в диапазоне 1–5 для выполнения действия. Если пользователь хочет создать новую позицию в базе данных, он вводит число 1. Например:

----- Меню ведения учета инструментов -----
 1. Создать новую позицию
 2. Прочитать позицию
 3. Обновить позицию
 4. Удалить позицию
 5. Выйти из программы

Введите свой вариант: 1

Создать новую позицию

Название позиции: Отбойный молоток

Цена: 299.99

Программа запрашивает у пользователя название и цену позиции. После ввода цены в базе данных создается позиция и снова появляется меню. Если пользователь хочет прочитать элемент из базы данных, то он вводит число 2, как показано ниже:

----- Меню ведения учета инструментов -----
 1. Создать новую позицию
 2. Прочитать позицию
 3. Обновить позицию

4. Удалить позицию

5. Выйти из программы

Введите ваш вариант: 2 [Enter]

Введите название искомой позиции: Отбойный молоток [Enter]

ID: 7 Название: Отбойный молоток Цена: 299.99

1 строк(а) найдено.

Пользователь вводит имя искомой позиции, и программа выполняет поиск без учета регистра позиций, соответствующих этому имени. Если какие-либо позиции найдены, то на экран выводятся их идентификатор, название и цена. Меню появляется снова, и если пользователь хочет обновить позицию, то он вводит число 3, как показано ниже:

----- Меню ведения учета инструментов -----

1. Создать новую позицию

2. Прочитать позицию

3. Обновить позицию

4. Удалить позицию

5. Выйти из программы

Введите ваш вариант: 3 [Enter]

Введите название искомой позиции: Отбойный молоток [Enter]

ID: 7 Название: Отбойный молоток Цена: 299.99

1 строк(а) найдено.

Выберите ID обновляемой позиции: 7 [Enter]

Введите новое название позиции: Мощный отбойный молоток [Enter]

Введите новую цену: 399.99 [Enter]

1 строк(а) обновлено.

Пользователь вводит имя позиции, и программа выполняет поиск позиций, соответствующих этому имени, без учета регистра. Если какие-либо товары найдены, то на экран выводятся их идентификатор, название и цена. Возможно, что будет найдено несколько совпадающих позиций, поэтому программа предложит пользователю ввести идентификатор нужной позиции. Затем программа предложит пользователю ввести новое название позиции и цену. Затем выбранная позиция обновится новыми значениями, и снова появится меню.

Если пользователь хочет удалить элемент, он вводит в меню число 4, как показано ниже:

----- Меню ведения учета инструментов -----

1. Создать новую позицию

2. Прочитать позицию

3. Обновить позицию

4. Удалить позицию

5. Выйти из программы

Введите ваш вариант: 4 [Enter]

Введите название искомой позиции: Мощный отбойный молоток [Enter]

ID: 7 Название: Мощный отбойный молоток Цена: 399.99

1 строк(а) найдено.

Выберите ID удаляемой позиции: 7 [Enter]

Вы уверены, что хотите удалить эту позицию? (д/н): д [Enter]

1 строк(а) удалено.

Пользователь вводит имя элемента, и программа выполняет поиск элементов, соответствующих этому имени, без учета регистра. Если какие-либо позиции найдены, то на экран выводятся их идентификатор, название и цена.

Возможно, что будет найдено несколько совпадающих позиций, поэтому программа предложит пользователю ввести идентификатор нужной позиции. Затем программа попросит пользователя подтвердить свое желание удалить позицию. Если пользователь отвечает д или Д, то позиция удаляется. В противном случае позиция не удаляется и снова появляется меню.

Для выхода из программы пользователь вводит в меню число 5.

Программа 14.15 (inventory_crud.py)

```
1 import sqlite3
2
3 MIN_CHOICE = 1
4 MAX_CHOICE = 5
5 CREATE = 1
6 READ = 2
7 UPDATE = 3
8 DELETE = 4
9 EXIT = 5
10
11 def main():
12     choice = 0
13     while choice != EXIT:
14         display_menu()
15         choice = get_menu_choice()
16
17         if choice == CREATE:
18             create()
19         elif choice == READ:
20             read()
21         elif choice == UPDATE:
22             update()
23         elif choice == DELETE:
24             delete()
25
26 # Функция display_menu выводит на экран главное меню.
27 def display_menu():
28     print('\n----- Меню ведения учета инструментов -----')
29     print('1. Создать новую позицию')
30     print('2. Прочитать позицию')
31     print('3. Обновить позицию')
32     print('4. Удалить позицию')
33     print('5. Выйти из программы')
34
```

```
35 # Функция get_menu_choice получает от пользователя пункт меню.
36 def get_menu_choice():
37     # Получить от пользователя вариант действия.
38     choice = int(input('Введите ваш вариант: '))
39
40     # Проверить входные данные.
41     while choice < MIN_CHOICE or choice > MAX_CHOICE:
42         print(f'Допустимые варианты таковы: {MIN_CHOICE} - {MAX_CHOICE}.')
43         choice = int(input('Введите ваш вариант: '))
44
45     return choice
46
47 # Функция create создает новую позицию.
48 def create():
49     print('Создать новую позицию')
50     name = input('Название позиции: ')
51     price = input('Цена: ')
52     insert_row(name, price)
53
54 # Функция read читает существующую позицию.
55 def read():
56     name = input('Введите название искомой позиции: ')
57     num_found = display_item(name)
58     print(f'{num_found} строк(а) найдено.')
59
60 # Функция update обновляет данные существующей позиции.
61 def update():
62     # Сначала показать пользователю найденные строки.
63     read()
64
65     # Получить ID выбранной позиции.
66     selected_id = int(input('Выберите ID обновляемой позиции: '))
67
68     # Получить новые значения для названия и цены.
69     name = input('Введите новое название позиции: ')
70     price = input('Введите новую цену: ')
71
72     # Обновить строку.
73     num_updated = update_row(selected_id, name, price)
74     print(f'{num_updated} строк(а) обновлено.')
75
76 # Функция delete удаляет позицию.
77 def delete():
78     # Сначала показать пользователю найденные строки.
79     read()
80
```

```
81     # Получить ID выбранной позиции.
82     selected_id = int(input('Выберите ID удаляемой позиции: '))
83
84     # Подтвердить удаление.
85     sure = input('Вы уверены, что хотите удалить эту позицию? (д/н): ')
86     if sure.lower() == 'д':
87         num_deleted = delete_row(selected_id)
88         print(f'{num_deleted} строк(а) удалено.')
89
90 # Функция insert_row вставляет строку в таблицу Inventory.
91 def insert_row(name, price):
92     conn = None
93     try:
94         conn = sqlite3.connect('inventory.db')
95         cur = conn.cursor()
96         cur.execute(''')INSERT INTO Inventory (ItemName, Price)
97                         VALUES (?, ?)'',
98                         (name, price))
99         conn.commit()
100    except sqlite3.Error as err:
101        print('Ошибка базы данных', err)
102    finally:
103        if conn != None:
104            conn.close()
105
106 # Функция display_item выводит на экран все позиции
107 # с совпадающими названиями позиций.
108 def display_item(name):
109     conn = None
110     results = []
111     try:
112         conn = sqlite3.connect('inventory.db')
113         cur = conn.cursor()
114         cur.execute(''')SELECT * FROM Inventory
115                         WHERE ItemName == ?'''',
116                         (name,))
117         results = cur.fetchall()
118
119         for row in results:
120             print(f'ID: {row[0]} Название: {row[1]} '
121                   f'Цена: {row[2]}')
122     except sqlite3.Error as err:
123         print('Ошибка базы данных', err)
124     finally:
125         if conn != None:
126             conn.close()
```

```
127     # Вернуть число совпавших строк.
128     return len(results)
129
130 # Функция update_row обновляет существующую строку новыми
131 # названием и ценой. Возвращается обновленное число строк.
132 def update_row(id, name, price):
133     conn = None
134     try:
135         conn = sqlite3.connect('inventory.db')
136         cur = conn.cursor()
137         cur.execute(''':UPDATE Inventory
138             SET ItemName = ?, Price = ?
139             WHERE ItemID == ?''',
140             (name, price, id))
141         conn.commit()
142         num_updated = cur.rowcount
143     except sqlite3.Error as err:
144         print('Ошибка базы данных', err)
145     finally:
146         if conn != None:
147             conn.close()
148
149     return num_updated
150
151 # Функция delete_row удаляет существующую позицию.
152 # Возвращается число удаленных строк.
153 def delete_row(id):
154     conn = None
155     try:
156         conn = sqlite3.connect('inventory.db')
157         cur = conn.cursor()
158         cur.execute(''':DELETE FROM Inventory
159             WHERE ItemID == ?''',
160             (id,))
161         conn.commit()
162         num_deleted = cur.rowcount
163     except sqlite3.Error as err:
164         print('Ошибка базы данных', err)
165     finally:
166         if conn != None:
167             conn.close()
168
169     return num_deleted
170
171 # Вызвать главную функцию.
172 if __name__ == '__main__':
173     main()
```

Давайте рассмотрим программный код подробнее.

Глобальные константы. Строки 3–9 определяют глобальные константы, которые используются в сочетании с меню. `MIN_CHOICE` — это наименьшее число, которое пользователь может ввести при выборе пункта меню, а `MAX_CHOICE` — наибольшее число. Константы `CREATE`, `READ`, `UPDATE`, `DELETE` и `EXIT` содержат номера пунктов меню, которые пользователь может ввести для выбора действия.

Функция `main`, или главная функция, использует цикл `while` для многократного вывода меню на экран и выполнения выбранного пользователем действия. Цикл повторяется до тех пор, пока пользователь не введет число 5, чтобы выйти из программы. Во время каждой итерации цикл вызывает функцию `display_menu` в строке 14, а затем функцию `get_menu_choice` в строке 15. Инструкция `if-elif` в строках 17–24 вызывает соответствующую функцию для выполнения выбранного пользователем действия.

Функция `display_menu` появляется в строках 27–33 и выводит меню на экран. Она вызывается функцией `main`.

Функция `get_menu_choice` появляется в строках 36–45 и получает выбранный пользователем пункт меню. Выбранный вариант проверяется в цикле в строках 41–43. Как только правильный вариант будет введен, управление передается обратно в вызвавшую функцию. Эта функция вызывается функцией `main`.

Функция `create` появляется в строках 48–52. Она вызывается функцией `main`, когда пользователь выбирает пункт меню 1. Функция получает название и цену позиции от пользователя, а затем передает эти значения в функцию `insert_row`.

Функция `read` появляется в строках 55–58. Она вызывается функцией `main`, когда пользователь выбирает пункт меню 2. (Эта функция также вызывается функциями `update` и `delete`.) Функция `read` получает название позиции от пользователя, а затем передает это название в функцию `display_item`. Функция `display_item` выводит на экран все строки таблицы `Inventory`, в которой столбец `ItemName` соответствует названию, введенному пользователем. Функция `display_item` также возвращает число найденных строк, и это значение выводится на экран.

Функция `update` появляется в строках 61–74. Она вызывается функцией `main`, когда пользователь выбирает пункт меню 3. В строке 63 вызывается функция `read`, которая позволяет пользователю найти строки таблицы `Inventory`, соответствующие названию позиции. Возможно, что функция `read` покажет несколько строк, поэтому строка 66 предлагает пользователю ввести ID той, которая должна быть обновлена. Затем строки 69 и 70 получают новые значения для названия и цены позиции. Стока 73 передает ID позиции и новые значения для названия и цены в функцию `update_row`, которая обновляет эту позицию в базе данных. Функция `update_row` возвращает число строк, которые были обновлены, и это значение выводится на экран.

Функция `delete` появляется в строках 77–88. Она вызывается функцией `main`, когда пользователь выбирает пункт меню 4. В строке 79 вызывается функция `read`, которая позволяет пользователю отыскать строки в таблице `Inventory`, соответствующие названию позиции. Возможно, что функция `read` выведет на экран несколько строк, поэтому строка 82 предлагает пользователю ввести ID той, которая должна быть удалена. Затем в строке 85 пользователю предлагается подтвердить свое желание удалить позицию. Если пользователь вводит `д` или `Д`, то ID позиции передается в функцию `delete_row`, которая удаляет эту позицию из

базы данных. Функция `delete_row` возвращает число строк, которые были удалены, и это значение выводится на экран.

Функция `insert_row` появляется в строках 91–104. Она вызывается функцией `create` и принимает название и цену новой позиции в качестве аргументов. Функция открывает соединение с базой данных `inventory.db`, получает объект `Cursor` для таблицы `Inventory` и использует SQL-инструкцию `INSERT` для вставки новой позиции в таблицу. Любые возникающие ошибки базы данных обрабатываются инструкцией `try/except`, и соединение с базой данных закрывается в блоке `finally`.

Функция `display_item` появляется в строках 108–128. Она вызывается функцией `read` и принимает название элемента в качестве аргумента. Функция открывает соединение с базой данных `inventory.db`, получает объект `Cursor` для таблицы `Inventory` и использует SQL-инструкцию `SELECT` для получения всех строк, в которых столбец `ItemName` соответствует названию, переданному в качестве аргумента. На экран выводятся результирующие строки. Любые возникающие ошибки базы данных обрабатываются инструкцией `try/except`, а в блоке `finally` закрывается соединение с базой данных. Функция возвращает число выводимых на экран строк.

Функция `update_row` появляется в строках 132–149. Она вызывается функцией `update` и принимает ID, название и цену позиции в качестве аргументов. Функция открывает соединение с базой данных `inventory.db`, получает объект `Cursor` для таблицы `Inventory` и использует SQL-инструкцию `UPDATE` для обновления строки, в которой столбец `ItemID` соответствует ID, переданному в функцию в качестве аргумента. Столбцы `Name` и `Price` совпадающей строки таблицы обновляются названием и ценой, которые были переданы в функцию в качестве аргументов. Любые возникающие ошибки базы данных обрабатываются инструкцией `try/except`, и соединение с базой данных закрывается в блоке `finally`. Функция возвращает число строк, которые были обновлены.

Функция `delete_row` появляется в строках 153–169. Она вызывается функцией `delete` и принимает ID позиции в качестве аргумента. Функция открывает соединение с базой данных `inventory.db`, получает объект `Cursor` для таблицы `Inventory` и использует SQL-инструкцию `DELETE` для удаления строки, в которой столбец `ItemID` соответствует ID, переданному в функцию в качестве аргумента. Любые возникающие ошибки базы данных обрабатываются инструкцией `try/except`, и соединение с базой данных закрывается в блоке `finally`. Функция возвращает число строк, которые были удалены.

14.11 Реляционные данные

Ключевые положения

В реляционной базе данных столбец из одной таблицы может быть ассоциирован со столбцом из других таблиц. Эта ассоциация создает связь между таблицами.

Базы данных должны строиться таким образом, чтобы данные не дублировались. Дублирование данных приводит не только к потере места для хранения, но и к хранению в базе данных несогласованной и конфликтующей информации. Например, предположим, что мы разрабатываем базу данных для хранения сведений о сотрудниках компании, имеющей офисы в нескольких городах. У нас может возникнуть искушение поместить все данные сотрудника в одну таблицу (рис. 14.6).

EmployeeID	Name	Position	Department	City
1	Арлин Мейерс	Директор	Исследования и разработки	Сан-Хосе
2	Джанель Грант	Инженер	Производство	Остин
3	Джек Смит	Менеджер	Маркетинг	Нью-Йорк
4	Соня Альварадо	Аудитор	Бухгалтерский учет	Бостон
5	Рене Кинкейд	Дизайнер	Маркетинг	Нью-Йорк
6	Курт Грин	Супервайзер	Производство	Остин

РИС. 14.6. Данные о сотрудниках

Обратите внимание, что на рис. 14.6 и Джек Смит, и Рене Кинкейд находятся в городском офисе Нью-Йорка. Предположим, компания закрывает офис в Нью-Йорке и переводит всех сотрудников этого офиса в другой город. Как следствие, нам придется обновлять таблицу. Местоположение Нью-Йорк появляется в нескольких строках, поэтому мы должны обеспечить изменение каждой из них. Если мы пропустим одну из строк, то в таблице появятся ошибочные данные.

Также обратите внимание, что и Джанель Грант, и Курт Грин работают в производственном отделе. Предположим, руководство решает изменить название производственного отдела на отдел разработки продукта. И снова нам придется обновлять таблицу, и мы должны обеспечить изменение каждой строки, содержащей слово "Производство" как название отдела.

Во избежание проблем, которые может вызвать дублирование данных, мы должны распределить данные по разным таблицам. Если вы внимательно рассмотрите таблицу на рис. 14.6, то заметите, что у нас есть данные о сотрудниках, данные об отделах и данные о местоположении. Вместо того чтобы хранить все эти данные в одной большой таблице, мы должны разделить их на три таблицы: таблицу отделов *Departments*, таблицу местоположений *Locations* и таблицу сотрудников *Employees*. Вот столбцы, которые мы будем иметь в каждой таблице.

Таблица *Departments*:

- ◆ *DepartmentID* — INTEGER (первичный ключ);
- ◆ *DepartmentName* — TEXT.

Таблица *Locations*:

- ◆ *LocationID* — INTEGER (первичный ключ);
- ◆ *City* — TEXT.

Таблица *Employees*:

- ◆ *EmployeeID* — INTEGER (первичный ключ);
- ◆ *Name* — TEXT;
- ◆ *Position* — TEXT;
- ◆ *DepartmentID* — INTEGER;
- ◆ *LocationID* — INTEGER.

В таблице `Departments` столбец `DepartmentID` является целочисленным первичным ключом (`INTEGER PRIMARY KEY`), а столбец `DepartmentName` содержит название отдела. В таблице `Locations` столбец `LocationID` является целочисленным первичным ключом, а столбец `City` содержит название города.

Таблица `Employees` содержит следующие столбцы:

- ◆ `EmployeeID` — `INTEGER PRIMARY KEY`;
- ◆ `Name` — фамилия и имя сотрудника;
- ◆ `Position` — должность сотрудника;
- ◆ `DepartmentID` — идентификационный номер (`identifier, ID`) отдела сотрудника;
- ◆ `LocationID` — ID местоположения сотрудника.

Когда мы добавляем сотрудника в таблицу `Employees`, вместо сохранения названия отдела мы сохраним ID отдела сотрудника. Названия отделов уже хранятся в таблице `Departments`, поэтому нет необходимости дублировать их в таблице `Employees`. Если нам нужно знать, в каком отделе работает сотрудник, то мы просто получаем ID отдела из строки этого сотрудника в таблице `Employees`, а затем используем его ID для получения названия отдела из таблицы `Departments`.

То же самое относится и к местоположению сотрудника. Вместо того чтобы хранить название города, мы храним ID местоположения. Затем мы можем использовать этот ID для получения названия города из таблицы `Locations`. Если нам нужно знать, в каком городе работает сотрудник, мы просто получаем ID местоположения из строки этого сотрудника в таблице `Employees`, а затем используем этот ID для получения названия города из таблицы `Locations`.

Перенеся повторяющиеся данные в отдельные таблицы, мы не только уменьшаем объем хранилища, необходимый для данных, но и снижаем вероятность появления ошибочных данных в базе данных при их изменении. Например, рассмотрим следующие ситуации, которые могут возникнуть с нашей базой данных сотрудников.

- ◆ Офис в некотором городе переезжает в другой город, и все сотрудники этого офиса также переезжают. Мы обновляем столбец `City` в таблице `Locations`, но оставляем `LocationID` без изменений. Все строки в таблице `Employees`, которые ссылались на старый город, теперь будут ссылаться на новый город.
- ◆ Если название отдела изменяется, мы обновляем название отдела в таблице `Departments`, но оставляем ID отдела без изменений. Все строки в таблице `Employees`, которые ссылались на старое название отдела, теперь будут ссылаться на новое название отдела.

Внешние ключи

Внешний ключ — это столбец в одной таблице, который ссылается на первичный ключ в другой таблице. В таблице `Employees` столбцы `DepartmentID` и `LocationID` являются внешними ключами:

- ◆ столбец `DepartmentID` в таблице `Employees` ссылается на столбец `DepartmentID` в таблице `Departments`. Напомним, что `DepartmentID` является первичным ключом в таблице `Departments`;
- ◆ столбец `LocationID` в таблице `Employees` ссылается на столбец `LocationID` в таблице `Locations`. Напомним, что `LocationID` является первичным ключом в таблице `Locations`.

Когда мы добавляем строку в таблицу `Employees`, значение, которое мы храним в столбце `DepartmentID`, должно соответствовать значению, которое уже хранится в столбце `DepartmentID` таблицы `Departments`. Это создает связь между таблицей `Employees` и таблицей `Departments`. Аналогично значение, которое мы храним в столбце `LocationID` таблицы `Employees`, должно соответствовать значению, которое уже хранится в столбце `LocationID` таблицы `Locations`. Это создает связь между таблицей `Employees` и таблицей `Locations`.

Диаграммы связей между сущностями

Разработчики систем обычно используют диаграммы связей (или отношений) между сущностями для отображения связей между таблицами базы данных. На рис. 14.7 показана диаграмма связей между сущностями для таблиц `Departments`, `Locations` и `Employees` из нашего примера. На диаграмме первичные ключи обозначаются PK (от Primary Key), а внешние ключи — FK (от Foreign Key). Линии, которые соединяют таблицы, показывают, как таблицы связаны между собой. На этой диаграмме есть два типа связей:

- ◆ отношение *"один ко многим"* означает, что для каждой строки в таблице *A* может быть много строк в таблице *B*, которые на нее ссылаются;
- ◆ отношение *"многие к одному"* означает, что многие строки в таблице *A* могут ссылаться на одну строку в таблице *B*.

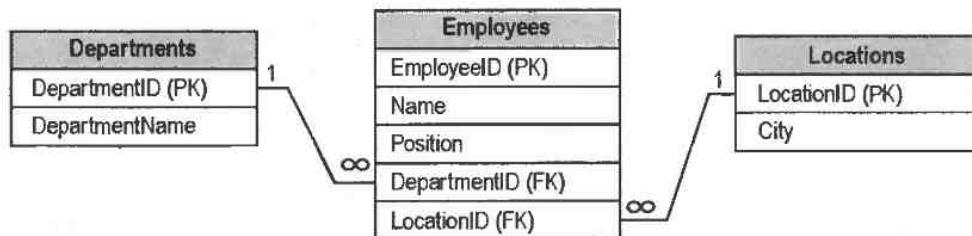


РИС. 14.7. Диаграмма связей между сущностями

Обратите внимание, что на концах соединяющих линий появляется либо 1, либо символ бесконечности (∞). Вы можете интерпретировать символ бесконечности как означающий *"многие"*, а число 1 — как *"один"*. Посмотрите на линию, которая соединяет таблицу `Departments` с таблицей `Employees`. 1 находится на конце линии рядом с таблицей `Departments`, а символ бесконечности — на конце линии рядом с таблицей `Employees`. Это означает, что на одну строку в таблице `Departments` могут ссылаться многие строки в таблице `Employees`. Это имеет смысл, потому что в отделе может быть много сотрудников.

Если мы посмотрим на связь в другом направлении, то увидим, что многие строки в таблице `Employees` могут ссылаться на одну строку в таблице `Departments`. Вот краткое описание всех связей, показанных на диаграмме.

- ◆ Между таблицей `Departments` и таблицей `Employees` существует связь *"один ко многим"*. На одну строку в таблице `Departments` может ссылаться много строк в таблице `Employees`.
- ◆ Между таблицей `Employees` и таблицей `Departments` существует связь *"многие к одному"*. Многие строки в таблице `Employees` могут ссылаться на одну строку в таблице `Departments`.

- ◆ Между таблицей `Locations` и таблицей `Employees` существует связь "один ко многим". На одну строку в таблице `Locations` может ссылаться много строк в таблице `Employees`.
- ◆ Между таблицей `Employees` и таблицей `Locations` существует связь "многие к одному". Многие строки в таблице `Employees` могут ссылаться на одну строку в таблице `Locations`.

Создание внешних ключей на языке SQL

Предположим, мы используем следующий ниже SQL-код для создания таблицы `Departments` и таблицы `Locations` в базе данных:

```
CREATE TABLE Departments(DepartmentID INTEGER PRIMARY KEY NOT NULL,  
                         DepartmentName TEXT)  
CREATE TABLE Locations(LocationID INTEGER PRIMARY KEY NOT NULL,  
                        City TEXT)
```

При создании таблицы `Employees` мы будем использовать табличное ограничение внешнего ключа (`FOREIGN KEY`) для обозначения внешних ключей, как показано ниже:

```
CREATE TABLE Employees(EmployeeID INTEGER PRIMARY KEY NOT NULL,  
                       Name TEXT,  
                       Position TEXT,  
                       DepartmentID INTEGER,  
                       LocationID INTEGER,  
                       FOREIGN KEY(DepartmentID) REFERENCES  
                           Departments(DepartmentID),  
                       FOREIGN KEY(LocationID) REFERENCES  
                           Locations(LocationID))
```

Обратите внимание, что мы использовали два табличных ограничения внешнего ключа. Первое из них указывает на то, что столбец `DepartmentID` ссылается на столбец `DepartmentID` в таблице `Departments`. Второе указывает на то, что столбец `LocationID` ссылается на столбец `LocationID` в таблице `Locations`. Вот общий формат табличных ограничений внешнего ключа:

```
FOREIGN KEY(ИмяСтолбца) REFERENCES ИмяТаблицы(ИмяСтолбца)
```

Ограничение внешнего ключа приведет к тому, что СУБД будет выполнять проверку, когда мы вознамеримся вставить строку в таблицу сотрудников. Мы сможем это сделать только в том случае, если столбец `DepartmentID` содержит допустимое значение из столбца `DepartmentID` таблицы `Departments`, а столбец `LocationID` содержит допустимое значение из столбца `LocationID` таблицы `Locations`. В противном случае произойдет ошибка. Этим обеспечивается целостность связей между двумя таблицами.

Поддержка внешних ключей в SQLite

По умолчанию SQLite не обеспечивает целостность внешних ключей. Если вы хотите обеспечить поддержку внешних ключей в базе данных SQLite, вы должны явно активировать эту функциональность. Допустим, что `cur` является объектом `Cursor`, тогда следующая ниже инструкция активирует поддержку внешних ключей:

```
cur.execute('PRAGMA foreign_keys=ON')
```

Давайте рассмотрим пример того, как можно вставить новую строку в таблицу Employees. Предположим, что мы уже создали базу данных employees.db и таблицы Employees, Departments и Locations содержат значения, показанные на рис. 14.8.

Таблица Employees

EmployeeID	Name	Position	DepartmentID	LocationID
1	Арлин Мейерс	Директор	4	4
2	Джанель Грант	Инженер	2	1
3	Джек Смит	Менеджер	3	3
4	Соня Альварадо	Аудитор	1	2
5	Рене Кинкейд	Дизайнер	3	3
6	Курт Грин	Супервайзер	2	1

Таблица Departments

DepartmentID	DepartmentName
1	Бухгалтерский учет
2	Производство
3	Маркетинг
4	Исследования и разработки

Таблица Locations

LocationID	City
1	Остин
2	Бостон
3	Нью-Йорк
4	Сан-Хосе

РИС. 14.8. Содержимое базы данных employees.db перед добавлением нового сотрудника

Мы только что наняли нового сотрудника по имени Анжела Тейлор в качестве программиста в отдел исследований и разработок, расположенный в Сан-Хосе. Программа 14.16 демонстрирует, каким образом можно добавить новую строку, содержащую ее данные, в таблицу Employees.

Программа 14.16 (add_employee.py)

```

1 import sqlite3
2
3 def main():
4     conn = None
5     try:
6         # Подсоединиться к базе данных и получить курсор.
7         conn = sqlite3.connect('employees.db')
8         cur = conn.cursor()
9
10        # Задействовать поддержку внешнего ключа.
11        cur.execute('PRAGMA foreign_keys=ON')
12
13        # Вставить новую строку в таблицу Employees.
14        cur.execute(''INSERT INTO Employees
15                           (Name, Position, DepartmentID, LocationID)

```

```
16             VALUES
17             ("Анжела Тейлор", "Программист", 4, 4))
18         conn.commit()
19         print('Сотрудник успешно добавлен.')
20     except sqlite3.Error as err:
21         # Если произошло исключение, то напечатать сообщение об ошибке.
22         print(err)
23     finally:
24         # Если соединение открыто, то закрыть его.
25         if conn != None:
26             conn.close()
27
28 # Вызвать главную функцию.
29 if __name__ == '__main__':
30     main()
```

Вывод программы

```
Сотрудник успешно добавлен.
```

Давайте рассмотрим эту программу подробнее. Стока 4 инициализирует переменную `conn` значением `None`. Затем мы входим в инструкцию `try/except` в строке 5. Внутри блока `try` строки 7 и 8 подсоединяются к базе данных и получают объект `cursor`. Стока 11 активирует функциональность поддержания внешних ключей `SQLite`.

В строках 14–17 мы исполняем инструкцию `INSERT`, которая добавляет новые данные о сотруднике в таблицу `Employees`. Обратите внимание, что мы добавляем в столбцы следующие данные:

- ◆ `EmployeeID` — мы не предоставляем значение для этого столбца. Так как это целочисленный первичный ключ, мы позволим СУБД генерировать для него значение;
- ◆ `Name` — Анжела Тейлор;
- ◆ `Position` — программист;
- ◆ `DepartmentID` — 4 (отдел исследований и разработок);
- ◆ `LocationID` — 4 (идентификатор местоположения для Сан-Хосе).

В строке 18 мы фиксируем изменения в базе данных, а в строке 19 печатаем сообщение с информацией о том, что сотрудник был успешно добавлен.

Если из-за ошибки базы данных возникает исключение, то блок `except` в строке 20 его обрабатывает. Инструкция в строке 22 выводит сообщение об ошибке, принятое по умолчанию для данного исключения. Блок `finally` в строке 23 будет выполняться независимо от наличия или отсутствия исключения. Инструкция `if` в строке 25 определяет наличие открытого соединения с базой данных. Если соединение открыто, то инструкция в строке 26 закрывает соединение. После запуска этой программы база данных будет содержать данные, которые приведены на рис. 14.9.

При добавлении новых строк в таблицу `Employees` мы должны помнить о том, что столбцы `DepartmentID` и `LocationID` являются внешними ключами. Любое значение, сохраненное в столбце `DepartmentID` таблицы `Employees`, уже должно иметься в столбце `DepartmentID`

таблицы `Departments`. Кроме того, любое значение, сохраненное в столбце `LocationID`, уже должно иметься в столбце `LocationID` таблицы `Locations`. Программа 14.17 демонстрирует, что произойдет, если попытаться добавить строку, нарушающую одно из этих правил.

Таблица `Employees`

EmployeeID	Name	Position	DepartmentID	LocationID
1	Арлин Мейерс	Директор	4	4
2	Джанель Грант	Инженер	2	1
3	Джек Смит	Менеджер	3	3
4	Соня Альварадо	Аудитор	1	2
5	Рене Кинкейд	Дизайнер	3	3
6	Курт Грин	Супервайзер	2	1
7	Анжела Тейлор	Программист	4	4

Таблица `Departments`

DepartmentID	DepartmentName
1	Бухгалтерский учет
2	Производство
3	Маркетинг
4	Исследования и разработки

Таблица `Locations`

LocationID	City
1	Остин
2	Бостон
3	Нью-Йорк
4	Сан-Хосе

РИС. 14.9. Содержимое базы данных `employees.db` после добавления нового сотрудника

Программа 14.17 (add_bad_employee_data.py)

```

1 import sqlite3
2
3 def main():
4     conn = None
5     try:
6         # Подсоединиться к базе данных и получить курсор.
7         conn = sqlite3.connect('employees.db')
8         cur = conn.cursor()
9
10        # Задействовать поддержку внешнего ключа.
11        cur.execute('PRAGMA foreign_keys=ON')
12
13        # Вставить новую строку в таблицу Employees.
14        cur.execute('''
15            INSERT INTO Employees
16                (Name, Position, DepartmentID, LocationID)
17                VALUES
18                ("Билл Свифт", "Стажер", 99, 1)
19        ''')
20        conn.commit()
21        print('Сотрудник успешно добавлен.')

```

```

20     except sqlite3.Error as err:
21         # Если произошло исключение, то напечатать сообщение об ошибке.
22         print(err)
23     finally:
24         # Если соединение открыто, то закрыть его.
25         if conn != None:
26             conn.close()
27
28 # Вызывать главную функцию.
29 if __name__ == '__main__':
30     main()

```

Вывод программы

```
FOREIGN KEY constraint failed
```

В строке 17 программы обратите внимание на то, что строка, которую мы пытаемся добавить в таблицу `Departments`, содержит 99 в качестве значения для столбца `DepartmentID`. Поскольку в таблице `Departments` нет строки со значением 99 в качестве ID отдела, СУБД выдает исключение.

Обновление реляционных данных

При обновлении строки, имеющей внешний ключ, вы должны быть уверены, что не измените внешний ключ на недопустимое значение. Например, предположим, что мы подсоединились к `employees.db` и `cur` — это объект `Cursor`. Давайте взглянем на следующий интерактивный сеанс:

```

>>> cur.execute('PRAGMA foreign_keys=ON')
<sqlite3.Cursor object at 0x00B5FD20>
>>> cur.execute('''
    UPDATE Employees
    SET DepartmentID = 99
    WHERE Name == "Джек Смит"
''')
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
sqlite3.IntegrityError: FOREIGN KEY constraint failed

```

Напомним, что в таблице `Employees` базы данных `employees.db` столбец `DepartmentID` является внешним ключом, который ссылается на столбец `DepartmentID` в таблице `Departments`. Если мы изменим значение столбца `DepartmentID` в таблице `Employees`, то мы должны изменить его на значение, которое появляется в столбце `DepartmentID` в таблице `Departments`. В этом примере мы пытаемся изменить столбец `DepartmentID` Джека Смита на 99 в таблице `Employees`. Произошла ошибка, поскольку значение 99 не появляется в столбце `DepartmentID` таблицы `Departments`.

Удаление реляционных данных

Предположим, что таблица *A* имеет внешний ключ, который ссылается на столбец в таблице *B*. Вы не можете удалять любые строки в таблице *B*, на которые в настоящее время ссы-

ляются строки в таблице *A*. Это приведет к тому, что столбцы в таблице *A* будут ссылаться на несуществующие строки в таблице *B*.

Напомним, что в таблице Employees базы данных employees.db столбец LocationID является внешним ключом, который ссылается на столбец LocationID в таблице Locations. Таким образом, вы не можете удалить любые строки в таблице Locations, на которые в настоящее время ссылаются строки в таблице Employees. Предполагая, что мы подключились к базе данных employees.db и cur — это объект Cursor, давайте взглянем на следующий интерактивный сеанс:

```
>>> cur.execute('PRAGMA foreign_keys=ON')
<sqlite3.Cursor object at 0x00B5FD20>
>>> cur.execute('DELETE FROM Locations WHERE LocationID == 1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

В этом сеансе мы попытались удалить строку со значением 1, хранящуюся в столбце LocationID таблицы Locations. Произошла ошибка, поскольку в настоящее время в таблице Employees имеются строки, ссылающиеся на эту строку.

Извлечение столбцов из нескольких таблиц в инструкции *SELECT*

Когда связанные данные хранятся в нескольких таблицах, как в базе данных employees.db, часто необходимо извлекать данные из нескольких таблиц в одной инструкции SELECT. Например, предположим, мы хотим распечатать список, в котором указаны имя, отдел и местоположение каждого сотрудника. Это предусматривает наличие столбцов из таблицы Employees, таблицы Departments и таблицы Locations. В инструкции SELECT нам нужно указать не только имена столбцов, которые мы хотим получить, но и имена таблиц, к которым принадлежат эти столбцы. Для этого мы будем использовать *квалифицированные имена столбцов*. Полное имя столбца принимает следующий формат:

ИмяТаблицы.ИмяСтолбца

Например, Employees.DepartmentID указывает столбец DepartmentID в таблице Employees и Departments.DepartmentID указывает столбец DepartmentID в таблице Departments. Взглядите на следующий ниже запрос:

```
SELECT
    Employees.Name,
    Departments.DepartmentName,
    Locations.City
FROM
    Employees, Departments, Locations
WHERE
    Employees.DepartmentID == Departments.DepartmentID AND
    Employees.LocationID == Locations.LocationID
```

Первая часть запроса указывает столбцы, которые мы хотим получить:

```
SELECT
    Employees.Name,
```

```
Departments.DepartmentName,  
Locations.City
```

Вторая часть запроса, в которой используется выражение `FROM`, определяет таблицы, из которых мы хотим извлечь данные:

```
FROM  
Employees, Departments, Locations
```

Третья часть запроса, в которой используется выражение `WHERE`, определяет критерии поиска:

```
WHERE  
Employees.DepartmentID == Departments.DepartmentID AND  
Employees.LocationID == Locations.LocationID
```

Программа 14.18 демонстрирует этот запрос. Она выводит на экран имя, отдел и местоположение каждого сотрудника.

Программа 14.18 (print_employee_dept_city.py)

```
1 import sqlite3  
2  
3 def main():  
4     conn = None  
5     try:  
6         # Подсоединиться к базе данных и получить курсор.  
7         conn = sqlite3.connect('employees.db')  
8         cur = conn.cursor()  
9  
10        # Задействовать поддержку внешнего ключа.  
11        cur.execute('PRAGMA foreign_keys=ON')  
12  
13        # Извлечь имена сотрудников, отделы и города.  
14        cur.execute(  
15            """SELECT  
16                Employees.Name,  
17                Departments.DepartmentName,  
18                Locations.LocationName  
19                FROM  
20                Employees, Departments, Locations  
21                WHERE  
22                    Employees.DepartmentID == Departments.DepartmentID AND  
23                    Employees.LocationID == Locations.LocationID""")  
24        results = cur.fetchall()  
25        for row in results:  
26            print(f'{row[0]:15} {row[1]:25} {row[2]}')  
27    except sqlite3.Error as err:  
28        # Если произошло исключение, то напечатать сообщение об ошибке.  
29        print(err)
```

```

30     finally:
31         # Если соединение открыто, то закрыть его.
32         if conn != None:
33             conn.close()
34
35 # Вызвать главную функцию.
36 if __name__ == '__main__':
37     main()

```

Вывод программы

Арлин Мейерс	Исследования и разработки	Сан-Хосе
Джанель Грант	Производство	Остин
Джек Смит	Маркетинг	Нью-Йорк
Соня Альварадо	Бухгалтерский учет	Бостон
Рене Кинкейд	Маркетинг	Нью-Йорк
Курт Грин	Производство	Остин
Анжела Тейлор	Исследования и разработки	Сан-Хосе



ВНИМАНИЕ!

При соединении данных из нескольких таблиц следует использовать выражение `WHERE`, чтобы указать критерии поиска, связывающие соответствующие столбцы. Невыполнение этого требования может привести к большому набору не связанных между собой данных.

В ЦЕНТРЕ ВНИМАНИЯ

Приложение с GUI для чтения базы данных



Программа 14.19 — приложение с GUI, которое обращается к базе данных `employees.db`. Когда программа запускается, она выводит на экран список имен сотрудников в прокручиваемом виджете `Listbox` (рис. 14.10). Когда пользователь нажимает на имя в списке, информация о сотруднике выводится в диалоговом окне (рис. 14.11).

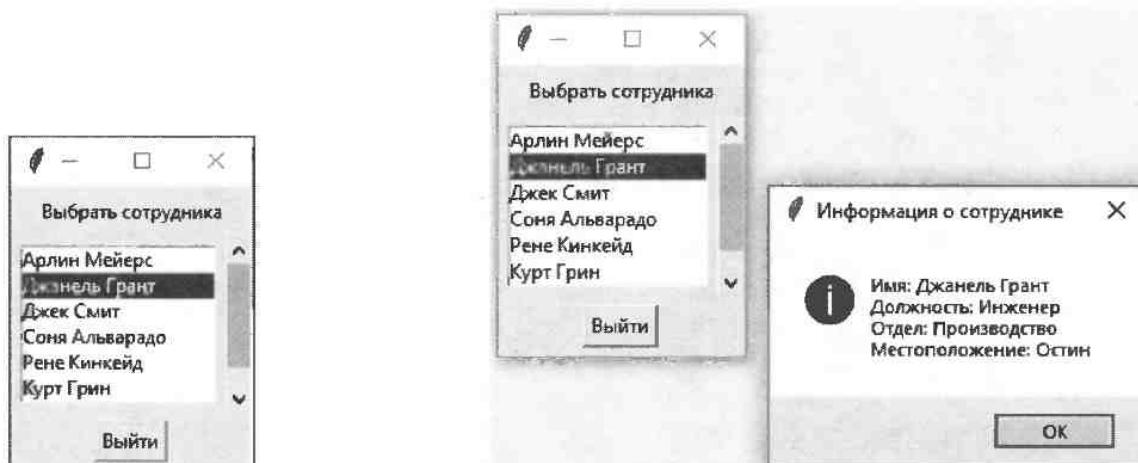


РИС. 14.10. Программа `employee_details`

РИС. 14.11. На экран выводятся сведения о сотруднике

Программа 14.19 (employee_details.py)

```
1 import tkinter
2 import tkinter.messagebox
3 import sqlite3
4
5 class EmployeeDetails:
6     def __init__(self):
7         # Создать главное окно.
8         self.main_window = tkinter.Tk()
9
10        # Скомпоновать содержимое главного окна.
11        self.__build_main_window()
12
13        # Запустить главный цикл.
14        tkinter.mainloop()
15
16        # Скомпоновать главное окно.
17    def __build_main_window(self):
18        # Создать надпись с подсказкой для пользователя.
19        self.__create_prompt_label()
20
21        # Скомпоновать рамку виджета Listbox.
22        self.__build_listbox_frame()
23
24        # Создать кнопку Выйти.
25        self.__create_quit_button()
26
27        # Создать надпись с подсказкой для пользователя.
28    def __create_prompt_label(self):
29        self.employee_prompt_label = tkinter.Label(
30            self.main_window, text='Выберите сотрудника')
31        self.employee_prompt_label.pack(side='top', padx=5, pady=5)
32
33        # Скомпоновать рамку, содержащую виджеты Listbox и Scrollbar
34    def __build_listbox_frame(self):
35        # Создать рамку для виджетов Listbox и Scrollbar.
36        self.listbox_frame = tkinter.Frame(self.main_window)
37
38        # Настроить виджет Listbox.
39        self.__setup_listbox()
40
41        # Создать полосу прокрутки для просмотра элементов в виджете Listbox.
42        self.__create_scrollbar()
43
44        # Заполнить виджет Listbox именами сотрудников.
45        self.__populate_listbox()
46
```

```
47     # Упаковать рамку виджета Listbox.
48     self.listbox_frame.pack()
49
50     # Создать виджет Listbox для вывода имен сотрудников на экран.
51     def __setup_listbox(self):
52         # Создать виджет Listbox.
53         self.employee_listbox = tkinter.Listbox(
54             self.listbox_frame, selectmode=tkinter.SINGLE, height=6)
55
56         # Привязать виджет Listbox к функции обратного вызова.
57         self.employee_listbox.bind(
58             '<<ListboxSelect>>', self.__get_details)
59
60         # Упаковать виджет Listbox.
61         self.employee_listbox.pack(side='left', padx=5, pady=5)
62
63     # Создать вертикальный виджет Scrollbar для использования с виджетом Listbox.
64     def __create_scrollbar(self):
65         self.scrollbar = tkinter.Scrollbar(self.listbox_frame,
66                                         orient=tkinter.VERTICAL)
67         self.scrollbar.config(command=self.employee_listbox.yview)
68         self.employee_listbox.config(yscrollcommand=self.scrollbar.set)
69         self.scrollbar.pack(side='right', fill=tkinter.Y)
70
71     # Вывести на экран имена сотрудников в виджете Listbox.
72     def __populate_listbox(self):
73         for employee in self.__get_employees():
74             self.employee_listbox.insert(tkinter.END, employee)
75
76     # Создать кнопку выхода из программы.
77     def __create_quit_button(self):
78         self.quit_button = tkinter.Button(
79             self.main_window,
80             text='Выход',
81             command=self.main_window.destroy)
82         self.quit_button.pack(side='top', padx=10, pady=5)
83
84     # Получить список имен сотрудников из базы данных.
85     def __get_employees(self):
86         employee_list = []
87         conn = None
88         try:
89             # Подсоединиться к базе данных и получить курсор.
90             conn = sqlite3.connect('employees.db')
91             cur = conn.cursor()
92
```

```
93         # Исполнить запрос SELECT.
94         cur.execute('SELECT Name FROM Employees')
95
96         # Получить результаты запроса в виде списка.
97         employee_list = [n[0] for n in cur.fetchall()]
98     except sqlite3.Error as err:
99         tkinter.messagebox.showinfo('Ошибка базы данных', err)
100    finally:
101        # Если соединение открыто, то закрыть его.
102        if conn != None:
103            conn.close()
104
105    return employee_list
106
107 # Получить подробную информацию по выбранному сотруднику.
108 def __get_details(self, event):
109     # Получить выбранное имя из виджета Listbox.
110     listbox_index = self.employee_listbox.curselection()[0]
111     selected_emp = self.employee_listbox.get(listbox_index)
112
113     # Запросить в базе данных информацию о выбранном сотруднике.
114     conn = None
115     try:
116         # Подсоединиться к базе данных и получить курсор.
117         conn = sqlite3.connect('employees.db')
118         cur = conn.cursor()
119
120         # Исполнить запрос SELECT.
121         cur.execute(
122             """SELECT
123                 Employees.Name,
124                 Employees.Position,
125                 Departments.DepartmentName,
126                 Locations.City
127             FROM
128                 Employees, Departments, Locations
129             WHERE
130                 Employees.Name == ? AND
131                 Employees.DepartmentID == Departments.DepartmentID AND
132                 Employees.LocationID == Locations.LocationID""",
133             (selected_emp,))
134
135         # Получить результаты запроса.
136         results = cur.fetchone()
137
```

```

138         # Вывести на экран информацию о сотруднике.
139         self._display_details(name=results[0], position=results[1],
140                               department=results[2], location=results[3])
141     except sqlite3.Error as err:
142         tkinter.messagebox.showinfo('Ошибка базы данных', err)
143     finally:
144         # Если соединение открыто, то закрыть его.
145         if conn != None:
146             conn.close()
147
148     # Вывести в диалоговом окне на экран информацию о сотруднике.
149     def __display_details(self, name, position, department, location):
150         tkinter.messagebox.showinfo('Информация о сотруднике',
151                                     'Имя: ' + name +
152                                     '\nДолжность: ' + position +
153                                     '\nОтдел: ' + department +
154                                     '\nМестоположение: ' + location)
155
156 # Создать экземпляр класса EmployeeDetails.
157 if __name__ == '__main__':
158     employee_details = EmployeeDetails()

```

Давайте рассмотрим класс EmployeeDetails подробнее.

Строки 6–14, метод `__init__` создает главное окно и запускает функцию `mainloop` модуля `tkinter`.

Строки 17–25, метод `__build_main_window` вызывает три других метода: `__create_prompt_label`, `__build_listbox_frame` и `__create_quit_button`. А они, в свою очередь, создают виджеты, которые будут выводиться на экран в главном окне.

Строки 28–31, метод `__create_prompt_label` создает виджет `Label`, который выводит на экран текст 'Выберите сотрудника'. Указанный метод вызывается из метода `__build_main_window`.

Строки 34–48, метод `__build_listbox_frame` создает рамку и вызывает методы `__setup_listbox` и `__create_scrollbar` для размещения виджетов `Listbox` и `Scrollbar` внутри рамки. Затем он вызывает метод `__populate_listbox`, чтобы прочитать все имена сотрудников из базы данных `employees.db` и вывести их в виджете `Listbox`. Указанный метод вызывается из метода `__build_main_window`.

Строки 51–61, метод `__setup_listbox` создает виджет `Listbox` и привязывает к нему метод `__get_details` в качестве функции обратного вызова. Как следствие, метод `__get_details` будет вызываться всякий раз, когда пользователь нажимает на имя в виджете `Listbox`. Указанный метод вызывается из метода `__build_listbox_frame`.

Строки 64–69, метод `__create_scrollbar` создает виджет `Scrollbar`, ориентированный вертикально. Строки 67 и 68 выполняют необходимые настройки для присоединения виджета `Scrollbar` к виджету `Listbox`. Указанный метод вызывается из метода `__build_listbox_frame`.

Строки 72–74, метод `__populate_listbox` вызывает метод `__get_employees`, чтобы получить список всех имен сотрудников из базы данных. Цикл `for` перебирает список и добавляет каждое имя в виджет `Listbox`. Указанный метод вызывается из метода `__build_listbox_frame`.

Строки 77–82, метод `__create_quit_button` создает виджет `Button`, который завершает работу программы при нажатии кнопки. Указанный метод вызывается из метода `__build_main_window`.

Строки 85–105, метод `__get_employees` открывает соединение с базой данных `employees.db` и выполняет инструкцию с запросом `SELECT`, который получает все имена сотрудников из таблицы `Employees`. Имена возвращаются из метода в виде списка. Указанный метод вызывается из метода `__populate_listbox`.

Строки 108–146, метод `__get_details` получает имя, выбранное в виджете `Listbox`. Затем он открывает соединение с базой данных `employees.db` и выполняет запрос `SELECT`, который получает из базы данных имя, должность, название отдела и местоположение выбранного сотрудника. Эти порции данных передаются в метод `__display_details` для вывода на экран. Указанный метод является функцией обратного вызова для виджета `Listbox`, поэтому он вызывается всякий раз, когда пользователь выбирает имя из виджета `Listbox`.

Строки 149–154, метод `__display_details` принимает в качестве аргументов имя сотрудника, должность, название отдела и местоположение. Эти значения выводятся в диалоговом окне.



Контрольная точка

- 14.41. Почему возникает потребность в уменьшении дублирования данных в базе данных?
- 14.42. Что такое внешний ключ?
- 14.43. Что такое связь "один ко многим"?
- 14.44. Что такое связь "многие к одному"?

Вопросы для повторения

Множественный выбор

1. Стандартный язык для работы с СУБД — это _____.
 - a) Python;
 - б) COBOL;
 - в) SQL;
 - г) BASIC.
2. Хранящиеся в таблице данные организованы в _____.
 - а) строки;
 - б) файлы;
 - в) папки;
 - г) страницы.

3. Хранящиеся в строке данные делятся на _____.
- разделы;
 - байты;
 - столбцы;
 - таблицы.
4. _____ — это столбец, который содержит уникальное значение для каждой строки и может использоваться для идентификации конкретных строк.
- идентификационный столбец;
 - открытый ключ;
 - столбец обозначителя;
 - первичный ключ.
5. Значение `None` в Python эквивалентно значению _____ в SQL.
- 0;
 - 1;
 - `NULL`;
 - отрицательная бесконечность.
6. _____ содержит уникальные значения, генерируемые СУБД.
- самоопределяющийся столбец;
 - идентификационный столбец;
 - столбец серийного номера;
 - столбец хеш-значения.
7. Если столбец _____, то это означает, что всякий раз, когда в таблицу добавляется новая строка, СУБД автоматически присваивает столбцу целое число, которое на 1 больше наибольшего значения, хранящегося в данный момент в столбце.
- автоматически наращивается (автоинкрементируется);
 - переворачивается;
 - бутстрапируется;
 - автоматически дополняется.
8. Функция `sqlite3.connect` возвращает этот тип объекта.
- `Cursor`;
 - `Database`;
 - `File`;
 - `Connection`.
9. Вы используете метод _____ объекта `Cursor` для передачи инструкции SQL в СУБД SQLite.
- `commit()`;
 - `execute()`;

- б) pass();
г) run_sql().
10. Эта инструкция SQL используется для создания таблицы.
а) CREATE TABLE;
б) ADD TABLE;
в) INSERT TABLE;
г) NEW TABLE.
11. Эта инструкция SQL используется для удаления таблицы.
а) DELETE TABLE;
б) DROP TABLE;
в) ERASE TABLE;
г) REMOVE TABLE.
12. Эта инструкция SQL используется для вставки строк в таблицу.
а) INSERT;
б) ADD;
в) CREATE;
г) UPDATE.
13. Эта инструкция SQL используется для удаления строк из таблицы.
а) REMOVE;
б) ERASE;
в) PURGE;
г) DELETE.
14. Эта инструкция SQL используется для изменения содержимого строки.
а) EDIT;
б) UPDATE;
в) CHANGE;
г) TRANSFORM.
15. Этот тип инструкции SQL используется для извлечения строк из таблицы.
а) RETRIEVE;
б) GET;
в) SELECT;
г) READ.
16. Это выражение позволяет указывать критерии поиска в инструкции SELECT.
а) SEARCH;
б) WHERE;

- в) AS;
г) CRITERIA.
17. Этот метод объекта Cursor возвращает все результаты ранее исполненной инструкции SELECT в виде списка кортежей.
- а) get_results;
б) getall;
в) fetchone;
г) fetchall.
18. Этот метод объекта Cursor возвращает только одну строку результатов ранее исполненной инструкции SELECT в виде кортежа.
- а) get_results;
б) getall;
в) fetchone;
г) fetchall.
19. Это выражение позволяет сортировать результаты инструкции SELECT.
- а) SORT BY;
б) ARRANGE;
в) ORDER BY;
г) DESCEND.
20. Эта функция SQL возвращает среднее значение столбца, содержащего числовые значения.
- а) AVERAGE;
б) MEAN;
в) MEDIAN;
г) AVG.
21. Эта функция SQL возвращает сумму по столбцу, содержащему числовые значения.
- а) SUM;
б) TOTAL;
в) ADD;
г) ALL.
22. Эта функция SQL возвращает наименьшее, или минимальное, значение, находящееся в столбце, содержащем числовые значения.
- а) LEAST;
б) SMALLEST;
в) MIN;
г) MINIMUM.

23. Эта функция SQL возвращает наибольшее, или максимальное, значение, находящееся в столбце, содержащем числовые значения.
- а) GREATEST;
 - б) LARGEST;
 - в) MAXIMUM;
 - г) MAX.
24. Эта функция SQL возвращает число строк в таблице или число строк, соответствующих критериям поиска.
- а) COUNT;
 - б) NUM_ROWS;
 - в) QUANTITY;
 - г) TOTAL.
25. Этот публичный атрибут объекта Cursor содержит число строк, которые были изменены последней исполненной инструкцией SQL.
- а) rows;
 - б) num_rows;
 - в) altered_rows;
 - г) rowcount.
26. При создании таблицы в SQLite таблица автоматически будет иметь столбец с типом INTEGER под названием _____.
- а) RowNumber;
 - б) RowID;
 - в) ID;
 - г)RowIndex.
27. _____ ключ — это два столбца или более, которые совмещаются для создания уникального значения.
- а) комбинированный;
 - б) составной;
 - в) композитный;
 - г) соединенный.
28. Модуль sqlite3 определяет исключение с именем _____, которое вызывается при возникновении ошибки базы данных.
- а) DBException;
 - б) DataError;
 - в) SQLEException;
 - г) Error.

29. _____ — это столбец в одной таблице, который ссылается на первичный ключ в другой таблице.
- вторичный ключ;
 - поддельный ключ;
 - внешний ключ;
 - дублированный ключ.

Истина или ложь

- Для хранения крупных объемов данных СУБД предпочтительнее традиционного файла.
- Все идентификационные столбцы в таблице должны содержать одно и то же значение.
- Таблица может иметь более одного первичного ключа.
- Если при подсоединении к базе данных SQLite она не существует, то будет выдано исключение.
- При внесении изменений в базу данных SQLite эти изменения не сохраняются в базе данных до тех пор, пока не будет вызван метод `commit()` объекта `Connection`.
- В SQLite метод `execute()` объекта `Cursor` используется для передачи инструкции SQL в СУБД.
- База данных может содержать только одну таблицу.
- С помощью инструкции SQL `UPDATE` можно обновлять более одной строки.
- При создании таблицы в SQLite столбец `RowID` не создается автоматически.
- В SQLite при назначении столбца в качестве целочисленного первичного ключа (`INTEGER PRIMARY KEY`) этот столбец становится псевдонимом для столбца `RowID`.

Короткий ответ

- Почему при написании приложений для крупного бизнеса по хранению записей о клиентах и ведению учета товарных запасов не используются традиционные текстовые или двоичные файлы?
- Когда мы говорим об организации базы данных, мы говорим о таких вещах, как строки, таблицы и столбцы. Опишите, каким образом данные в базе данных организованы в эти концептуальные единицы.
- Что такое первичный ключ?
- Какие типы данных SQL соответствуют следующим ниже типам Python:
 - `int`;
 - `float`;
 - `str`?
- Каковы реляционные операторы в SQL для следующих ниже сравнений:
 - больше;
 - меньше;
 - больше или равно;
 - меньше или равно;

- равно;
- не равно?

6. Посмотрите на следующую ниже инструкцию SQL.

```
SELECT Name FROM Employee
```

Как называется таблица, из которой эта инструкция извлекает данные? Каково имя извлекаемого столбца?

7. Что такое параметризованный запрос?
8. В чем разница между методами `fetchall()` и `fetchone()` объекта `Cursor` в SQLite?
9. Что такое агрегатная функция?
10. Как при обновлении таблицы в SQLite определить число обновленных строк?
11. Что такое составной ключ?
12. Предположим, что в базе данных SQLite таблица содержит три строки и столбец `RowID` содержит значения 1, 2 и 3. Вы добавляете еще одну строку в таблицу, не указывая значение для ее столбца `RowID`. Какое значение СУБД автоматически присвоит новой строке в столбце `RowID`?
13. Предположим, что в базе данных SQLite таблица содержит четыре строки и столбец `RowID` содержит значения 1, 2, 3 и 19. Вы добавляете еще одну строку в таблицу, не указывая значение для ее столбца `RowID`. Какое значение СУБД автоматически присвоит новой строке в столбце `RowID`?
14. Псевдонимом какого столбца становится целочисленный первичный ключ (`INTEGER PRIMARY KEY`) в таблице в SQLite при его создании?
15. Что означает CRUD?
16. Что такое внешний ключ?

Алгоритмический тренажер

Для следующих далее вопросов предположим, что база данных SQLite имеет таблицу с именем `Stock` (Акции) (табл. 14.4).

Таблица 14.4

Имя столбца	Тип
<code>TradingSymbol</code> (Торговый символ)	TEXT
<code>CompanyName</code> (Название компании)	TEXT
<code>NumShares</code> (Число долей)	INTEGER
<code>PurchasePrice</code> (Цена покупки)	REAL
<code>SellingPrice</code> (Цена продажи)	REAL

1. Напишите SQL-инструкцию `SELECT`, которая вернет все столбцы из каждой строки в таблице `Stock`.
2. Напишите SQL-инструкцию `SELECT`, которая вернет столбец `TradingSymbol` из каждой строки в таблице `Stock`.

3. Напишите SQL-инструкцию SELECT, которая вернет столбец `TradingSymbol` и столбец `NumShares` из каждой строки таблицы `Stock`.
4. Напишите SQL-инструкцию SELECT, которая вернет столбец `TradingSymbol` только из тех строк, в которых значение `PurchasePrice` превышает 25.00.
5. Напишите SQL-инструкцию SELECT, которая вернет все столбцы из строк, в которых `TradingSymbol` начинается с "SU".
6. Напишите SQL-инструкцию SELECT, которая вернет столбец `TradingSymbol` только из тех строк, в которых `SellingPrice` больше, чем `PurchasePrice`, а `NumShares` больше 100.
7. Напишите SQL-инструкцию SELECT, которая вернет столбец `TradingSymbol` и столбец `NumShares` только из тех строк, в которых `SellingPrice` больше `PurchasePrice`, а `NumShares` превышает 100. Результаты должны быть отсортированы по столбцу `NumShares` в порядке возрастания.
8. Напишите SQL-инструкцию, которая вставит новую строку в таблицу `Stock`. Страна должна иметь следующие значения столбцов:

`TradingSymbol: XYZ`

`CompanyName: Компания XYZ`

`NumShares: 150`

`PurchasePrice: 12.55`

`SellingPrice: 22.47`

9. Напишите SQL-инструкцию, которая выполнит следующее: для каждой строки в таблице `Stock`, если столбец `TradingSymbol` имеет значение "XYZ", заменяет это значение на "ABC".
10. Напишите SQL-инструкцию, которая удалит строки в таблице `Stock`, в которых число акций меньше 10.
11. Напишите SQL-инструкцию, которая создаст таблицу `Stock`, если она не существует.
12. Напишите SQL-инструкцию, которая удалит таблицу `Stock`, если она существует.

Упражнения по программированию

1. **База данных населения.** В папке с исходным кодом этой главы вы найдете программу `create_cities_db.py`. Запустите программу. Она создаст базу данных с именем `cities.db`. База данных `cities.db` будет иметь таблицу с именем `Cities` (табл. 14.5).

Таблица 14.5

Имя столбца	Тип данных
<code>CityID</code>	<code>INTEGER PRIMARY KEY</code>
<code>CityName</code>	<code>TEXT</code>
<code>Population</code>	<code>REAL</code>



Видеозапись "Начало работы с базой данных `Population`" (*Getting Started with the Population Database Problem*)

В столбце `CityName` хранится название города, а в столбце `Population` — численность населения этого города. После запуска программы `create_cities_db.py` таблица городов будет содержать 20 строк с различными городами и их численностью населения.

Далее напишите программу, которая подсоединяется к базе данных cities.db и позволяет пользователю выбирать любую из следующих ниже операций:

- вывод на экран списка городов, отсортированных по численности населения в порядке возрастания;
 - вывод на экран списка городов, отсортированных по численности населения в порядке убывания;
 - вывод на экран списка городов, отсортированных по названиям;
 - вывод на экран общей численности населения всех городов;
 - вывод на экран среднего населения всех городов;
 - вывод на экран города с наибольшей численностью населения;
 - вывод на экран города с наименьшей численностью населения.
2. **Телефонная база данных.** Напишите программу, которая создает базу данных с именем phonebook.db. В базе данных должна быть таблица Entries со столбцами для имени и номера телефона человека. Далее напишите приложение CRUD, которое позволяет пользователю добавлять строки в таблицу Entries, отыскивать номер телефона человека, изменять его номер телефона и удалять заданные строки.
3. **Проект реляционной базы данных.** В этом задании вы создадите базу данных с именем student_info.db, содержащую следующую информацию о студентах в колледже:
- фамилию и имя студента;
 - специальность студента;
 - кафедру, на которую зачислен студент.

База данных должна содержать таблицы Majors (Специальности), Departments (Факультеты), Students (Студенты), табл. 14.6–14.8.

Таблица 14.6. Таблица Majors (Специальности)

Имя столбца	Тип
MajorID	INTEGER PRIMARY KEY
Name	TEXT

Таблица 14.7. Таблица Departments (Факультеты)

Имя столбца	Тип
DeptID	INTEGER PRIMARY KEY
Name	TEXT

Таблица 14.8. Таблица Students (Студенты)

Имя столбца	Тип
StudentID	INTEGER PRIMARY KEY
Name	TEXT
MajorID	INTEGER (внешний ключ, ссылающийся на столбец MajorID в таблице Majors)
DeptID	INTEGER (внешний ключ, ссылающийся на столбец DeptID в таблице Departments)

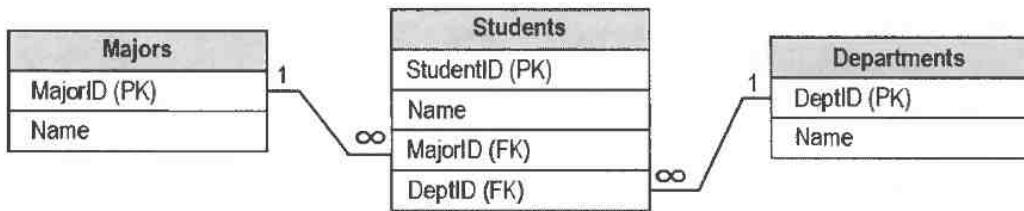


РИС. 14.12. Диаграмма связей между сущностями базы данных student_info.db

На рис. 14.12 показана диаграмма связей между сущностями базы данных.

- Напишите программу, которая создает базу данных и таблицы.
- Напишите программу, которая выполняет операции CRUD над таблицей Majors. В частности, программа должна позволять пользователю выполнять следующие действия:
 - добавлять новую специальность;
 - отыскивать существующую специальность;
 - обновлять существующую специальность;
 - удалять существующую специальность;
 - выводить на экран список всех специальностей.
- Напишите программу, которая выполняет операции CRUD над таблицей Departments. В частности, программа должна позволять пользователю выполнять следующие действия:
 - добавлять новый факультет;
 - отыскивать существующий факультет;
 - обновлять существующий факультет;
 - удалять существующий факультет;
 - выводить на экран список всех факультетов.
- Напишите программу, которая выполняет операции CRUD над таблицей Students. В частности, программа должна позволять пользователю выполнять следующие действия:
 - добавлять нового студента;
 - отыскивать существующего студента;
 - обновлять существующего студента;
 - удалять существующего студента;
 - выводить на экран список всех студентов.

При добавлении, обновлении и удалении строк следует активировать поддержку внешних ключей. При добавлении нового студента в таблицу Students пользователю должно быть разрешено выбирать только существующую специальность из таблицы Majors и существующий факультет из таблицы Departments.

ПРИЛОЖЕНИЯ

Скачивание Python

Для того чтобы выполнить приведенные в этой книге программы, вам потребуется установить Python 3.6 или более позднюю версию. Вы можете скачать последнюю версию Python по адресу www.python.org/downloads. В этом приложении рассматривается процесс установки Python в операционной системе Windows. Python также доступен для Mac OS X, Linux и некоторых других платформ. Ссылки на скачивание версий Python, предназначенных для этих операционных систем, приведены на сайте для скачивания Python по адресу www.python.org/downloads.



СОВЕТ

Следует иметь в виду, что существует два семейства языка Python, которые вы можете скачать: Python 3.x и Python 2.x. Программы в этой книге работают только с семейством Python 3.x.

Установка Python 3.x в Windows

Перейдя по ссылке www.python.org/downloads на сайт для скачивания Python, вы должны будете скачать последнюю из имеющихся версий Python 3.x. На рис. П1.1 показано, как выглядела веб-страница сайта, с которой можно скачать Python, в момент написания этого приложения. Как видно из рисунка, последней версией в это время был Python 3.9.5.

После того как вы скачаете установщик Python, его следует запустить. На рис. П1.2 представлен установщик Python 3.9.5. Настоятельно рекомендуем поставить флажки напротив обеих опций внизу экрана: **Install launcher for all users** (Установить средство запуска для всех пользователей) и **Add Python 3.x to PATH** (Добавить Python 3.x в системный путь). Сделав это, щелкните по ссылке **Install Now** (Установить сейчас).

Появится сообщение "Do you want to allow this app to make changes to your device?" (Хотите разрешить этому приложению внести изменения в ваше устройство?) Нажмите кнопку **Да**, чтобы продолжить установку. Когда процесс установки завершится, вы увидите сообщение "Installation was successful." (Установка завершилась успешно). Нажмите кнопку закрытия окна для выхода из установщика.

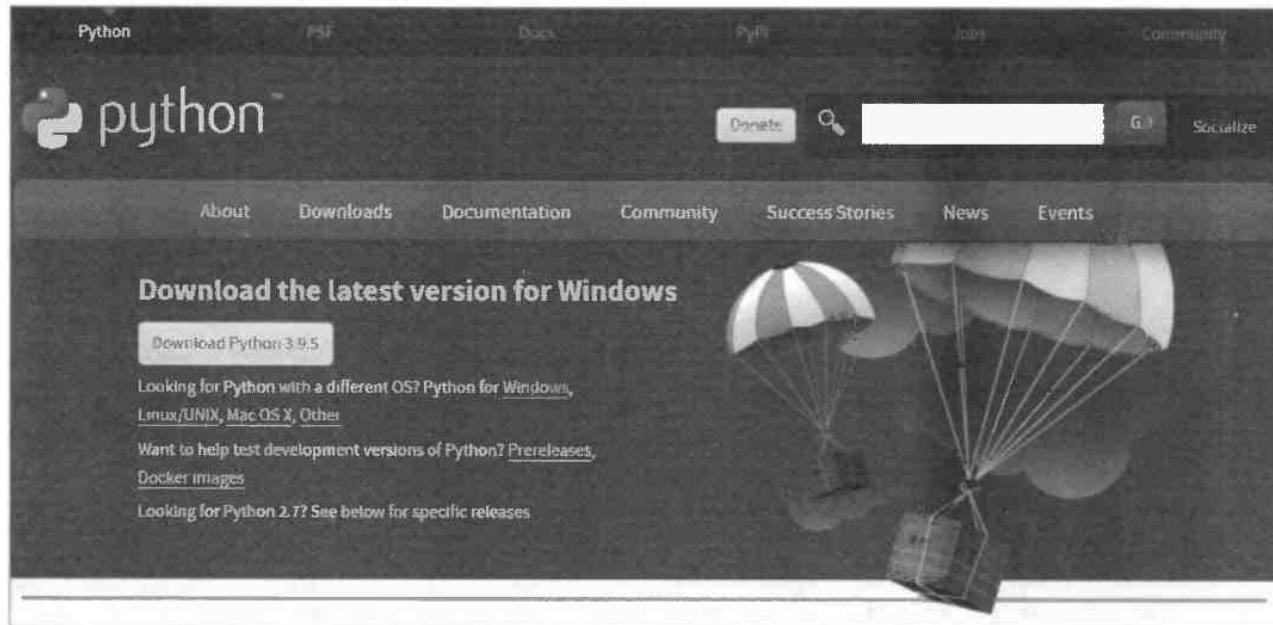


РИС. П1.1. Скачайте последнюю версию языка Python

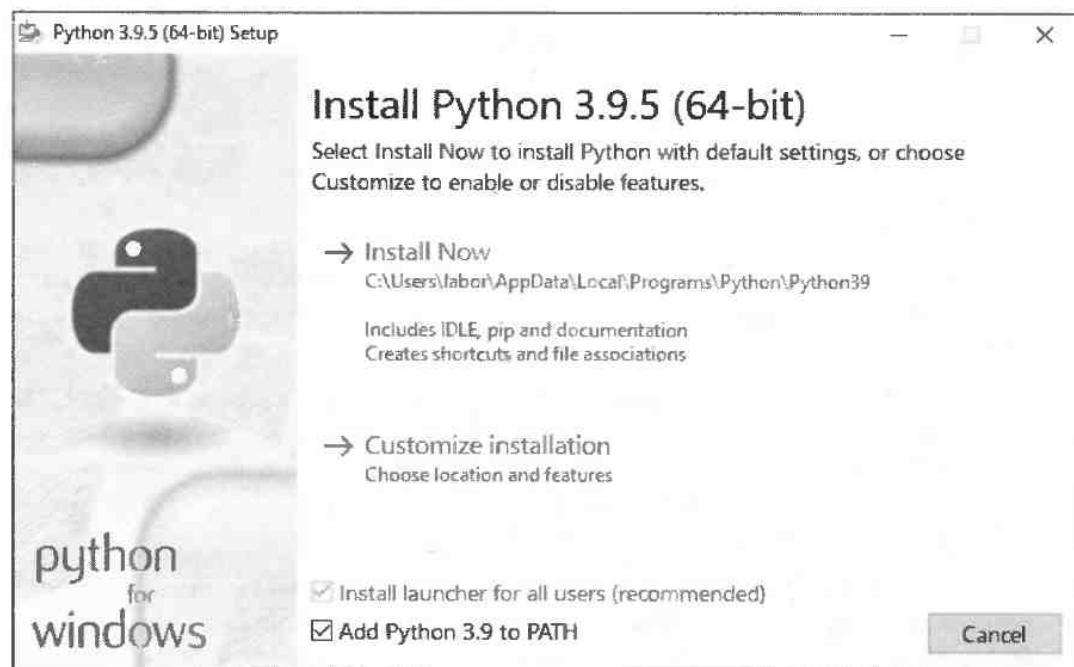


РИС. П1.2. Окно установки Python



Видеозапись "Введение в среду IDLE" (Intro to IDLE)

IDLE (Integrated Development Environment) — это интегрированная среда разработки, которая сочетает в одной программе несколько инструментов разработки, в том числе:

- ◆ оболочку Python, работающую в интерактивном режиме. Можно набирать инструкции Python напротив подсказки оболочки и сразу же их выполнять. Кроме того, можно выполнять законченные программы Python;
- ◆ текстовый редактор, который выделяет цветом ключевые слова Python и другие части программ;
- ◆ модуль проверки, который проверяет программу Python на наличие синтаксических ошибок без выполнения программы;
- ◆ средства поиска, позволяющие находить текст в одном или нескольких файлах;
- ◆ инструменты форматирования текста, которые помогают поддержать в программе Python единообразные уровни отступов;
- ◆ отладчик, который позволяет выполнять программу Python в пошаговом режиме и следить за изменением значений переменных по ходу исполнения каждой инструкции;
- ◆ несколько других продвинутых инструментов для разработчиков.

Среда IDLE поставляется в комплекте с Python. Во время установки интерпретатора Python среда IDLE устанавливается автоматически. В этом приложении предоставлено краткое введение в среду IDLE и описание основных шагов создания, сохранения и выполнения программы Python.

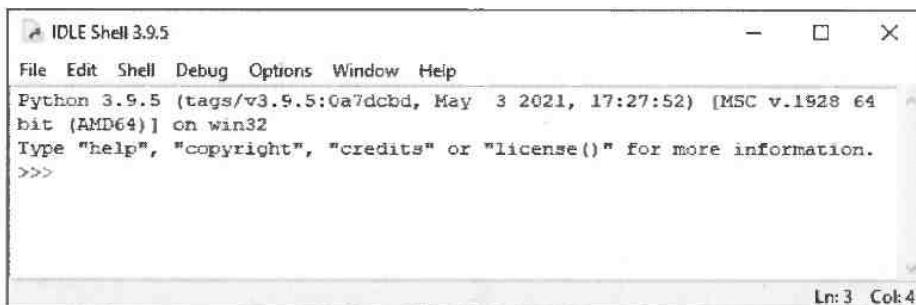
Запуск среды IDLE и использование оболочки Python

После того как вы установите Python, в списке программ меню Пуск вашей операционной системы появится группа программ Python. Один из элементов в этой группе программ будет называться **IDLE (Python 3.x 64(32)-bit)**. Для запуска IDLE щелкните по нему, и вы увидите окно оболочки Python 3.x (рис. П2.1). Внутри этого окна интерпретатор Python выполняется в интерактивном режиме. Вверху окна расположена строка меню, которая предоставляет доступ ко всем инструментам IDLE.

Подсказка `>>>` говорит о том, что интерпретатор ожидает от вас ввода инструкции Python. Когда вы набираете инструкцию напротив подсказки `>>>` и нажимаете клавишу `<Enter>`, эта инструкция немедленно исполняется. Например, на рис. П2.2 показано окно оболочки Python после того, как были введены и исполнены три инструкции.

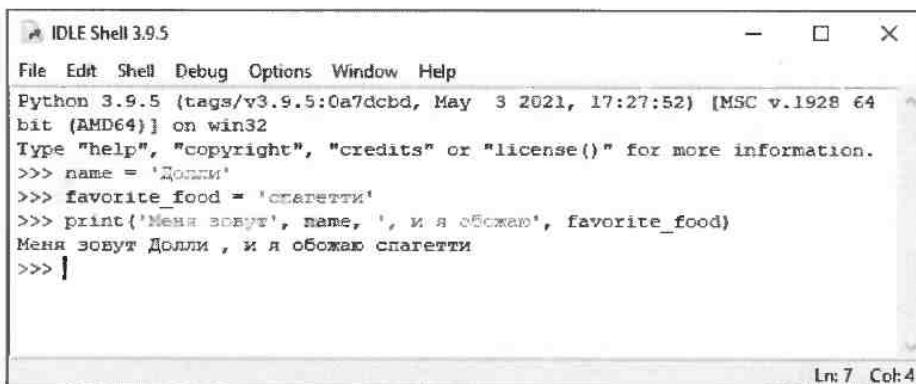
Когда вы набираете начало многострочной инструкции, в частности инструкцию `if` или цикл, каждая последующая строка автоматически располагается с отступом. Нажатие кла-

виши <Enter> на пустой строке обозначает конец многострочной инструкции и приводит к ее исполнению интерпретатором. На рис. П2.3 представлено окно оболочки Python после того, как цикл `for` был введен и выполнен.



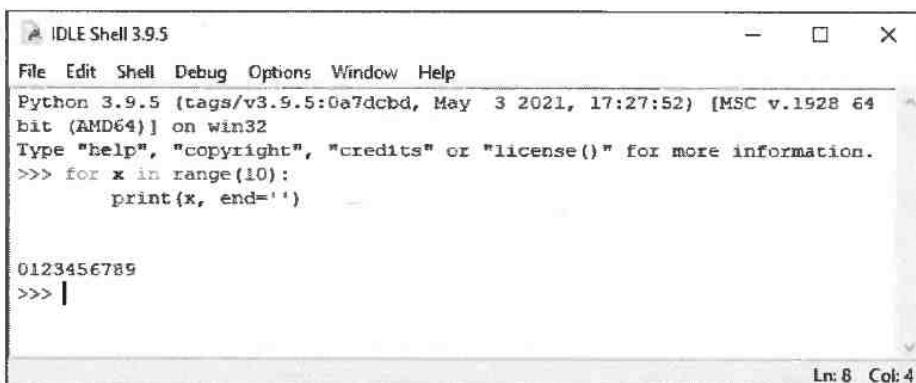
Снимок экрана окна оболочки IDLE. Видно меню (File, Edit, Shell, Debug, Options, Window, Help), информацию о Python (Python 3.9.5 (tags/v3.9.5:0a7dcbd, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32) и сообщение о том, что можно ввести "help", "copyright", "credits" или "license()". В окне введен текст >>>.

РИС. П2.1. Окно оболочки IDLE



Снимок экрана окна оболочки IDLE. Видно меню (File, Edit, Shell, Debug, Options, Window, Help), информацию о Python (Python 3.9.5 (tags/v3.9.5:0a7dcbd, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32) и сообщение о том, что можно ввести "help", "copyright", "credits" или "license()". В окне введены следующие инструкции: >>> name = 'Долли' >>> favorite_food = 'спагетти' >>> print('Меня зовут', name, ', и я обожаю', favorite_food) >>> Меня зовут Долли , и я обожаю спагетти >>> |

РИС. П2.2. Инструкции, исполненные интерпретатором Python



Снимок экрана окна оболочки IDLE. Видно меню (File, Edit, Shell, Debug, Options, Window, Help), информацию о Python (Python 3.9.5 (tags/v3.9.5:0a7dcbd, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32) и сообщение о том, что можно ввести "help", "copyright", "credits" или "license()". В окне введены инструкции: >>> for x in range(10): print(x, end=''). Вывод: 0123456789 >>> |

РИС. П2.3. Многострочная инструкция, исполненная интерпретатором Python

Написание программы Python в редакторе IDLE

Для того чтобы написать новую программу Python в среде IDLE, следует открыть новое окно редактирования. Для этого в меню **File** (Файл) нужно выбрать команду **New File** (Новый файл) — рис. П2.4. (Как вариант, можно нажать комбинацию клавиш **<Ctrl>+<N>**.) Будет открыто окно редактирования текста (рис. П2.5).

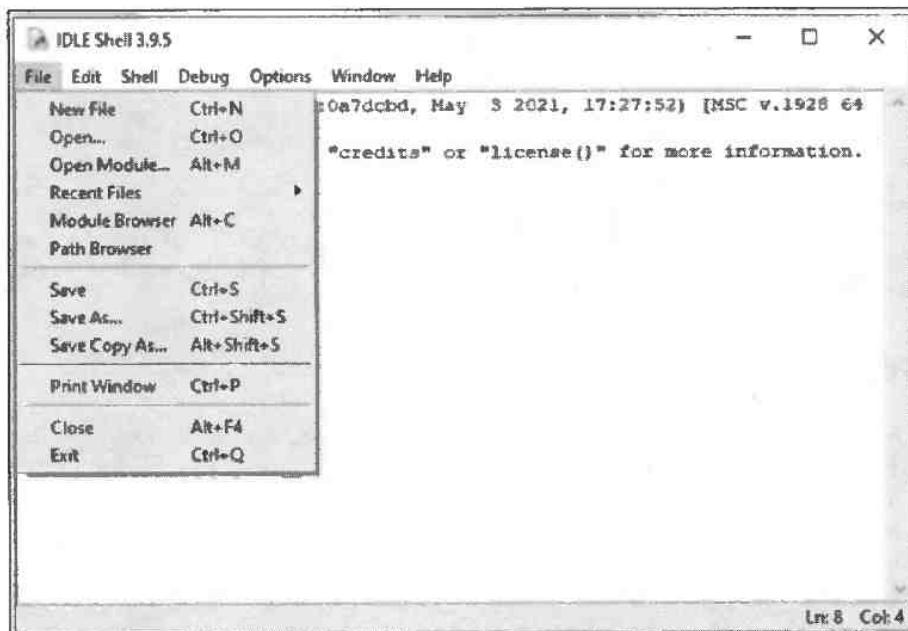


РИС. П2.4. Меню File

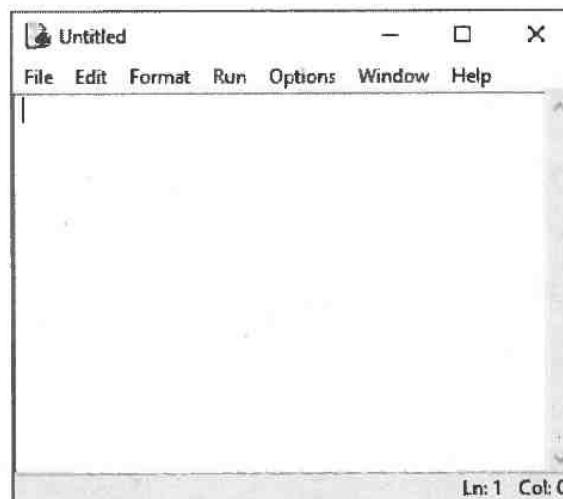


РИС. П2.5. Окно текстового редактора

Для того чтобы открыть уже существующую программу, в меню **File** (Файл) выберите команду **Open** (Открыть). Перейдите в место расположения файла, выберите нужный файл, и он будет открыт в окне редактора.

Цветовая разметка

Программный код, вводимый в окне редактора, а также в окне оболочки Python, выделяется цветом следующим образом:

- ◆ ключевые слова Python выводятся в оранжевом цвете;
- ◆ комментарии — в красном цвете;
- ◆ строковые литералы — в зеленом цвете;
- ◆ определенные в программе имена, такие как имена функций и классов, — в синем цвете;
- ◆ встроенные функции — в фиолетовом цвете.



СОВЕТ

Вы можете изменить настройки выделения цветом в среде IDLE, выбрав в меню **Options** (Параметры) команду **Configure IDLE** (Сконфигурировать среду IDLE). В появившемся диалоговом окне перейдите на вкладку **Highlights** (Выделение цветом) и задайте цвета для каждого элемента программы Python.

Автоматическое выделение отступом

Редактор IDLE имеет функциональные возможности, которые помогают поддерживать в своих программах Python единообразное выделение отступами. Пожалуй, самой полезной из этих функциональных возможностей является автоматическое выделение отступом. Когда вы набираете строку, которая заканчивается двоеточием, в частности выражение `if`, первую строку цикла или заголовок функции и затем нажимаете клавишу `<Enter>`, редактор автоматически выделяет отступом строки, которые вы набираете потом. Например, предположим, что вы набираете фрагмент кода (рис. П2.6). После того как вы нажмете клавишу `<Enter>` в конце строк, помеченной ①, редактор автоматически расположит с отступом строки, которые вы будете набирать далее. После того как вы нажмете клавишу `<Enter>` в конце строк, помеченной ②, редактор снова добавит отступ. Нажатие клавиши `<Backspace>` в начале выделенной отступом строк отменяет один уровень выделения отступом.

```

Untitled
File Edit Format Run Options Window Help
# Esta программа демонстрирует применение
# функции range вместе с циклом for
def main():
    # Намечатель сообщение пять раз
    for x in range(5):
        print('Hello, world!')

    # Вызвать главную функцию
main()

```

Ln: 11 Col: 0

РИС. П2.6. Строки, которые вызывают автоматическое добавление отступов

По умолчанию в среде IDLE для каждого уровня выделения отступом в качестве отступа используются *четыре пробела*. Количество пробелов можно изменить, выбрав в меню **Options** команду **Configure IDLE**. Перейдите на вкладку **Fonts/Tabs** (Шрифты/Отступы) появившегося диалогового окна, и вы увидите панель ползунка, который позволяет изменить количество пробелов, используемых в качестве ширины отступа. Однако поскольку в Python четыре пробела являются стандартной шириной отступа, рекомендуется все же оставить текущую настройку.

Сохранение программы

В окне редактора можно сохранить текущую программу, выбрав соответствующую команду из меню **File**:

- ◆ **Save (Сохранить);**
- ◆ **Save As (Сохранить как);**
- ◆ **Save Copy As (Сохранить копию как).**

Команды **Save** и **Save As** работают так же, как и в любом приложении Windows. Команда **Save Copy As** работает как **Save As**, но оставляет исходную программу в окне редактора.

Выполнение программы

После того как программа набрана в редакторе, ее можно выполнить, нажав клавишу **<F5>** или выбрав в меню **Run** (Выполнить) команду **Run Module** (Выполнить модуль) — рис. П2.7. Если после последнего внесения изменения в исходный код программа не была сохранена, появится диалоговое окно (рис. П2.8). Нажмите кнопку **OK**, чтобы сохранить программу. Во время выполнения программы вы увидите, что ее результаты будут выведены в окно оболочки Python IDLE (рис. П2.9).

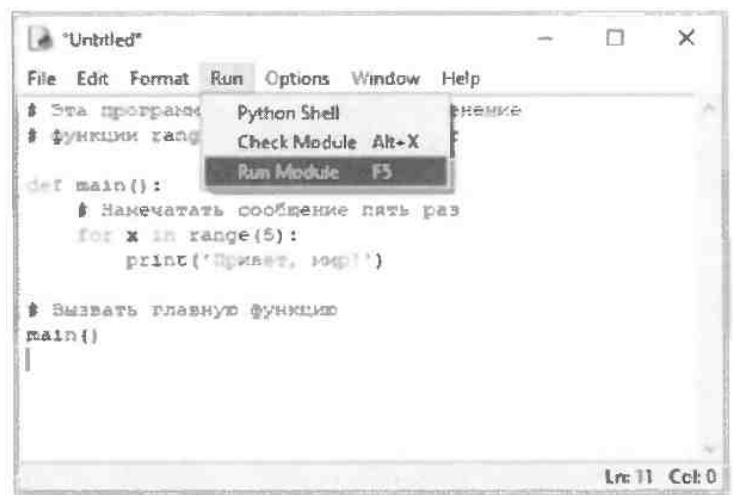


РИС. П2.7. Меню Run окна редактора

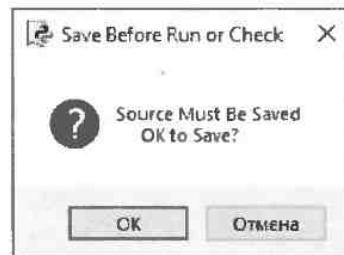
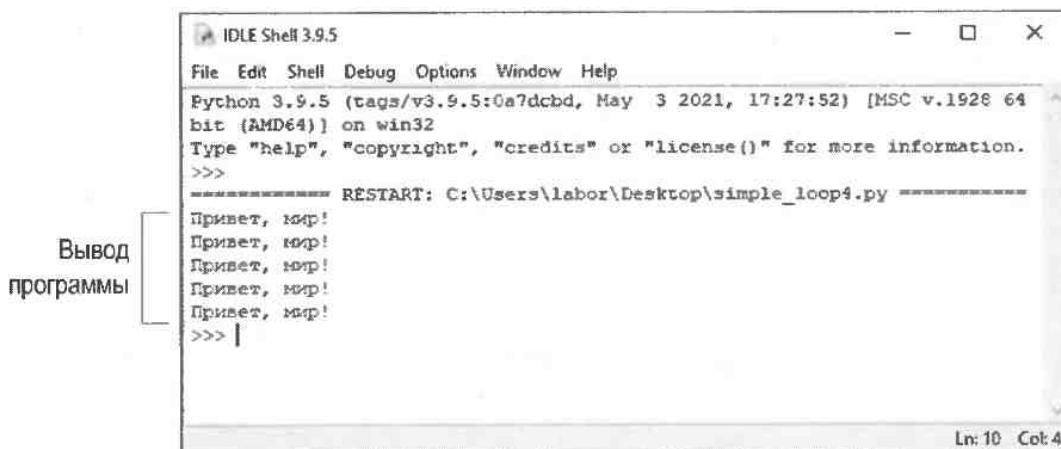


РИС. П2.8. Диалоговое окно подтверждения сохранения

При выполнении программы, содержащей синтаксическую ошибку, вы увидите диалоговое окно (рис. П2.10). После нажатия на кнопке **OK** редактор выделит цветом местоположение ошибки в программном коде. Если вы захотите проверить синтаксис программы, не пытаясь



```

IDLE Shell 3.9.5
File Edit Shell Debug Options Window Help
Python 3.9.5 (tags/v3.9.5:0a7dcbf, May 3 2021, 17:27:52) [MSC v.1928 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: C:\Users\labor\Desktop\simple_loop4.py =====
Привет, мир!
Привет, мир!
Привет, мир!
Привет, мир!
Привет, мир!
>>> |
Ln: 10 Col: 4

```

РИС. П2.9. Результаты в окне оболочки Python

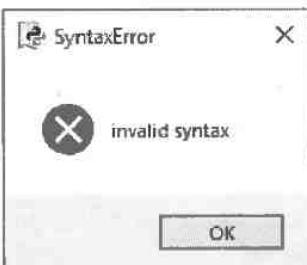


РИС. П2.10. Диалоговое окно с сообщением о синтаксической ошибке

ее выполнить, то в меню **Run** (Выполнить) выберите команду **Check Module** (Проверить модуль). В результате при обнаружении любой синтаксической ошибки будет выведено соответствующее сообщение.

Другие ресурсы

В этом приложении был предоставлен лишь краткий обзор применения среды IDLE для создания, хранения и выполнения программ. Среда IDLE предлагает множество других расширенных функциональных возможностей. Для того чтобы узнать о них, обратитесь к официальной документации IDLE по адресу www.python.org/idle.

В табл. П3.1 перечислен набор символов ASCII (American Standard Code for Information Interchange, стандартный американский код обмена информацией), который совпадает с первыми 127 кодами символов Юникода. Эта группа кодов называется *латинским подмножеством Юникода*. Столбцы "Код" показывают коды символов, столбцы "Символ" — соответствующие символы. Например, код 65 представляет английскую букву А. Обратите внимание, что коды от 0 до 31 и код 127 представляют управляющие символы, которые не отображаются на экране и при печати на принтере.

Таблица П3.1

Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	ESC	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32	(Пробел)	58	:	84	T	110	n
7	BEL	33	!	59	;	85	U	111	o
8	BS	34	"	60	<	86	V	112	p
9	HT	35	#	61	=	87	W	113	q
10	LF	36	\$	62	>	88	X	114	r
11	VT	37	%	63	?	89	Y	115	s
12	FF	38	&	64	@	90	Z	116	t
13	CR	39	'	65	А	91	[117	u
14	SO	40	(66	Б	92	\	118	v
15	SI	41)	67	С	93]	119	w
16	DLE	42	*	68	Д	94	^	120	x
17	DC1	43	+	69	Е	95	_	121	y
18	DC2	44	,	70	Ғ	96	‘	122	z
19	DC3	45	-	71	Ғ	97	а	123	{
20	DC4	46	.	72	Ҳ	98	б	124	
21	NAK	47	/	73	Ӣ	99	с	125	}
22	SYN	48	0	74	Ҷ	100	ҳ	126	~
23	ETB	49	1	75	Ҹ	101	ҵ	127	DEL
24	CAN	50	2	76	ҩ	102	Ҷ		
25	EM	51	3	77	Ҫ	103	ҵ		

Предопределенные именованные цвета

В табл. П4.1 перечислены предопределенные названия цветов, которые можно использовать с библиотекой черепашьей графики, пакетами `matplotlib` и `tkinter`.

Таблица П4.1

'snow'	'ghost white'	'white smoke'
'gainsboro'	'floral white'	'old lace'
'linen'	'antique white'	'papaya whip'
'blanched almond'	'bisque'	'peach puff'
'navajo white'	'lemon chiffon'	'mint cream'
'azure'	'alice blue'	'lavender'
'lavender blush'	'misty rose'	'dark slate gray'
'dim gray'	'slate gray'	'light slate gray'
'gray'	'light grey'	'midnight blue'
'navy'	'cornflower blue'	'dark slate blue'
'slate blue'	'medium slate blue'	'light slate blue'
'medium blue'	'royal blue'	'blue'
'dodger blue'	'deep sky blue'	'sky blue'
'light sky blue'	'steel blue'	'light steel blue'
'light blue'	'powder blue'	'pale turquoise'
'dark turquoise'	'medium turquoise'	'turquoise'
'cyan'	'light cyan'	'cadet blue'
'medium aquamarine'	'aquamarine'	'dark green'
'dark olive green'	'dark sea green'	'sea green'
'medium sea green'	'light sea green'	'pale green'
'spring green'	'lawn green'	'medium spring green'
'green yellow'	'lime green'	'yellow green'
'forest green'	'olive drab'	'dark khaki'
'khaki'	'pale goldenrod'	'light goldenrod yellow'
'light yellow'	'yellow'	'gold'

Таблица П4.1 (продолжение)

'light goldenrod'	'goldenrod'	'dark goldenrod'
'rosy brown'	'indian red'	'saddle brown'
'sandy brown'	'dark salmon'	'salmon'
'light salmon'	'orange'	'dark orange'
'coral'	'light coral'	'tomato'
'orange red'	'red'	'hot pink'
'deep pink'	'pink'	'light pink'
'pale violet red'	'maroon'	'medium violet red'
'violet red'	'medium orchid'	'dark orchid'
'dark violet'	'blue violet'	'purple'
'medium purple'	'thistle'	'snow2'
'snow3'	'snow4'	'seashell2'
'seashell3'	'seashell4'	'AntiqueWhite1'
'AntiqueWhite2'	'AntiqueWhite3'	'AntiqueWhite4'
'bisque2'	'bisque3'	'bisque4'
'PeachPuff2'	'PeachPuff3'	'PeachPuff4'
'NavajoWhite2'	'NavajoWhite3'	'NavajoWhite4'
'LemonChiffon2'	'LemonChiffon3'	'LemonChiffon4'
'cornsilk2'	'cornsilk3'	'cornsilk4'
'ivory2'	'ivory3'	'ivory4'
'honeydew2'	'honeydew3'	'honeydew4'
'LavenderBlush2'	'LavenderBlush3'	'LavenderBlush4'
'MistyRose2'	'MistyRose3'	'MistyRose4'
'azure2'	'azure3'	'azure4'
'SlateBlue1'	'SlateBlue2'	'SlateBlue3'
'SlateBlue4'	'RoyalBlue1'	'RoyalBlue2'
'RoyalBlue3'	'RoyalBlue4'	'blue2'
'blue4'	'DodgerBlue2'	'DodgerBlue3'
'DodgerBlue4'	'SteelBlue1'	'SteelBlue2'
'SteelBlue3'	'SteelBlue4'	'DeepSkyBlue2'
'DeepSkyBlue3'	'DeepSkyBlue4'	'SkyBlue1'
'SkyBlue2'	'SkyBlue3'	'SkyBlue4'
'LightSkyBlue1'	'LightSkyBlue2'	'LightSkyBlue3'
'LightSkyBlue4'	'SlateGray1'	'SlateGray2'
'SlateGray3'	'SlateGray4'	'LightSteelBlue1'

Таблица П4.1 (продолжение)

'LightSteelBlue2'	'LightSteelBlue3'	'LightSteelBlue4'
'LightBlue1'	'LightBlue2'	'LightBlue3'
'LightBlue4'	'LightCyan2'	'LightCyan3'
'LightCyan4'	'PaleTurquoise1'	'PaleTurquoise2'
'PaleTurquoise3'	'PaleTurquoise4'	'CadetBlue1'
'CadetBlue2'	'CadetBlue3'	'CadetBlue4'
'turquoise1'	'turquoise2'	'turquoise3'
'turquoise4'	'cyan2'	'cyan3'
'cyan4'	'DarkSlateGray1'	'DarkSlateGray2'
'DarkSlateGray3'	'DarkSlateGray4'	'aquamarine2'
'aquamarine4'	'DarkSeaGreen1'	'DarkSeaGreen2'
'DarkSeaGreen3'	'DarkSeaGreen4'	'SeaGreen1'
'SeaGreen2'	'SeaGreen3'	'PaleGreen1'
'PaleGreen2'	'PaleGreen3'	'PaleGreen4'
'SpringGreen2'	'SpringGreen3'	'SpringGreen4'
'green2'	'green3'	'green4'
'chartreuse2'	'chartreuse3'	'chartreuse4'
'OliveDrab1'	'OliveDrab2'	'OliveDrab4'
'DarkOliveGreen1'	'DarkOliveGreen2'	'DarkOliveGreen3'
'DarkOliveGreen4'	'khaki1'	'khaki2'
'khaki3'	'khaki4'	'LightGoldenrod1'
'LightGoldenrod2'	'LightGoldenrod3'	'LightGoldenrod4'
'LightYellow2'	'LightYellow3'	'LightYellow4'
'yellow2'	'yellow3'	'yellow4'
'gold2'	'gold3'	'gold4'
'goldenrod1'	'goldenrod2'	'goldenrod3'
'goldenrod4'	'DarkGoldenrod1'	'DarkGoldenrod2'
'DarkGoldenrod3'	'DarkGoldenrod4'	'RosyBrown1'
'RosyBrown2'	'RosyBrown3'	'RosyBrown4'
'IndianRed1'	'IndianRed2'	'IndianRed3'
'IndianRed4'	'sienna1'	'sienna2'
'sienna3'	'sienna4'	'burlywood1'
'burlywood2'	'burlywood3'	'burlywood4'
'wheat1'	'wheat2'	'wheat3'
'wheat4'	'tan1'	'tan2'

Таблица П4.1 (продолжение)

'tan4'	'chocolate1'	'chocolate2'
'chocolate3'	'firebrick1'	'firebrick2'
'firebrick3'	'firebrick4'	'brown1'
'brown2'	'brown3'	'brown4'
'salmon1'	'salmon2'	'salmon3'
'salmon4'	'LightSalmon2'	'LightSalmon3'
'LightSalmon4'	'orange2'	'orange3'
'orange4'	'DarkOrange1'	'DarkOrange2'
'DarkOrange3'	'DarkOrange4'	'coral1'
'coral2'	'coral3'	'coral4'
'tomato2'	'tomato3'	'tomato4'
'OrangeRed2'	'OrangeRed3'	'OrangeRed4'
'red2'	'red3'	'red4'
'DeepPink2'	'DeepPink3'	'DeepPink4'
'HotPink1'	'HotPink2'	'HotPink3'
'HotPink4'	'pink1'	'pink2'
'pink3'	'pink4'	'LightPink1'
'LightPink2'	'LightPink3'	'LightPink4'
'PaleVioletRed1'	'PaleVioletRed2'	'PaleVioletRed3'
'PaleVioletRed4'	'maroon1'	'maroon2'
'maroon3'	'maroon4'	'VioletRed1'
'VioletRed2'	'VioletRed3'	'VioletRed4'
'magenta2'	'magenta3'	'magenta4'
'orchid1'	'orchid2'	'orchid3'
'orchid4'	'plum1'	'plum2'
'plum3'	'plum4'	'MediumOrchid1'
'MediumOrchid2'	'MediumOrchid3'	'MediumOrchid4'
'DarkOrchid1'	'DarkOrchid2'	'DarkOrchid3'
'DarkOrchid4'	'purple1'	'purple2'
'purple3'	'purple4'	'MediumPurple1'
'MediumPurple2'	'MediumPurple3'	'MediumPurple4'
'thistle1'	'thistle2'	'thistle3'
'thistle4'	'gray1'	'gray2'
'gray3'	'gray4'	'gray5'
'gray6'	'gray7'	'gray8'

Таблица П4.1 (окончание)

'gray9'	'gray10'	'gray11'
'gray12'	'gray13'	'gray14'
'gray15'	'gray16'	'gray17'
'gray18'	'gray19'	'gray20'
'gray21'	'gray22'	'gray23'
'gray24'	'gray25'	'gray26'
'gray27'	'gray28'	'gray29'
'gray30'	'gray31'	'gray32'
'gray33'	'gray34'	'gray35'
'gray36'	'gray37'	'gray38'
'gray39'	'gray40'	'gray42'
'gray43'	'gray44'	'gray45'
'gray46'	'gray47'	'gray48'
'gray49'	'gray50'	'gray51'
'gray52'	'gray53'	'gray54'
'gray55'	'gray56'	'gray57'
'gray58'	'gray59'	'gray60'
'gray61'	'gray62'	'gray63'
'gray64'	'gray65'	'gray66'
'gray67'	'gray68'	'gray69'
'gray70'	'gray71'	'gray72'
'gray73'	'gray74'	'gray75'
'gray76'	'gray77'	'gray78'
'gray79'	'gray80'	'gray81'
'gray82'	'gray83'	'gray84'
'gray85'	'gray86'	'gray87'
'gray88'	'gray89'	'gray90'
'gray91'	'gray92'	'gray93'
'gray94'	'gray95'	'gray97'
'gray98'	'gray99'	

Модуль — это файл с исходным кодом на языке Python, который содержит функции и/или классы. Многие функции в стандартной библиотеке Python хранятся в модулях. Например, математический модуль `math` содержит различные математические функции, а модуль `random` — функции для работы со случайными числами.

Для того чтобы применять функции и/или классы, которые хранятся в модуле, нужно импортировать модуль. Для этого в самом начале своей программы следует поместить инструкцию `import`. Вот пример инструкции `import`, которая импортирует модуль `math`:

```
import math
```

Эта инструкция приводит к тому, что интерпретатор Python загрузит содержимое модуля `math` в оперативную память, делая функции и/или классы, которые хранятся в модуле `math`, доступными вашей программе. Для того чтобы использовать любой элемент, который находится в модуле, следует использовать *полностью определенное имя* элемента. Это означает, что перед именем элемента надо поставить имя модуля и затем точку. Например, модуль `math` содержит функцию с именем `sqrt`, которая возвращает квадратный корень числа. Для того чтобы вызвать функцию `sqrt`, следует написать имя `math.sqrt`. Следующий ниже интерактивный сеанс демонстрирует этот пример:

```
>>> import math
>>> x = math.sqrt(25)
>>> print(x)
5.0
>>>
```

Импортирование конкретной функции или класса

Приведенная выше форма инструкции `import` загружает все содержимое модуля в оперативную память. Иногда требуется импортировать из модуля *только* конкретную функцию или класс. В подобном случае можно воспользоваться ключевым словом `from` вместе с инструкцией `import`:

```
from math import sqrt
```

Эта инструкция приводит к импортированию из модуля `math` только функции `sqrt`. Такой подход позволяет вызывать функцию `sqrt`, не ставя имя модуля перед именем функции. Вот пример:

```
>>> from math import sqrt
>>> x = sqrt(25)
>>> print(x)
5.0
>>>
```

При использовании этой формы инструкции `import` можно указать имена нескольких элементов, разделенных запятыми. Например, инструкция `import` в следующем ниже интерактивном сеансе импортирует из модуля `math` только функции `sqrt` и `radians`:

```
>>> from math import sqrt, radians
>>> x = sqrt(25)
>>> a = radians(180)
>>> print(x)
5.0
>>> print(a)
3.141592653589793
>>>
```

Импорт с подстановочным символом

Инструкция `import` с подстановочным символом загружает все содержимое модуля. Вот пример:

```
from math import *
```

Разница между этой инструкцией и инструкцией `import math` состоит в том, что инструкция `import` с подстановочным символом не требует использования в модуле полностью определенных имен элементов. Например, вот интерактивный сеанс, который демонстрирует применение инструкции `import` с подстановочным символом:

```
>>> from math import *
>>> x = sqrt(25)
>>> a = radians(180)
>>>
```

А вот интерактивный сеанс, который использует обычную инструкцию `import`:

```
>>> import math
>>> x = math.sqrt(25)
>>> a = math.radians(180)
>>>
```

По возможности следует избегать использования инструкции `import` с подстановочным символом, потому что она может привести к конфликту имен во время импортирования нескольких модулей. *Конфликт имен* происходит, когда программа импортирует два модуля, которые имеют функции или классы с одинаковыми именами. Конфликта имен не возникает при использовании полностью определенных имен функций и/или классов модуля.

Использование псевдонимов

Ключевое слово `as` можно использовать для присвоения модулю *псевдонима* во время его импортирования. Вот пример:

```
import math as mt
```

Эта инструкция загружает в оперативную память модуль `math`, присваивая ему псевдоним `mt`. Для того чтобы использовать любой элемент, который находится в модуле, перед именем элемента следует поставить псевдоним и после него точку. Например, для вызова функции

sqrt следует использовать имя mt.sqrt. Следующий ниже интерактивный сеанс демонстрирует пример:

```
>>> import math as mt
>>> x = mt.sqrt(25)
>>> a = mt.radians(180)
>>> print(x)
5.0
>>> print(a)
3.141592653589793
>>>
```

Псевдоним также можно присвоить определенной функции или классу во время их импортирования. Следующая ниже инструкция импортирует функцию sqrt из модуля math и присваивает этой функции псевдоним square_root:

```
from math import sqrt as square_root
```

После применения этой инструкции import для вызова функции sqrt будет использоваться имя square_root. Следующий интерактивный сеанс показывает пример:

```
>>> from math import sqrt as square_root
>>> x = square_root(25)
>>> print(x)
5.0
>>>
```

В приведенном далее интерактивном сеансе мы импортируем из модуля import две функции, давая каждой из них псевдоним. Функция sqrt импортируется как square_root, а функция tan — как tangent:

```
>>> from math import sqrt as square_root, tan as tangent
>>> x = square_root(25)
>>> y = tangent(45)
>>> print(x)
5.0
>>> print(y)
1.6197751905438615
```



ПРИМЕЧАНИЕ

На протяжении всей этой книги мы использовали f-строки в качестве предпочтительного метода форматирования вывода программ. F-строки были введены в Python версии 3.6. Если вы работаете с более ранней версией Python, рассмотрите возможность применения функции *format()*, как описано в этом приложении.

Программиста не всегда устраивает то, как отображаются на экране числа, в особенности числа с плавающей точкой. Когда функция *print()* выводит на экран число с плавающей точкой, оно может содержать до 12 значащих цифр. Это показано в выводе программы П6.1. Поскольку эта программа выводит на экран сумму в долларах, было бы неплохо, чтобы эта сумма была округлена до двух знаков после запятой. К счастью, Python дает нам такую возможность, а также многое другое с помощью встроенной функции форматирования *format()*.

Программа П6.1 (no_formatting.py)

```
1 # Эта программа демонстрирует, как число с плавающей
2 # точкой выводится на экран без форматирования.
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12.0
5 print('Ежемесячный платеж составляет', monthly_payment)
```

Вывод программы

Ежемесячный платеж составляет 416.666666667

При вызове встроенной функции *format()* передаются два аргумента: числовое значение и спецификатор формата. *Спецификатор формата* — это строковое значение, которое содержит специальные символы, указывающие на то, как должно быть отформатировано числовое значение. Давайте рассмотрим пример:

```
format(12345.6789, '.2f')
```

Первый аргумент, представляющий собой число с плавающей точкой 12345.6789, является числом, которое мы хотим отформатировать. Второй аргумент, представляющий собой строку '.2f', является спецификатором формата. Вот смысл его содержимого:

- ◆ .2 сообщает точность. Этот спецификатор указывает на то, что мы хотим округлить число до двух знаков после точки;
- ◆ f указывает на то, что тип данных числа, которое мы форматируем, является числом с плавающей точкой (от англ. *float*).

Функция `format` возвращает строковое значение, содержащее отформатированное число. В приведенном ниже сеансе интерактивного режима показано, как можно использовать функцию `format` вместе с функцией `print` для вывода на экран отформатированного числа:

```
>>> print(format(12345.6789, '.2f')) [Enter]
12345.68
>>>
```

Обратите внимание, что число *округлено* до двух знаков после точки. В следующем ниже примере показано то же число, округленное до одного десятичного знака:

```
>>> print(format(12345.6789, '.1f')) [Enter]
12345.7
>>>
```

Вот еще один пример:

```
>>> print('Число равно', format(1.234567, '.2f')) [Enter]
Число равно 1.23
>>>
```

В программе П6.2 показано, как можно модифицировать программу П6.1, чтобы она форматировала свой результат с помощью этого технического приема.

Программа П6.2 (formatting.py)

```
1 # Эта программа демонстрирует возможный способ
2 # форматирования числа с плавающей точкой.
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12
5 print('Ежемесячный платеж составляет',
6      format(monthly_payment, '.2f'))
```

Вывод программы

```
Ежемесячный платеж составляет 416.67
```

Форматирование в научной нотации

Если вы предпочтете выводить на экран числа с плавающей точкой в научной нотации, то вместо буквы `f` вы можете использовать букву `e` или `E`. Вот несколько примеров:

```
>>> print(format(12345.6789, 'e')) [Enter]
1.234568e+04
>>> print(format(12345.6789, '.2e')) [Enter]
1.23e+04
>>>
```

Первая инструкция просто форматирует число в научной нотации. Число выводится на экран с буквой `e`, обозначающей показатель степени. (Если в спецификаторе формата используется `E` в верхнем регистре, то результат будет содержать `E` в верхнем регистре, обозначая показатель степени.) Во второй инструкции дополнительно указывается точность в два знака после точки.

Вставка запятых в качестве разделителей

Если вы хотите, чтобы число было отформатировано с запятыми в качестве разделителей, можете вставить запятую в спецификатор формата, как показано ниже:

```
>>> print(format(12345.6789, ',.2f')) Enter
12,345.68
>>>
```

Вот пример, который форматирует длинное число:

```
>>> print(format(123456789.456, ',.2f')) Enter
123,456,789.46
>>>
```

Обратите внимание, что в спецификаторе формата запятая записывается *перед (слева)* указателем точности. Вот пример, в котором задана запятая в качестве разделителя, но не указана точность:

```
>>> print(format(12345.6789, ',f')) Enter
12,345.678900
>>>
```

В программе П6.3 показано, как запятая в качестве разделителя и точность в два знака после точки могут использоваться для форматирования больших чисел в виде сумм в валюте.

Программа П6.3 (dollar_display.py)

```
1 # Эта программа демонстрирует вывод на экран
2 # числа с плавающей точкой в качестве валюты.
3 monthly_pay = 5000.0
4 annual_pay = monthly_pay * 12
5 print('Ваш годовой платеж составляет $',
6      format(annual_pay, ',.2f'),
7      sep='')
```

Вывод программы

```
Ваш годовой платеж составляет $60,000.00
```

Обратите внимание, что в строке 7 мы передали в функцию `print` аргумент `sep=""`. Он указывает на то, что между выводимыми на печать элементами не должно быть пробелов. Если бы этот аргумент не был бы передан, то после знака \$ был бы напечатан пробел.

Указание минимальной ширины поля

Спецификатор формата также может содержать минимальную ширину поля, т. е. минимальное число символов, которые должны использоваться для отображения значения на экране. В следующем примере выводится число в поле шириной 12 символов:

```
>>> print('Число равно', format(12345.6789, '12,.2f')) Enter
Число равно 12,345.68
>>>
```

В этом примере 12, появляющемся в спецификаторе формата, указывает на то, что число должно выводиться в поле шириной не менее 12 символов. В данном случае выводимое число короче отводимого под него поля. Число 12,345.68 используется на экране только 9 знаков, но выводится в поле шириной 12 символов. В этом случае число в поле выравнивается по правому краю. Если значение слишком велико, чтобы поместиться в заданную ширину поля, то поле автоматически увеличивается.

В предыдущем примере обратите внимание на то, что условное обозначение ширины поля записывается *перед* (слева) запятой-разделителем. Вот пример, который задает ширину и точность поля, но не использует запятые-разделители:

```
>>> print('Число равно', format(12345.6789, '12.2f')) Enter
Число равно 12345.68
>>>
```

Ширина полей помогает в ситуациях, когда нужно вывести числа, выровненные в столбце. Например, посмотрите программу П6.4. Все переменные выводятся в поле шириной семь символов.

Программа П6.4 (columns.py)

```
1 # Эта программа демонстрирует столбец из
2 # чисел с плавающей точкой,
3 # выровненных по десятичной точке.
4 num1 = 127.899
5 num2 = 3465.148
6 num3 = 3.776
7 num4 = 264.821
8 num5 = 88.081
9 num6 = 799.999
10
11 # Показать каждое число в поле из 7 символов
12 # с 2 десятичными знаками.
13 print(format(num1, '7.2f'))
14 print(format(num2, '7.2f'))
15 print(format(num3, '7.2f'))
16 print(format(num4, '7.2f'))
17 print(format(num5, '7.2f'))
18 print(format(num6, '7.2f'))
```

Вывод программы

```
127.90
3465.15
      3.78
  264.82
  88.08
800.00
```

Процентный формат чисел с плавающей точкой

Помимо использования `f` в качестве условного обозначения типа можно использовать символ `%` для процентного форматирования числа с плавающей точкой. Применение символа `%` приводит к тому, что число умножается на 100 и выводится со знаком `%` после него. Вот пример:

```
>>> print(format(0.5, '%')) [Enter]
50.000000%
>>>
```

Вот пример, в котором в качестве точности указывается 0:

```
>>> print(format(0.5, '.0%')) [Enter]
50%
>>>
```

Форматирование целых чисел

Все приведенные выше примеры демонстрировали способы форматирования чисел с плавающей точкой. Функцию `format` также можно использовать для форматирования целых чисел. При написании спецификатора формата в данном случае следует иметь в виду две особенности:

- ◆ в качестве условного обозначения типа используется `d`;
- ◆ точность не указывается.

Давайте рассмотрим несколько примеров в интерактивном интерпретаторе. В следующем ниже сеансе число 123456 выводится на экран без специального форматирования:

```
>>> print(format(123456, 'd')) [Enter]
123456
>>>
```

В следующем далее сеансе число 123456 выводится с запятой в качестве разделителя тысяч:

```
>>> print(format(123456, ',d')) [Enter]
123,456
>>>
```

В следующем ниже сеансе число 123456 выводится в поле шириной 10 символов:

```
>>> print(format(123456, '10d')) [Enter]
123456
>>>
```

В следующем ниже сеансе число 123456 выводится с запятой в качестве разделителя тысяч в поле шириной 10 символов:

```
>>> print(format(123456, '10,d')) [Enter]
123,456
>>>
```

Установка модулей при помощи менеджера пакетов *pip*

Стандартная библиотека Python предоставляет классы и функции, которые ваши программы могут использовать для выполнения базовых операций, а также многих продвинутых задач. Вместе с тем существуют операции, которые стандартная библиотека выполнить не сможет. Когда нужно сделать нечто, выходящее за рамки стандартной библиотеки, можно либо написать программный код самому, либо использовать программный код, который уже кем-то был создан.

К счастью, существуют тысячи модулей Python, написанных независимыми программистами, предлагающие возможности, которые отсутствуют в стандартной библиотеке Python. Они называются *сторонними модулями*. Огромная коллекция сторонних модулей существует на веб-сайте pypi.python.org, так называемом *каталоге пакетов Python*, или PyPI (Python Package Index).

Имеющиеся в PyPI модули организованы в пакеты. *Пакет* — это просто коллекция из одного или нескольких связанных между собой модулей. Самый простой способ скачать и установить любой пакет предполагает использование менеджера пакетов *pip*. Менеджер пакетов *pip* является составной частью стандартной установки Python, начиная с Python 3.4. Для того чтобы установить необходимый пакет в операционной системе Windows с помощью менеджера пакетов *pip*, следует открыть окно командной оболочки и ввести команду в следующем формате:

```
pip install имя_пакета
```

где *имя_пакета* — это имя пакета, который вы хотите скачать и установить. Если вы работаете в среде операционной системы Mac OS X или Linux, вместо команды *pip* следует использовать команду *pip3*. Помимо этого, чтобы выполнить команду *pip3* в операционной системе Mac OS X или Linux, потребуются полномочия суперпользователя, поэтому придется снабдить эту команду префиксом в виде команды *sudo*:

```
sudo pip3 install имя_пакета
```

После ввода команды менеджер пакетов *pip* начнет скачивать и устанавливать пакет. В зависимости от размера пакета выполнение всего процесса установки может занять несколько минут. Когда этот процесс будет завершен, удостовериться, что пакет был правильно установлен, как правило, можно, запустив среду IDLE и введя команду

```
>>> import имя_пакета
```

где *имя_пакета* — это имя пакета, который вы установили. Если вы не видите сообщение об ошибке, то можете считать, что пакет был успешно установлен.

Глава 1

- 1.1. Программа — это набор инструкций, который компьютер выполняет, чтобы решить задачу.
- 1.2. Аппаратное обеспечение — это все физические устройства, или компоненты, из которых состоит компьютер.
- 1.3. Центральный процессор (ЦП), оперативная память, внешние устройства хранения, устройства ввода и устройства вывода.
- 1.4. ЦП.
- 1.5. Основная память.
- 1.6. Вторичная память.
- 1.7. Устройство ввода.
- 1.8. Устройство вывода.
- 1.9. Операционная система.
- 1.10. Обслуживающая программа, или утилита.
- 1.11. К прикладному программному обеспечению.
- 1.12. Достаточно одного байта.
- 1.13. Бит, или разряд.
- 1.14. В двоичной системе исчисления.
- 1.15. Схема кодирования ASCII использует набор из 128 числовых кодов для представления английских букв, различных знаков препинания и других символов. Эти числовые коды нужны для хранения символов в памяти компьютера. (Аббревиатура ASCII означает стандартный американский код обмена информацией.)
- 1.16. Юникод.
- 1.17. Цифровые данные — это данные, которые хранятся в двоичном файле, цифровое устройство — это любое устройство, которое работает с двоичными данными.
- 1.18. Этот язык называется машинным языком.
- 1.19. Этот тип памяти называется основной памятью, или ОЗУ.
- 1.20. Этот процесс называется циклом выборки-декодирования-исполнения.

- 1.21. Это альтернатива машинному языку. Вместо двоичных чисел ассемблер использует в качестве инструкций короткие слова, которые называются мнемокодами.
- 1.22. Высокоуровневый язык.
- 1.23. Этот набор правил называется синтаксическими правилами.
- 1.24. Компилятор.
- 1.25. Интерпретатор.
- 1.26. Они являются причинами синтаксической ошибки.

Глава 2

- 2.1. Любой человек, группа или организация, которые поручают написать программу.
- 2.2. Отдельная функция, которую программа должна выполнить для удовлетворения потребностей клиента.
- 2.3. Набор четко сформулированных логических шагов, которые должны быть проделаны для выполнения задачи.
- 2.4. Неформальный язык, который не имеет каких-либо синтаксических правил и не предназначен для компиляции или исполнения. Программисты используют псевдокод для создания моделей или "макетов" программ.
- 2.5. Диаграмма, которая графически изображает шаги, имеющие место в программе.
- 2.6. Овалы — это терминальные символы. Параллелограммы — это входные либо выходные символы. Прямоугольники — это обрабатывающие символы.
- 2.7. `print('Джимми Смит')`
- 2.8. `print("Python – лучше всех!")`
- 2.9. `print('Кошка сказала "мяу".')`
- 2.10. Имя, которое ссылается на значение в оперативной памяти компьютера.
- 2.11. Имя `99bottles` недопустимо, потому что оно начинается с цифры. Имя `r&d` недопустимо, т. к. использовать символ & не разрешается.
- 2.12. Не являются, потому что имена переменных регистрозависимы.
- 2.13. Недопустима, потому что переменная, которая получает присваивание (в данном случае `amount`), должна находиться с левой стороны от оператора `=`.
- 2.14. Значение равняется `val`
- 2.15. Переменная `value1` будет ссылаться на целочисленный тип `int`. Переменная `value2` будет ссылаться на вещественный тип `float`. Переменная `value3` будет ссылаться на вещественный тип `float`. Переменная `value4` будет ссылаться на целочисленный тип `int`. Переменная `value5` будет ссылаться на строковый тип `str` (`string`).
- 2.16. 0
- 2.17. `last_name = input("Введите фамилию клиента: ")`
- 2.18. `sales = float(input('Введите объем продаж за неделю: '))`

2.19. Вот заполненная таблица:

Выражение	Значение
$6 + 3 * 5$	21
$12 / 2 - 4$	2
$9 + 14 * 2 - 6$	31
$(6 + 2) * 3$	24
$14 / (11 - 4)$	2
$9 + 12 * (8 - 3)$	69

2.20. 4.

2.21. 1.

2.22. Конкатенация строк — это добавление одного строкового литерала в конец другого.

2.23. '12'.

2.24. 'привет'.

2.25. Если требуется, чтобы функция `print` не начинала новую строку вывода по завершении вывода данных, следует передать в эту функцию специальный аргумент `end = ' '`.

2.26. Можно передать в функцию `print` аргумент `sep=`, указав нужный символ.

2.27. Это экранированный символ новой строки.

2.28. Привет {name}

2.29. Привет Карли

2.30. Значение равно 100

2.31. Значение равно 65.43

2.32. Значение равно 987,654.13

2.33. Значение равно 9,876,543,210

2.34. Это условное обозначение ширины поля. Оно указывает на то, что значение должно выводиться на экран в поле шириной не менее 10 символов.

2.35. Это условное обозначение ширины поля. Оно указывает на то, что значение должно выводиться на экран в поле шириной не менее 15 символов.

2.36. Это условное обозначение ширины поля. Оно указывает на то, что значение должно выводиться на экран в поле шириной не менее 8 символов.

2.37. Это спецификатор выравнивания. Он указывает на то, что значение должно быть выровнено по левому краю.

2.38. Это спецификатор выравнивания. Он указывает на то, что значение должно быть выровнено по правому краю.

2.39. Это спецификатор выравнивания. Он указывает на то, что значение должно быть выровнено по середине.

2.40. Именованные константы:

- делают программы более очевидными;
- позволяют легко вносить изменения в программный код, не затрачивая много усилий;
- помогают избежать опечаток, которые часто случаются при использовании магических чисел.

2.41. `DISCOUNT_PERCENTAGE = 0.1`**2.42. 0 градусов.****2.43. При помощи команды `turtle.forward()`.****2.44. При помощи команды `turtle.right(45)`.****2.45. Сначала надо применить команду `turtle.penup()`, чтобы поднять перо.****2.46. `turtle.heading()`****2.47. `turtle.circle(100)`****2.48. `turtle.pensize(8)`****2.49. `turtle.pencolor('blue')`****2.50. `turtle.bgcolor('black')`****2.51. `turtle.setup(500, 200)`****2.52. `turtle.goto(100, 50)`****2.53. `turtle.pos()`****2.54. `turtle.speed(10)`****2.55. `turtle.speed(0)`****2.56. В целях заполнения фигуры цветом следует перед ее рисованием применить команду `turtle.begin_fill()`. А после того как фигура нарисована, применить команду `turtle.end_fill()`. Во время исполнения команды `turtle.end_fill()` фигура будет заполнена текущим цветом заливки.****2.57. С помощью команды `turtle.write()`.****2.58. `radius = turtle.numinput('Введите значение',
'Каков радиус окружности?')`**

Глава 3

- 3.1. Это логическая схема, которая управляет порядком, в котором исполняется набор инструкций.
- 3.2. Это структура программы, которая может выполнить ряд инструкций только при определенных обстоятельствах.
- 3.3. Структура решения, которая обеспечивает единственный вариант пути исполнения. Если проверяемое условие является истинным, то программа принимает этот вариант пути.
- 3.4. Выражение, которое в результате его вычисления может иметь только одно из двух значений: истина либо ложь.

3.5. Можно определить, является ли одно значение больше другого, меньше его, больше или равно ему, меньше или равно ему, равно или не равно ему.

3.6. `if y == 20:`

`x = 0`

3.7. `if sales >= 10000:`

`commissionRate = 0.2`

3.8. Структура принятия решения с двумя альтернативными вариантами имеет два возможных пути исполнения; один путь принимается, если условие является истинным, а другой путь принимается, если условие является ложным.

3.9. Инструкцию `if-else`.

3.10. Когда условие является ложным.

3.11. `z` не меньше `a`.

3.12. Бостон

Нью-Йорк

3.13. `if number == 1:`

`print('Один')`

`elif number == 2:`

`print('Два')`

`elif number == 3:`

`print('Три')`

`else:`

`print('Неизвестное')`

3.14. Это выражение, которое создается при помощи логического оператора для объединения двух булевых выражений.

3.15. л

и

л

л

и

и

и

л

л

и

3.16. и

л

и

и

и

3.17. Оператор `and`: если выражение слева от оператора `and` ложное, то выражение справа от него не проверяется. Оператор `or`: если выражение слева от оператора `or` является истинным, то выражение справа от него не проверяется.

3.18. `if speed >= 0 and speed <= 200:`

`print('Допустимое число')`

3.19. if speed < 0 or speed > 200:
 print('Число не является допустимым')

3.20. Истина (True) или ложь (False).

3.21. Переменная, которая сигнализирует, когда в программе возникает некое условие.

3.22. Необходимо применить функции `turtle.xcor()` и `turtle.ycor()`.

3.23. Необходимо применить оператор `not` с функцией `turtle.isdown()`, как здесь:

```
if not turtle.isdown():  
    инструкция
```

3.24. Необходимо применить функцию `turtle.heading()`.

3.25. Необходимо применить функцию `turtle.isvisible()`.

3.26. Для определения цвета пера применяется функция `turtle.pencolor()`. Для определения текущего цвета заливки — функция `turtle.fillcolor()`. Для определения текущего фонового цвета графического окна черепахи — функция `turtle.bgcolor()`.

3.27. Необходимо применить функцию `turtle.pensize()`.

3.28. Необходимо применить функцию `turtle.speed()`.

Глава 4

- 4.1. Это структура, которая приводит к неоднократному исполнению набора инструкций.
- 4.2. Это цикл, который использует логическое условие со значениями истина/ложь для управления количеством повторений.
- 4.3. Это цикл, который повторяется заданное количество раз.
- 4.4. Каждое отдельное исполнение инструкций в теле цикла.
- 4.5. До.
- 4.6. Нисколько. Условие `count < 1` будет ложным с самого начала.
- 4.7. Цикл, который не имеет возможности завершиться и продолжается до тех пор, пока программа не будет прервана.
- 4.8. `for x in range(6):
 print('Обожаю эту программу!')`

- 4.9. 0
1
2
3
4
5

- 4.10. 2
3
4
5

4.11. 0

100
200
300
400
500

4.12. 10

9
8
7
6

4.13. Переменная, которая используется для накапливания суммы ряда чисел.

4.14. Да, он должен быть инициализирован значением 0. Это вызвано тем, что значения добавляются в него в цикле. Если накопитель не начнется со значения 0, то он не будет содержать правильную сумму добавленных чисел, когда цикл завершится.

4.15. 15

4.16. 15
5

4.17. a) `quantity += 1;` б) `days_left -= 5;` в) `price *= 10;` г) `price /= 2.`

4.18. Сигнальная метка — это специальное значение, которое отмечает конец последовательности значений.

4.19. Значение сигнальной метки должно быть характерным настолько, чтобы программа не приняла его ошибочно за регулярное значение последовательности.

4.20. Она означает, что если на входе в программу предоставлены плохие данные (мусор), то программа произведет плохие данные (мусор) на выходе.

4.21. Когда программе предоставляются входные данные, они должны быть обследованы прежде, чем будут обработаны. Если входные данные недопустимые, то их следует отбросить, а пользователю предложить ввести правильные данные.

4.22. Считываются входные данные, затем исполняется цикл с предусловием. Если входные данные недопустимые, то исполняется тело цикла. В теле цикла выводится сообщение об ошибке, чтобы дать пользователю понять, что входные данные были недопустимыми, и затем входные данныечитываются снова. Этот цикл повторяется до тех пор, пока входные данные не будут допустимыми.

4.23. Это входная операция, которая происходит непосредственно перед циклом валидации входного значения. Его задача состоит в том, чтобы получить первое входное значение.

4.24. Нисколько.

Глава 5

- 5.1.** Функция — это группа инструкций, которая существует внутри программы с целью выполнения конкретной задачи.
- 5.2.** Большая задача подразделяется на несколько меньших задач, которые легко выполнить.

- 5.3. Если конкретная операция выполняется в программе в нескольких местах, то для выполнения этой операции можно один раз написать функцию, и затем ее вызывать в любое время, когда она понадобится.
- 5.4. Функции могут быть написаны для выполнения распространенных задач, которые требуются разными программами. Такие функции затем могут быть включены в состав любой программы, которая в них нуждается.
- 5.5. Когда программа разрабатывается как набор функций, каждая из которых выполняет отдельную задачу, в этом случае разным программистам может быть поручено написание разных функций.
- 5.6. Определение функции имеет две части: заголовок функции и блок инструкций. Заголовок отмечает начальную точку функции, а блок является списком инструкций, составляющих одно целое.
- 5.7. Вызвать функцию означает исполнить эту функцию.
- 5.8. Когда достигнут конец функции, управление возвращается назад к той части программы, которая вызвала эту функцию, и программа возобновляет исполнение с этой точки.
- 5.9. Потому что интерпретатор Python использует выделение отступом для определения мест, где блок инструкций начинается и завершается.
- 5.10. Локальная переменная — это переменная, которая объявляется внутри функции. Она принадлежит функции, в которой была объявлена, и к такой переменной могут получать доступ только инструкции в этой функции.
- 5.11. Это часть программы, в которой можно обращаться к переменной.
- 5.12. Да, разрешается.
- 5.13. Они называются аргументами.
- 5.14. Они называются параметрами.
- 5.15. Область действия параметрической переменной — это вся функция, в которой объявлен параметр.
- 5.16. Нет, не влияет.
- 5.17. а) передает именованные аргументы; б) передает позиционные аргументы.
- 5.18. Вся программа.
- 5.19. Вот три причины.
- Глобальные переменные затрудняют отладку программы. Значение глобальной переменной может быть изменено любой инструкцией в программном файле. Если вы обнаружите, что в глобальной переменной хранится неверное значение, то придется отыскать все инструкции, которые к ней обращаются, чтобы определить, откуда поступает плохое значение. В программе с тысячами строк программного кода такая работа может быть сопряжена с большими трудностями.
 - Функции, которые используют глобальные переменные, обычно зависят от этих переменных. Если вы захотите применить такую функцию в другой программе, то скорее всего вам придется эту функцию перепроектировать, чтобы она не опиралась на глобальную переменную.

- Глобальные переменные затрудняют понимание программы. Их можно модифицировать любой инструкцией в программе. Если вы собираетесь разобраться в какой-то части программы, которая использует глобальную переменную, то вам придется узнать обо всех других частях программы, которые обращаются к этой глобальной переменной.

- 5.20. Глобальная константа — это имя, которое доступно в программе любой функции. Глобальные константы разрешается использовать. Поскольку во время исполнения программы их значение не может быть изменено, не придется беспокоиться о том, что ее значение поменяется.
- 5.21. Разница между ними состоит в том, что функция с возвратом значения возвращает значение в инструкцию, которая ее вызвала. Простая функция значение не возвращает.
- 5.22. Заранее написанная функция, которая выполняет некую часто решаемую задачу.
- 5.23. Термин "черный ящик" используется для описания любого механизма, который принимает нечто на входе, выполняет с полученным на входе некоторую работу (которую невозможно наблюдать) и производит результат на выходе.
- 5.24. Она присваивает случайное целое число в диапазоне от 1 до 100 переменной `x`.
- 5.25. Она печатает случайное целое число в диапазоне от 1 до 20.
- 5.26. Она печатает случайное целое число в диапазоне от 10 до 19.
- 5.27. Она печатает случайное число с плавающей точкой в диапазоне от 0.0 до 1.0, но не включая 1.0.
- 5.28. Она печатает случайное число с плавающей точкой в диапазоне от 0.1 до 0.5.
- 5.29. Она применяет системное время из внутреннего генератора тактовых импульсов компьютера.
- 5.30. Если для генерации случайных чисел всегда использовать одинаковое начальное значение, то функции генерации случайных чисел всегда будут генерировать одинаковые ряды псевдослучайных чисел.
- 5.31. Она возвращает значение в ту часть программы, которая вызвала функцию.
- 5.32. а) `do_something`; б) возвращает удвоенное значение переданного в нее аргумента; в) 20.
- 5.33. Это функция, которая возвращает истину (`True`) либо ложь (`False`).
- 5.34. `import math`
- 5.35. `square_root = math.sqrt(100)`
- 5.36. `angle = math.radians(45)`

Глава 6

- 6.1. Это файл, в который программа записывает данные. Он называется файлом вывода, потому что программа отправляет в него выходные данные.
- 6.2. Это файл, из которого программа считывает данные. Он называется файлом ввода, потому что программа получает из него входные данные.
- 6.3. 1) открыть файл; 2) обработать файл; 3) закрыть файл.
- 6.4. Текстовый и двоичный файлы. Текстовый файл содержит данные, которые были закодированы в виде текста при помощи такой схемы кодирования, как ASCII. При этом,

даже если файл содержит числа, эти числа в файле хранятся как набор символов. В результате файл можно открыть и просмотреть в текстовом редакторе, таком как Блокнот. Двоичный файл содержит данные, которые не были преобразованы в текст. Вследствие этого содержимое двоичного файла невозможно просмотреть в текстовом редакторе.

- 6.5. Последовательный доступ и прямой доступ. Когда вы работаете с файлом с последовательным доступом, вы обращаетесь к данным с самого начала файла и до конца файла. Когда вы работаете с файлом с прямым доступом (который также называется файлом с произвольным доступом), вы можете непосредственно перескочить к любой порции данных в файле, не читая данные, которые идут перед ней.
- 6.6. Имя файла на диске и имя переменной, которая ссылается на файловый объект.
- 6.7. Содержимое файла стирается.
- 6.8. В результате открытия файла создается связь между файлом и программой, а также ассоциативная связь между файлом и файловым объектом.
- 6.9. В результате закрытия файла связь между программой и файлом разрывается.
- 6.10. Позиция считывания файла отмечает позицию следующего элемента, который будет прочитан из файла. Когда файл ввода открыт, его позиция считывания первоначально устанавливается на первом элементе в файле.
- 6.11. Файл открывается в режиме дозаписи. Когда данные записываются в файл в этом режиме, они записываются в конец существующего файла.
- 6.12.

```
outfile = open('numbers.txt', 'w')
for num in range(1, 11):
    outfile.write(str(num) + '\n')
outfile.close()
```
- 6.13. Метод `readline()` возвращает пустое строковое значение (''), когда он попытается прочитать за пределами конца файла.
- 6.14.

```
infile = open('data.txt', 'r')
line = infile.readline()
while line != '':
    print(line)
    line = infile.readline()
infile.close()
```
- 6.15.

```
infile = open('data.txt', 'r')
for line in infile:
    print(line)
infile.close()
```
- 6.16. Запись — это полный набор данных, который описывает один элемент, поле — отдельная порция данных в записи.
- 6.17. Все записи исходного файла копируются друг за другом во временный файл, но когда вы добираетесь до записи, которая должна быть изменена, старое содержимое записи во временный файл не записывается. Вместо этого во временный файл записываются измененные значения записи. Затем из исходного файла во временный файл копируются все остальные записи.

- 6.18.** Все записи исходного файла друг за другом копируются во временный файл, за исключением записи, которая должна быть удалена. Затем временный файл занимает место исходного файла. Исходный файл удаляется, а временный файл переименовывается, получая имя, которое исходный файл имел на диске компьютера.
- 6.19.** Исключение — это ошибка, которая происходит во время работы программы. В большинстве случаев исключение заставляет программу внезапно остановиться.
- 6.20.** Программа останавливается.
- 6.21.** IOError.
- 6.22.** ValueError.

Глава 7

- 7.1.** [1, 2, 99, 4, 5]
- 7.2.** [0, 1, 2]
- 7.3.** [10, 10, 10, 10, 10]
- 7.4.** 1
3
5
7
9
- 7.5.** 4
- 7.6.** Следует применить встроенную функцию len.
- 7.7.** [1, 2, 3]
[10, 20, 30]
[1, 2, 3, 10, 20, 30]
- 7.8.** [1, 2, 3]
[10, 20, 30, 1, 2, 3]
- 7.9.** [2, 3]
- 7.10.** [2, 3, 4, 5]
- 7.11.** [1]
- 7.12.** [1, 2, 3, 4, 5]
- 7.13.** [3, 4, 5]
- 7.14.** Семья Жасмин:
['Джим', 'Джилл', 'Джон', 'Жасмин']
- 7.15.** Метод remove() отыскивает и удаляет элемент, содержащий определенное значение. Инструкция del удаляет элемент в заданной индексной позиции.
- 7.16.** Для этого следует применить встроенные функции min и max.
- 7.17.** Нужно применить инструкцию names.append('Вэнди'). Это связано с тем, что элемент 0 не существует. Если попытаться применить инструкцию names[0] = 'Вэнди', то произойдет ошибка.

7.18. а) метод `index()` отыскивает значение в списке и возвращает индекс первого элемента, содержащего это значение; б) метод `insert()` вставляет значение в список в заданной индексной позиции; в) метод `sort()` сортирует значения в списке, в результате чего они появляются в возрастающем порядке; г) метод `reverse()` инвертирует порядок следования значений в списке.

7.19. Выражение результата таково: `x`. Выражение итерации таково: `for x in my_list`.

7.20. `[2, 24, 4, 40, 6, 30, 8]`

7.21. `[12, 20, 15]`

7.22. Этот список содержит 4 строки и 2 столбца.

7.23. `mylist = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]`

7.24. `for r in range(4):
 for c in range(2):
 print(numbers[r][c])`

7.25. Главное различие между кортежами и списками состоит в том, что кортежи являются немутабельными последовательностями, т. е. после создания кортежа он не может быть изменен.

7.26. Вот три причины.

- Обработка кортежа выполняется быстрее, чем обработка списка, и поэтому кортежи являются хорошим вариантом, когда обрабатывается большой объем данных, и эти данные не будут изменяться.
- Кортежи безопасны. Поскольку содержимое кортежа изменять нельзя, в нем можно хранить данные, оставаясь уверенным, что они не будут (случайно или каким-либо иным образом) изменены в программе.
- В Python существуют определенные операции, которые требуют применения кортежа.

7.27. `my_tuple = tuple(my_list)`

7.28. `my_list = list(my_tuple)`

7.29. Два списка: один с координатами X точек данных и другой с координатами Y точек данных.

7.30. Она строит линейный график.

7.31. Применяются функции `xlabel` и `ylabel`.

7.32. Нужно вызвать функции `xlim` и `ylim`, передав значения для именованных аргументов `xmin`, `xmax`, `ymax` и `ymax`.

7.33. Нужно вызвать функции `xticks` и `yticks`. Следует передать в эти функции два аргумента. Первый аргумент — это список позиций меток делений, а второй аргумент — список подписей в указанных позициях.

7.34. Два списка: один с координатами X левого края каждого столбика, а другой с высотами каждого столбика вдоль оси y .

7.35. Столбики будут красным, синим, красным и синим.

7.36. Следует в качестве аргумента передать список значений. Функция `rie` вычислит сумму значений в списке и затем будет использовать ее в качестве значения целого. Далее каждый элемент в списке станет сектором в круговой диаграмме. Размер сектора представляет значение этого элемента как процентное содержание целого.

Глава 8

8.1. `for letter in name:
 print(letter)`

8.2. 0.

8.3. 9.

8.4. Исключение `IndexError` произойдет, если попытаться использовать индекс, который выходит за пределы диапазона определенного строкового значения.

8.5. Следует применить встроенную функцию `len`.

8.6. Вторая инструкция пытается присвоить значение отдельному символу в строковом значении. Однако строковые данные являются немутируемыми последовательностями, поэтому выражение `animal[0]` недопустимо с левой стороны от оператора присваивания.

8.7. cde

8.8. defg

8.9. abc

8.10. abcdefg

8.11. `if 'd' in mystring:
 print('Да, она там.')`

8.12. `little = big.upper()`

8.13. `if ch.isdigit():
 print('Цифра')
else:
 print('Цифр нет')`

8.14. а А

8.15. `again = input('Желаете повторить ' +
 'программу или выйти? (П/В)')
while again.upper() != 'П' and again.upper() != 'В':
 again = input('Желаете повторить ' +
 'программу или выйти? (П/В)')`

8.16. §

8.17. `for letter in mystring:
 if letter.isupper():
 count += 1`

- 8.18. `my_list = days.split()`
8.19. `my_list = values.split('$')`

Глава 9

- 9.1. Ключ и значение.
9.2. Ключ.
9.3. Строковый литерал 'старт' является ключом, целое число 1472 — значением.
9.4. Она сохраняет пару "ключ : значение" 'id':54321 в словаре `employee`.
9.5. `bbb`
9.6. Для того чтобы проверить наличие конкретного ключа, используется оператор `in`.
9.7. Она удаляет элемент с ключом 654.
9.8. 3
9.9. 1
2
3
9.10. Метод `pop()` принимает ключ в качестве аргумента, возвращает значение, которое связано с этим ключом, и удаляет эту пару "ключ : значение" из словаря. Метод `popitem()` возвращает произвольно отобранный пару "ключ : значение" в качестве кортежа и удаляет эту пару "ключ : значение" из словаря.
9.11. Он возвращает все ключи и связанные с ними значения в словаре в качестве последовательности кортежей.
9.12. Он возвращает все ключи в словаре в виде последовательности кортежей.
9.13. Он возвращает все значение в словаре в виде последовательности кортежей.
9.14. `result = {item:len(item) for item in names}`
9.15. `phonebook_copy = {k:v for k,v in phonebook.items() if v.startswith('919')}`
9.16. Неупорядоченными.
9.17. Нет.
9.18. Нужно вызвать встроенную функцию `set`.
9.19. Множество будет содержать следующие элементы (в произвольном порядке): 'и', 'е', 'р', 'п', 'т', 'ю'.
9.20. Множество будет содержать один элемент — 25.
9.21. Множество будет содержать следующие элементы (в произвольном порядке): 'ю', 'я', 'э', 'и' и 'ъ'.
9.22. Множество будет содержать следующие элементы (в произвольном порядке): 1, 2, 3 и 4.
9.23. Множество будет содержать следующие элементы (в произвольном порядке): 'эээ', 'юю', 'ъъъ' и 'яяя'.

- 9.24. Следует передать множество в качестве аргумента в функцию `len`.
- 9.25. Множество будет содержать следующие элементы (в произвольном порядке): 10, 9, 8, 1, 2 и 3.
- 9.26. Множество будет содержать следующие элементы (в произвольном порядке): 10, 8, 9, 'a', 'б' и 'в'.
- 9.27. Если заданный элемент, подлежащий удалению, отсутствует в множестве, то метод `remove()` вызывает исключение `KeyError`, а метод `discard()` исключения не вызывает.
- 9.28. Для того чтобы проверить наличие элемента, используется оператор `in`.
- 9.29. `{10, 20, 30, 100, 200, 300}`.
- 9.30. `{3, 4}`.
- 9.31. `{1, 2}`.
- 9.32. `{5, 6}`.
- 9.33. `('a', 'г')`.
- 9.34. Множество `set2` является подмножеством множества `set1`, а множество `set1` — надмножеством множества `set2`.
- 9.35. Это процесс преобразования объекта в поток байтов, которые можно сохранить в файле для последующего извлечения.
- 9.36. `'wb'`.
- 9.37. `'rb'`.
- 9.38. Модуль `pickle`.
- 9.39. `pickle.dump`.
- 9.40. `pickle.load`.

Глава 10

- 10.1. Объект — это программная сущность, которая содержит данные и процедуры.
- 10.2. Инкапсуляция — объединение данных и программного кода в одном объекте.
- 10.3. Когда внутренние данные объекта скрыты от внешнего кода и доступ к этим данным ограничен методами объекта, эти данные защищены от случайного повреждения. Помимо этого, программному коду за пределами объекта не требуется знать о формате и внутренней структуре данных объекта.
- 10.4. Доступ к публичным методам может быть получен сущностями, находящимися за пределами объекта. Доступ к приватным методам закрыт для сущностей, находящихся за пределами объекта. Они предназначены для внутренних целей.
- 10.5. Метафора со строительным проектом представляет класс.
- 10.6. Объекты — это печенье.
- 10.7. Его задача состоит в инициализации атрибутов данных в объекте. Он исполняется сразу после создания объекта.

- 10.8. Во время выполнения метод должен знать, атрибутами данных какого объекта он призван оперировать. Именно здесь на первый план выходит параметр `self`. Во время вызова метода Python делает так, что параметр `self` ссылается на конкретный объект, которым этот метод призван оперировать.
- 10.9. Предварив имя атрибута двумя символами подчеркивания.
- 10.10. Он возвращает строковое представление объекта.
- 10.11. Путем передачи объекта во встроенный метод `str()`.
- 10.12. Атрибут, который принадлежит определенному экземпляру класса.
- 10.13. 10.
- 10.14. Метод, который возвращает значение из атрибута класса, и при этом его не изменяет, называется методом-получателем. Метод, который сохраняет значение в атрибуте данных либо каким-либо иным образом изменяет значение атрибута данных, называется методом-мутатором.
- 10.15. Верхняя секция — это место для записи имени класса. Средняя секция содержит список полей класса. Нижняя секция — список методов класса.
- 10.16. Письменное описание объектов реального мира, участников и крупных событий, связанных с задачей.
- 10.17. Если вы адекватно понимаете природу задачи, которую пытаетесь решить, то вы можете составить описание проблемной области задачи самостоятельно. Если же вы не полностью понимаете природу задачи, то за вас это может сделать эксперт.
- 10.18. Сначала нужно идентифицировать именные группы (существительные, местоимения и именные словосочетания) в описании предметной области задачи. Затем уточнить полученный список, устранив повторы, удаляя элементы, в которых вы не нуждаетесь при решении задачи, элементы, представляющие объекты вместо классов, и элементы, представляющие простые значения, которые могут быть сохранены в переменных.
- 10.19. Сведения, которые класс обязан знать, и действия, которые класс обязан выполнять.
- 10.20. Что именно класс должен знать и что именно класс должен делать в контексте поставленной задачи?
- 10.21. Нет, не всегда.

Глава 11

- 11.1. Надкласс является общим, или родовым, классом, а подкласс — конкретизированным, или видовым, классом.
- 11.2. Когда один объект является конкретизированным вариантом другого объекта, между ними существует отношение классификации, или отношение "род — вид". Конкретизированный объект является "видом" общего, или родового, объекта.
- 11.3. Он наследует атрибуты всего надкласса.
- 11.4. Птица является надклассом, а канарейка — подклассом.
- 11.5. Я — овощ.
Я — картофель.

Глава 12

- 12.1. Рекурсивный алгоритм требует многократных вызовов метода. Каждый вызов метода требует выполнения нескольких действий. Эти действия включают выделение памяти под параметры и локальные переменные и для хранения адреса местоположения программы, куда поток управления возвращается после завершения метода. Все эти действия называются накладными расходами. В итеративном алгоритме, в котором используется цикл, такие накладные расходы не требуются.
- 12.2. Случай, в котором задача может быть решена без рекурсии.
- 12.3. Случай, в котором задача решается с использованием рекурсии.
- 12.4. Ситуация, когда он достигает базового случая.
- 12.5. В прямой рекурсии рекурсивный метод вызывает сам себя. В косвенной рекурсии метод А вызывает метод В, который вызывает метод А.

Глава 13

- 13.1. Компонент компьютера и его операционной системы, с которым пользователь взаимодействует.
- 13.2. Интерфейс командной строки, как правило, выводит подсказку или приглашение, а пользователь набирает команду, которая затем исполняется.
- 13.3. Программа.
- 13.4. Программа, которая реагирует на происходящие события, такие как нажатие пользователем кнопки.
- 13.5. а) Label — область, в которой показывается одна строка текста или изображение; б) Entry — область, в которой пользователь может набирать на клавиатуре одну строку входных данных; в) Button — кнопка, которая вызывает наступление действия при ее нажатии; г) Frame — контейнер, который может содержать другие виджеты.
- 13.6. Создается экземпляр класса Tk модуля tkinter.
- 13.7. Данная функция выполняется как бесконечный цикл до тех пор, пока не будет закрыто главное окно.
- 13.8. Метод pack() размещает виджеты в соответствующих позициях и делает их видимыми при выводе главного окна на экран.
- 13.9. Они будут расположены один под другим.
- 13.10. side='left'.
- 13.11.

```
self.label = tkinter.Label(self.main_window,
                           text='Привет, мир!',
                           borderwidth=3,
                           relief='raised')
```
- 13.12.

```
self.my_label.pack(ipadx=10, ipady=20)
```
- 13.13.

```
self.my_label.pack(padx=10, pady=20)
```
- 13.14.

```
self.my_label.pack(padx=10, pady=20, ipadx=10, ipady=10)
```

- 13.15. Для извлечения данных, которые пользователь ввел в элемент интерфейса `Entry`, используется метод `get()` этого элемента.
- 13.16. Это строковый литерал.
- 13.17. `tkinter`.
- 13.18. Любое значение, которое хранится в объекте `StringVar`, будет автоматически показано в элементе `Label`.
- 13.19. Для таких значений используются радиокнопки.
- 13.20. Для таких значений используются флаговые кнопки.
- 13.21. При создании группы элементов `RadioButton` их следует связать с одним и тем же объектом `IntVar`. При этом каждому элементу `RadioButton` присваивается уникальное целочисленное значение. При выборе одного из элементов `RadioButton` он сохраняет свое уникальное целочисленное значение в объекте `IntVar`.
- 13.22. С каждым элементом `Checkbutton` следует связать отдельный объект `IntVar`. При выборе элемента `Checkbutton` связанный с ним объект `IntVar` будет содержать значение 1. При снятии галочки с элемента `Checkbutton` связанный с ним объект `IntVar` будет содержать значение 0.
- 13.23.
- ```
self.listBox.insert(0, 'Январь')
self.listBox.insert(1, 'Февраль')
self.listBox.insert(2, 'Март')
```
- 13.24. По умолчанию высота равна 10 строкам, а ширина — 20 символам.
- 13.25.
- ```
self.listBox = tkinter.Listbox(self.main_window,
                                height=20, width=30)
```
- 13.26. 'Петр' хранится в индексной позиции 0, 'Павел' хранится в индексной позиции 1, 'Мария' хранится в индексной позиции 2.
- 13.27. Кортеж, содержащий индексы элементов, которые в данный момент выбраны в списке.
- 13.28. Элемент из списка удаляется путем вызова метода `delete()` виджета `Listbox` с передачей индекса элемента, который вы хотите удалить.
- 13.29. (0, 0).
- 13.30. (639, 479).
- 13.31. В элементе `Canvas` точка (0, 0) расположена в левом верхнем углу окна. В черепашьей графике точка (0, 0) расположена в центре окна. Кроме того, в элементе `Canvas` координаты `Y` увеличиваются по мере перемещения вниз экрана. В черепашьей графике координаты `Y` уменьшаются по мере перемещения вниз экрана.
- 13.32. а) `create_oval()`; б) `create_rectangle()`; в) `create_rectangle()`;
г) `create_polygon()`; д) `create_oval()`; е) `create_arc()`.

Глава 14

- 14.1.** Система управления базами данных (СУБД) — это программное обеспечение, специально разработанное для организованного и эффективного хранения, извлечения и управления большими объемами данных.
- 14.2.** Традиционные файлы, такие как текстовые файлы, непрактичны, когда необходимо хранить и обрабатывать большой объем данных. Многие компании хранят сотни тысяч или даже миллионы элементов данных в файлах. Когда традиционный файл содержит такое количество данных, простые операции, такие как поиск, вставка и удаление, становятся громоздкими и незэффективными.
- 14.3.** Разработчик должен знать только о том, как взаимодействовать с СУБД. СУБД выполняет фактическое чтение, запись и поиск данных.
- 14.4.** Стандартный язык для работы с системами управления базами данных.
- 14.5.** Приложение Python создает инструкции SQL в виде строк, а затем использует библиотечный метод для передачи этих строк в СУБД.
- 14.6.** `import sqlite3`
- 14.7.** а) *база данных* содержит данные, организованные в таблицы, строки и столбцы; б) *таблица* содержит набор связанных данных; данные, хранящиеся в таблице, организованы в строки и столбцы; в) *строка* содержит полный набор данных об одном элементе; данные, хранящиеся в строке, делятся на столбцы; г) *столбец* содержит отдельный фрагмент данных об элементе.
- 14.8.** 1) в; 2) г; 3) а; 4) д; 5) б.
- 14.9.** Первичный ключ — это столбец, который можно использовать для идентификации конкретной строки.
- 14.10.** Идентификационный столбец содержит уникальные значения, которые генерируются СУБД и обычно используются в качестве первичного ключа.
- 14.11.** Курсор — это объект, который имеет доступ к данным в базе данных и может манипулировать ими.
- 14.12.** После.
- 14.13.** Когда вы вносите изменения в базу данных, эти изменения фактически не сохраняются в ней до тех пор, пока вы их не зафиксируете.
- 14.14.** Метод `connect()` модуля `SQLite`.
- 14.15.** Будет создан файл базы данных.
- 14.16.** Объект `Cursor`.
- 14.17.** Метод `cursor()` объекта `Connection`.
- 14.18.** Метод `commit()` объекта `Connection`.
- 14.19.** Метод `close()` объекта `Connection`.
- 14.20.** Метод `execute()` объекта `Cursor`.

14.21. CREATE TABLE Book (Publisher TEXT, Author TEXT, Pages INTEGER, ISBN TEXT)

14.22. DROP TABLE Book

14.23. INSERT INTO Inventory (ItemID, ItemName, Price)
VALUES (10, "Циркулярная пила", 199.99)

14.24. INSERT INTO Inventory (ItemName, Price)
VALUES ("Зубило", 8.99)

14.25. cur.execute(''INSERT INTO Inventory (ItemName, Price)
VALUES (?, ?)'',
(name_input, price_input))

14.26. а) Account; 6) Id.

14.27. а) SELECT * FROM Inventory;

б) SELECT ProductName FROM Inventory;

в) SELECT ProductName, QtyOnHand FROM Inventory;

г) SELECT ProductName FROM Inventory WHERE Cost < 17.00;

д) SELECT * FROM Inventory WHERE ProductName LIKE "%ZZ".

14.28. Он позволяет определить, содержит ли столбец указанный шаблон символов.

14.29. Символ % используется в качестве подстановочного знака. Он представляет собой любую последовательность из нуля или более символов.

14.30. Вы можете использовать выражение ORDER BY.

14.31. Метод fetchall() возвращает результаты запроса SELECT в виде списка кортежей. Когда запрос возвращает только одно значение, метод fetchone() можно использовать для возврата кортежа с одним элементом, т. е. в индексной позиции 0.

14.32. UPDATE Products

SET RetailPrice = 4.99

WHERE Description LIKE "%Стружка"

14.33. DELETE FROM Products WHERE UnitCost > 4.0

14.34. Нет.

14.35. Нет.

14.36. INTEGER.

14.37. 100.

14.38. Если вы создадите столбец, который является INTEGER PRIMARY KEY (т. е. целочисленным первичным ключом) в таблице, то этот столбец станет псевдонимом для столбца RowID.

14.39. Составной ключ — это ключ, созданный путем объединения двух или более существующих столбцов.

14.40. `sqlite3.Error`.

14.41. Поскольку дубликаты данных отнимают место для хранения и могут привести к хранению в базе данных несогласованной и конфликтующей информации.

14.42. Внешний ключ — это столбец в одной таблице, который ссылается на первичный ключ в другой таблице.

14.43. Связь (или отношение) "один ко многим" означает, что для каждой строки в одной таблице может быть много строк в другой таблице, которые на нее ссылаются.

14.44. Связь (или отношение) "многие к одному" означает, что многие строки в одной таблице могут ссылаться на одну строку в другой таблице.

Предметный указатель

#

#, символ 46

>

>>>, подсказка 29

A

Ada, язык программирования 24

ASCII 18

B

BASIC, язык программирования 24

C

C#, язык программирования 24

C, язык программирования 24

C++, язык программирования 24

COBOL, язык программирования 24

E

ENIAC, компьютер 12

F

FORTRAN, язык программирования 24

F-строка 79

I

issuperset, метод 506

J

Java, язык программирования 25

JavaScript, язык программирования 25

N

None, встроенное значение 279

NULL, значение столбца 723

P

Pascal, язык программирования 24

pip, менеджер пакетов 825

Python, язык программирования 25

R

Ruby, язык программирования 25

S

SQL-инъекция 737

Structured Query Language (SQL) 719

◊ AVG 751

◊ COUNT 752

◊ CREATE TABLE 727

◊ CREATE TABLE IF NOT EXISTS 730

◊ DELETE 758

◊ DROP TABLE 731

◊ DROP TABLE IF EXISTS 731

◊ INSERT INTO 731

◊ LIKE 748

◊ MAX 752

◊ MIN 752

◊ ORDER BY 750

◊ PRIMARY KEY 764

◊ SELECT 739

◊ SUM 751

◊ UPDATE 754

◊ WHERE 745

◊ передача инструкции в Python 726

U

Unified Modeling Language (UML) 571

V

Visual Basic, язык программирования 25

А

- Аккумулятор 198
- Алгоритм 40
- Аппаратное обеспечение 10
- Аргумент 44, 243
 - ◊ именованный 250
 - ◊ передача:
 - по значению 250
 - по позиции 247
- Ассемблер 23
- Атрибут данных 529
 - ◊ приватный 530, 538
 - ◊ скрытый 529
 - ◊ экземпляра 548

Б

- База данных 720
 - ◊ закрытие соединения 725
 - ◊ извлечение данных 739
 - ◊ место хранения на диске 726
 - ◊ обработка исключения 765
 - ◊ открытие соединения 725
 - ◊ сортировка результатов запроса 750
 - ◊ сохранение изменений 725
 - ◊ таблица:
 - вставка значений переменных 735
 - выбор всех столбцов 743
 - добавление нулевых данных 734
 - добавление строк 731
 - извлечение данных по критерию 745
 - обновление нескольких столбцов 757
 - обновление строки 754
 - создание 727, 729, 730
 - удаление 731
 - удаление строки 758
 - число удаленных строк 760
 - число обновленных строк 757
- ◊ тип данных 721
- Байт 15
- Библиотека функций 257
- Бит 16
- Блок 230
 - ◊ инструкций 130
- Блок-схема 41
- Буфер 313

В

- Валидация, `ValueError` 347–349, 353
- Ввод 14
 - ◊ данных с клавиатуры 57
- Виджет 636
 - ◊ `Button` 636, 651

- ◊ `Canvas` 636, 691
- ◊ `Checkbutton` 636, 669
- ◊ `Entry` 636, 655
- ◊ `Frame` 636, 649
- ◊ `Label` 636, 639
- ◊ `Listbox` 636, 671
- ◊ `Menu` 636
- ◊ `Menubutton` 636
- ◊ `Message` 636
- ◊ `Radiobutton` 636, 665
- ◊ `Scale` 636
- ◊ `Scrollbar` 636
- ◊ `Text` 636
- ◊ `Toplevel` 636
- Включение в список 401
- Выборка 20
- Вывод 14, 76
 - ◊ выравнивание значений 85
- Выражение:
 - ◊ `finally` 357
 - ◊ булево 130
 - ◊ булево, составное 152
 - ◊ математическое 61
 - ◊ смешанное 72

Г

- График 412
 - ◊ заголовок 413
 - ◊ маркировка точек 418
 - ◊ настройка координат 415
- Графика черепашья 90, 160, 215

Д

- Данные:
 - ◊ атрибут 529
 - приватный 530, 538
 - скрытый 529
 - ◊ входные 43
 - ◊ дозапись в файл 321
 - ◊ запись в файл 313
 - ◊ сокрытие 529
 - ◊ цифровые 19
 - ◊ числовые, чтение и запись в файл 321
 - ◊ чтение из файла 309, 315
- Диаграмма:
 - ◊ круговая 424
 - заголовок 426
 - метка 425
 - цвет 427
 - ◊ столбчатая 420
 - заголовок 423
 - метки осей 423
 - цвет 422

Диск:

- ◊ USB 14
- ◊ жесткий 13

3

Запись 332

- ◊ добавление в файл 336
- ◊ изменение 340
- ◊ поиск в файле 338

Заполнение 643

Знак подстановочный 749

Значение строковое 45

И

Имя полностью определенное 817

Индекс 439

- ◊ в строковом значении 439
- ◊ отрицательный 439

Индексация 369, 409, 439

Инкапсуляция 529

Инструкция:

- ◊ del 385
- ◊ if 129
- ◊ if-elif-else 150
- ◊ if-else 137
 - вложенная 155
- ◊ import 257, 265
- ◊ return 267
- ◊ try/except 356
 - исполнение 348

Интегрированная среда разработки (IDLE)

- ◊ окно оболочки Python 806

Интерпретатор 26

Интерфейс:

- ◊ командной строки 633
 - ◊ пользователя 633
 - графический 634
- Иключение 60, 345
- ◊ EOFError 513
 - ◊ IndexError 369, 440
 - ◊ IOError 350, 351, 353, 358
 - ◊ KeyError 474, 476, 501
 - ◊ ValueError 348, 353, 354, 358, 383
 - ◊ ZeroDivisionError 358
 - ◊ необработанное 358
- Итерация 183

К

Карта памяти 14

Класс 531

- ◊ базовый 591

- ◊ идентификация 571

◊ идентификация обязанностей 577

◊ определение 533

◊ производный 591

◊ экземпляр 531

Ключ 472

◊ внешний 777

◊ первичный 722, 727, 761

◊ не целочисленный 763

◊ целочисленный 762

◊ составной 763, 764

◊ тип данных 473

Ключевое слово:

◊ global 254

◊ pass 240

Кнопка флаговая 668

Код:

◊ исходный 26

◊ повторное использование 228

Комментарий 46

◊ концевой 47

Компилятор 26

Конец файла 326

Конкатенация 74, 441

◊ f-строк 87

◊ неявная строковых значений 75

◊ списков 372

Консервация 512

Константа:

◊ глобальная 254

◊ именованная 89

Конфликт имен 818

Кортеж 408

◊ индексация 409

◊ преобразование в список 410

Л

Лексема 459

Лексемизация 460

Литерал:

◊ неформатированный 312

◊ отформатированный строковый 79

◊ числовой 54

Ловушка ошибок 206

М

Массив 367

Местозаполнитель 80

Метка сигнальная 202

Метод 313, 529

◊ __init__() 534, 593, 597

◊ __str__() 546

◊ add() 500

Метод (*прод.*):

- ◊ `append` 379
- ◊ `clear()` 480
- ◊ `close()` 512
- ◊ `create_arc()` 699
- ◊ `create_line()` 693
- ◊ `create_oval()` 697
- ◊ `create_polygon()` 704
- ◊ `create_rectangle()` 695
- ◊ `create_text()` 706
- ◊ `dict()` 479
- ◊ `difference()` 505
- ◊ `discard()` 501
- ◊ `endswith` 453
- ◊ `find` 453
- ◊ `get()` 481, 566, 655, 657
- ◊ `index` 381
- ◊ `insert` 382
- ◊ `intersection()` 504
- ◊ `isalnum()` 449
- ◊ `isalpha()` 449
- ◊ `isdigit()` 449
- ◊ `islower()` 449
- ◊ `isspace()` 450
- ◊ `issubset()` 506
- ◊ `isupper()`, 450
- ◊ `items()` 481
- ◊ `keys()` 482
- ◊ `lower()` 451, 452
- ◊ `lstrip()` 451
- ◊ `pack()` 640
- ◊ `pop()` 483
- ◊ `popitem()` 484
- ◊ `read` 315
- ◊ `readline` 326, 328
- ◊ `readlines` 399
- ◊ `remove` 383
- ◊ `remove()` 501
- ◊ `replace` 453
- ◊ `reverse` 384
- ◊ `rstrip` 320
- ◊ `rstrip()` 319, 451
- ◊ `set()` 661
- ◊ `sort` 383
- ◊ `split()` 458
- ◊ `startswith` 453
- ◊ `strip()` 451
- ◊ `symmetric_difference()` 505
- ◊ `union()` 503
- ◊ `update()` 500
- ◊ `upper()` 451, 452
- ◊ `values()` 485
- ◊ `write` 313

- ◊ `writelines` 397
- ◊ `геттер` 553
- ◊ `инициализации` 534
- ◊ `переопределение` 604
- ◊ `получатель` 553
- ◊ `приватный` 530
- ◊ `публичный` 530
- ◊ `сеттер` 553
- ◊ `сторонний` 825
- Микропроцессор 12
- Мнемоника 23
- Множества, разность 505
- Множество 498
 - ◊ `добавление элемента` 500
 - ◊ `количество элементов` 500
 - ◊ `объединение` 503
 - ◊ `перебор элементов` 502
 - ◊ `пересечение` 504
 - ◊ `проверка существования значения` 503
 - ◊ `разность симметричная` 505
 - ◊ `создание` 499
 - ◊ `удаление элемента` 501
- Модуль 257, 284, 817
 - ◊ `math` 257, 281
 - ◊ `pickle` 512, 558
 - ◊ `pyplot` 411
 - ◊ `random` 258, 265
 - ◊ `sqlite3` 719
 - ◊ `tkinter` 635
- Модуляризация 284
- Мусор 204

Н

- Надкласс 591
- Надмножество 506
- Наибольший общий делитель 624
- Накопитель 198
- Наследование 591

О

- Обработчик:
 - ◊ `исключений` 347
 - ◊ `ошибок` 206
- Объект 529
 - ◊ `файловый` 311
 - ◊ `хранение в словаре` 560
- Объект-исключение 354
- Операнд 61
- Оператор:
 - ◊ `!=` 133
 - ◊ `&` 504
 - ◊ `*` 457

- ◊ / 64
- ◊ // 64
- ◊ ^ 506
- ◊ | 504
- ◊ += 441
- ◊ <= 132, 506
- ◊ == 132
- ◊ >= 132, 507
- ◊ and 153
- ◊ in 377
- ◊ not 154
- ◊ or 153
- ◊ возвведения в степень 68
- ◊ деления 64
 - по модулю 68
- ◊ логический 152
- ◊ математический 61
- ◊ остаток от деления 68
- ◊ повторения 367
- ◊ приоритет 65
- ◊ присваивания 48
 - расширенный 200
- ◊ сравнения 130
- Операционная система 14
- Отношение "род — вид" 590
- Отображение 472
- Ошибка:
 - ◊ логическая 39
 - ◊ синтаксическая 27

П

- Пакет matplotlib 410
- Память оперативная 13
- Пара "ключ : значение" 472
 - ◊ как отображения ключа на значение 472
- Параметр 243
- Переключатель 665
- Переменная 47
 - ◊ глобальная 253
 - ◊ локальная 241
 - ◊ область действия 241
 - ◊ параметрическая 243
 - ◊ повторное присвоение значения 55
 - ◊ правила именования 51
 - ◊ целевая 188
- Пикセル 19, 691
- Подкласс 591
- Подмножество 506
- Подстрока 443
 - ◊ поиск 452
- Позиция считывания 317
- Поле 332
- Полиморфизм 604

- Пользователь конечный 61
- Последовательность 365
 - ◊ мутируемая 371
 - ◊ символьная 45
 - ◊ Фибоначчи 623
- Предметная область задачи 572
- Представление словарное 481
- Присваивание 48
 - ◊ кратное 484
- Проверка входных данных 205
- Программа 9
 - ◊ модуляризованная 227
 - ◊ обслуживающая 14
 - ◊ разработки 15
- Программирование:
 - ◊ объектно-ориентированное 528, 529
 - ◊ процедурное 528
- Программист 9
- Программное обеспечение 9
 - ◊ прикладное 15
 - ◊ системное 14
 - ◊ технические требования 40
- Проектирование программы, разбиение задачи 40
- Процедура 528
- Псевдоним 818

Р

- Радиокнопка 665
- Разделитель тысяч 82
- Разработка нисходящая 236
- Разряд 16
- Расконсервация 513
- Расходы накладные 618
- Расширение файла 311
- Режим интерактивный 44
- Рекурсия:
 - ◊ глубина 617
 - ◊ косвенная 621
 - ◊ наибольший общий делитель 624
 - ◊ накладные расходы 618
 - ◊ последовательность Фибоначчи 622
 - ◊ прямая 621
 - ◊ случай:
 - базовый 618
 - рекурсивный 618
 - ◊ суммирование списка 621
 - ◊ факториал 618
- Рисование:
 - ◊ дуги 699
 - ◊ многоугольника 704
 - ◊ овала (эллипса) 697
 - ◊ прямой 693
 - ◊ прямоугольника 695

C

- Сериализация 512
- Символ:
 - ◊ новой строки 76
 - ◊ пробельный 450
 - ◊ продолжения строки 73
 - ◊ разделитель значений 77
 - ◊ экранированный 77
- Синтаксис 25
- Система:
 - ◊ исчисления двоичная 16
 - ◊ координат экранная 691
 - ◊ управления базами данных (СУБД) 718
- Словарь 472
 - ◊ добавление значения 475
 - ◊ количество элементов 477
 - ◊ перебор ключей 479
 - ◊ получение:
 - значения 473, 483, 485
 - ключа 481, 482
 - элемента 481
 - ◊ проверка ключа 474
 - ◊ смешанные типы данных 477
 - ◊ создание 473
 - пустого 479
 - ◊ удаление:
 - значения 476
 - элементов 480
- Слово ключевое 25
- Состояние объекта 545
- Спецификатор:
 - ◊ вывода, порядок следования 86
 - ◊ формата 80, 820
- Список 188, 365
 - ◊ вложенный 404
 - ◊ возвращение ссылки из функции 392
 - ◊ вставка значения 382
 - ◊ вывод на экран 366
 - ◊ двумерный 404
 - ◊ добавление значения 379
 - ◊ значение:
 - максимальное 385
 - минимальное 385
 - ◊ инвертирование порядка элементов 384
 - ◊ конкатенирование 372
 - ◊ копирование 387
 - ◊ обход 370
 - ◊ параметров 247
 - ◊ передача в функцию 391
 - ◊ поиск:
 - значения 377
 - позиции элемента 381
 - ◊ преобразование в кортеж 410
- ◊ разные типы данных 366
- ◊ сортировка 383
- ◊ сохранение в файл 397
- ◊ строковых значений 366
- ◊ сумма значений 389
- ◊ суммирование элементов 621
- ◊ удаление элемента 383
 - по позиции 385
- ◊ усреднение значений 390
- ◊ целых чисел 366
- ◊ чтение из файла 399
- Сравнение строковых выражений 141
- Среднее арифметическое 66
- Срез 374
 - ◊ строки 443
- Степень точности 81
- Стиль горбатый 51
- Стока таблицы 720
- Столбец:
 - ◊ идентификационный 722
 - ◊ таблицы 720
 - NULL 723
 - RowID 761
 - обновление 757
- Строка:
 - ◊ длина 440
 - ◊ конкатенация 441
 - ◊ разбиение 458
 - ◊ символы:
 - в верхнем регистре 452
 - в нижнем регистре 452
 - ◊ сравнение нерегистрочувствительное 452
 - ◊ срез 443
 - ◊ таблицы:
 - обновление 754
 - удаление 758
 - число:
 - обновлений 757
 - удаленных 760
- Структура принятия решения:
 - ◊ с двумя альтернативными вариантами 136
 - ◊ с единственным вариантом 129
 - ◊ управляющая 128
- Схема:
 - ◊ вычислений укороченная 154
 - ◊ иерархическая 236
 - ◊ структурная 236

T

Таблица 720

- ◊ базы данных:
 - вставка значений переменных 735
 - выбор всех столбцов 743

- добавление:
 - нулевых данных 734
 - строк 731
- извлечение данных 739
 - по критерию 745
- обновление:
 - нескольких столбцов 757
 - строки 754
- создание 727, 729, 730
- удаление 731
 - строки 758
- число:
 - обновленных строк 757
 - удаленных строк 760
- ◊ ввод — обработка — вывод 271
- Тип данных 54
- ◊ строковый 55
- Токен 460
- Токенизация 460
- Трассировка 346

У

- Устройство:
- ◊ ввода 14
 - ◊ вывода 14
 - ◊ цифровое 19
 - ◊ хранения, вторичное 13
 - Утилита 14

Ф

- Файл:
- ◊ cookie 308
 - ◊ CSV 462
 - ◊ ввода 309
 - ◊ вывода 309
 - ◊ двоичный 310
 - ◊ закрытие 512
 - ◊ открытие 311, 513
 - ◊ с последовательным доступом 310
 - ◊ с произвольным доступом 310
 - ◊ с прямым доступом 310
 - ◊ текстовый 310

Факториал 618

Флаг 159

Флажок 668

Флеш-накопитель 14

Функция 43, 226

- ◊ bar 420, 422

- ◊ dump 558

- ◊ float 59

- ◊ format 820

- ◊ input 57

- ◊ int 59

- ◊ isinstance 609
- ◊ len 370, 440, 477, 500
- ◊ list 366, 410
- ◊ main, исполнение по условию 288
- ◊ max 385
- ◊ min 385
- ◊ open 311, 513
- ◊ pie 424
- ◊ plot 411, 412
- ◊ print 24, 44, 52, 76
- ◊ randint 258
- ◊ random 265
- ◊ random.seed 266
- ◊ randrange 265
- ◊ range 190
- ◊ str 321
- ◊ title 413
- ◊ tuple 410
- ◊ turtle.heading 160
- ◊ turtle.isdown 160
- ◊ turtle.isVisible 161
- ◊ turtle.pencolor 161
- ◊ turtle.pensize 162
- ◊ turtle.speed 162
- ◊ turtle.xcor 160
- ◊ turtle.ycor 160
- ◊ uniform 265
- ◊ xlim 415
- ◊ ylim 415
- ◊ агрегатная 751
- ◊ без возврата значения 228
- ◊ библиотечная 257
- ◊ булева 276
- ◊ вложенная 59
- ◊ вызов 230
- ◊ обратного вызова 651
- ◊ пустая 240
- ◊ рекурсивная 615
 - остановка 616
- ◊ с возвратом значения 256

Х

Ханойские башни 625

Ц

Центральный процессор 11

Цикл:

- ◊ for 368
 - обход строкового значения 437
 - чтение данных из файла 328

Цикл (*проц.*):

- ◊ while 180
- ◊ бесконечный 186
- ◊ вложенный 208
- ◊ разработки программы 38
- ◊ с условием 180
- ◊ со счетчиком 180
- Цифра двоичная 16

Ч

Черепашья графика 290

Черный ящик 257

Число:

- ◊ в научной нотации 83
- ◊ магическое 88, 89
- ◊ псевдослучайное 265
- ◊ с плавающей точкой 80
 - в процентах 82
- ◊ случайное 258
- ◊ целое 83
- ◊ ширина поля вывода 84

Чтение первичное 205, 327

Ш

Шаблон симольный 749

Шаг, величина 191

Шрифт 708

Э

Экземпляр класса 531

Элемент списка 365

Ю

Юникод 19

- ◊ латинское подмножество 811

Я

Язык:

- ◊ высокоуровневый 24
- ◊ машинный 21
- ◊ структурированных запросов 719

НАЧИНАЕМ ПРОГРАММИРОВАТЬ НА >>> PYTHON®

5-Е ИЗДАНИЕ

В книге изложены принципы программирования, с помощью которых вы приобретете навыки алгоритмического решения задач на языке Python, даже если у вас нет опыта программирования. Для облегчения понимания сути алгоритмов широко использованы блок-схемы, псевдокод и другие инструменты. Приведено большое количество сжатых и практических примеров программ. В каждой главе предложены тематические задачи с пошаговым анализом их решения. Отличительной особенностью издания является его ясное, дружественное и легкое для понимания изложение материала.

Книга идеально подходит для вводного курса по программированию и разработке программного обеспечения на языке Python.

- Краткое введение в компьютеры и программирование
- Ввод, обработка и вывод данных
- Управляющие структуры и булева логика
- Структуры с повторением и функции
- Файлы и исключения
- Списки и кортежи
- Строковые данные, словари и множества
- Классы и объектно-ориентированное программирование
- Наследование и рекурсия
- Функциональное программирование
- Программирование баз данных



ii3388276354

Тони Гэддис — ведущий автор всемирно известной серии книг «Начинаем программировать...» (Starting Out With) с двадцатилетним опытом преподавания курсов информатики в колледже округа Хейвид, шт. Северная Каролина, удостоен звания «Преподаватель года», лауреат премии «Педагогическое мастерство».



Электронный архив, содержащий рассмотренные в книге проекты и исходный код примеров, ответы на вопросы для повторения, решения задач по программированию, а также главу 15 «Принципы функционального программирования» можно скачать по ссылке <https://zip.bhv.ru/9785977568036.zip>, а также со страницы книги на сайте <https://bhv.ru>.



Pearson



ISBN 978-5-9775-6803-6



9 785977 568036

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru