3-е издание

Кольцов Д. М.

Python

Создаем программы и игры

- Базовый синтаксис языка
- Разработка (с примерами кода) простейших приложений
- Графика, ООП, кортежи, виджеты и многое другое
- Создание собственной игры!

НиТ
издательство
nit.com.ru

Кольцов Д. М.

PYTHON

СОЗДАЕМ ПРОГРАММЫ И ИГРЫ

3-е издание

"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973

Кольцов Д. М.

PYTHON. Создаем программы и игры. 3-е издание — СПб.: Издательство Наука и Техника, 2022. — 432 с., ил.

ISBN 978-5-907592-01-8

Данная книга позволяет уже с первых шагов создавать свои программы на языке **Python**. Акцент сделан на написании компьютерных игр и небольших приложений. Для удобства начинающих пользователей в книге есть краткий вводный курс в основы языка, который поможет лучше ориентироваться на практике. По ходу изложения даются все необходимые пояснения, приводятся примеры, а все листинги (коды программ) сопровождаются подробными комментариями.

Книга будет полезна как начинающим программистам, так и всем, кто хочет быстро и эффективно научиться писать программы на Python.

ISBN 978-5-907592-01-8

9 785907 592018

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

Содержание

ЧАСТЬ I.

ЗНАКОМСТВО С PYTHON

В первой части книги вы узнаете, что представляет собой язык Python и чем он интересен для программиста, как установить его на свой компьютер, как создавать комментарии в коде, как писать, редактировать, запускать и сохранять программы. Конечно же, будет рассмотрен и практический пример, связанный с выводом текста.

Глава 1.

ВВЕДЕНИЕ В PYTHON

1.1. Вкратце о Python

Python – одновременно мощный и простой язык программирования, разработанный Гвидо ван Россумом (Guido van Rossum) в 1991 году. С одной стороны, язык довольно молодой (тот же Си был разработан в 1972 году), с другой стороны, ему уже 30 лет и за это время его успели довести до того уровня, когда на нем можно написать проект любого масштаба, в том числе коммерческие приложения, работающие с очень важными данными.

Python – это высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Другими словами, писать программы на Python довольно просто, а код самих программ будет гораздо лучше восприниматься, чем в случае с другими языками программирования.

> **Примечание**. Создатель языка Гвидо ван Россум заявил, что название языка происходит от ТВ-шоу «Летающий цирк Монти Пайтона». Да, об этом написано в FAQ (http://docs.python.org/faq/general#why-is-it-called-python), но кто ж его читает!

Синтаксис ядра этого языка программирования минималистичен, однако, стандартная библиотека содержит множество полезных функций.

Python поддерживает несколько парадигм программирования, в том числе *структурное*, *объектно-ориентированное*, *функциональное*, *императивное* и *аспектно-ориентированное*. Это означает, что вы можете выбрать любой стиль программирования. Новичкам, создающим относительно несложные программы, подойдет функциональный стиль. А для серьезных проектов, как правило, выбирают объектно-ориентированное и/или аспектно-ориентированное программирование.

Python активно развивается. Новые версии выходят каждые 2-3 года. В этой книге мы использовали текущую версию 3.10.2 от 14.02.2022. Для запусков примеров из этой книги понадобится как минимум версия 3.0. Также стоит отметить, что начиная с версии 3.9, Python больше не работает в Windows 7.

1.2. Кросс-платформенность

Python портирован и работает почти на всех известных платформах — от карманных компьютеров и смартфонов до мейнфреймов. Существуют версии Python под Microsoft Windows, практически все варианты UNIX (включая FreeBSD и Linux), Plan 9, macOS и macOS X, iPhone OS 2.0 и выше, Palm OS, OS/2, Amiga, HaikuOS, AS/400 и даже OS/390, Windows Mobile, Symbian и Android.

Важно понимать, что программы, написанные на Python, независимы от платформы. Другими словами, вы можете написать программу, которая будет работать и на вашем macOS X и на Windows-компьютере приятеля и даже на твоем iPhone. Главное, чтобы на устройстве, на котором планируется запускать программу, был установлен Python.

1.3. Python – один из самых простых языков программирования

Когда-то давно программы писались с помощью переключения различных разъемов на компьютерах, которые занимали целые залы. Затем появились

перфокарты – специальные карточки, используемые для ввода программы, после – язык программирования Ассемблер, позволяющий манипулировать данными прямо в регистрах процессора. Это пример низкоуровневого языка. Даже самая простая программа на этом языке представляла довольно большой, сложный и непонятный непосвященному человеку кусок кода.

С появлением высокоуровневых языков, таких как Си, Java все изменилось. Такие языки программирования ближе к человеческому языку, чем к машинному. Python делает синтаксис еще проще. Его правила приближаются к английскому языку. Создание программ на Python настолько простой процесс, что о нем говорят как о «программировании со скоростью мысли». Все это позволяет повысить производительность труда программиста – программы на Python требуют меньше времени на разработку, чем программы на других языках программирования.

1.4. Популярность Python

Если вы раньше ничего не слышали о Python и думаете, что он непопулярен, то вы ошибаетесь. Его используют разработчики со всего мира, ним пользуются крупнейшие корпорации, такие как Google, NASA, Red Hat, Yahoo!, Xerox, IBM, Microsoft и др. Так, Google предпочитает C++, Java и Python[1], а Microsoft даже открыла Python Developer Center[2] .

Популярные программные продукты Yahoo, в том числе Django[3], TurboGears[4] и Zope[5] написаны на Python.

Некоторым начинающим программистам, которые только выбирают язык программирования, программы на Python могут показаться неказистыми. Порой кажется, что ничего серьезного на этом языке не разработаешь, и он больше приближен к сценариям оболочки, чем к полноценным языкам программирования, таким как C#.

Это заблуждение. Многие популярные игры от EA Games, 2K Games и Disney Interactive, написаны на Python:

https://gamedev.stackexchange.com/questions/5035/famous-games-written-in-python

1 https://www.quora.com/Which-programming-languages-does-Google-use-internally
2 https://azure.microsoft.com/en-us/develop/python/
3 https://ru.wikipedia.org/wiki/Django
4 https://ru.wikipedia.org/wiki/TurboGears
5 https://ru.wikipedia.org/wiki/Zope

Игра – это одно из самых сложных приложений, поскольку оно сочетает работу с графикой, музыкой, сложную логику и т.д.

Также на Python пишут инсталляторы для различных дистрибутивов Linux. Так, инсталлятор Ubuntu написан на Python (рис. 1.1). Очень много приложений с графическим интерфейсом для Ubuntu также написано на этом языке.

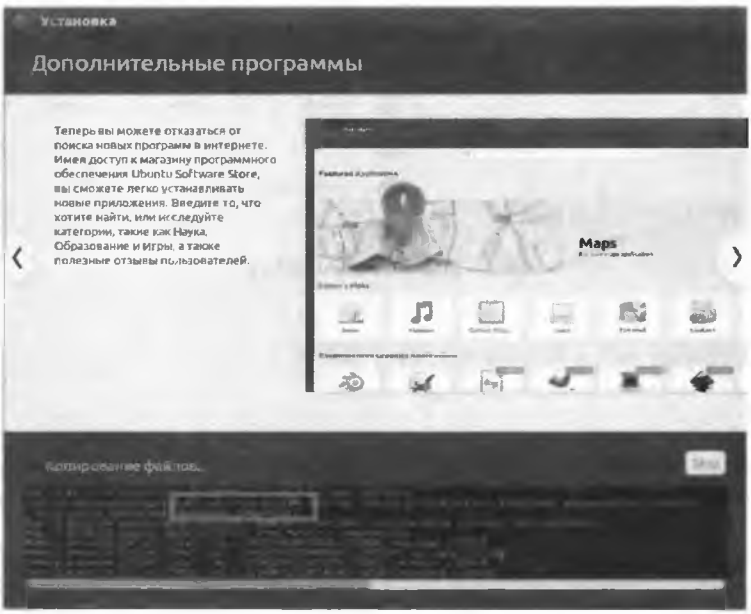


Рис. 1.1. Инсталлятор Ubuntu написан на Python

В соответствии с индексом TIOBE[6], Python занимает 1-ое место в мире! Только посмотрите на рейтинг популярности языков программирования. Пять лет назад, в 2017-ом году, данный язык занимал 5-ое место, а на первом месте был Java, после – семейство языков C. Сейчас же Python лидирует во всем мире.

1.5. Интегрируемость

Программист может легко интегрировать решения, написанные на других языках программирования, с программой на Python. Здесь все просто: вы можете использовать преимущества других языков программирования, например, производительность C/C++/C# и простоту написания кода Python.

6 https://www.tiobe.com/tiobe-index/

Feb 2022	Feb 2021	Change	Programming Language	Ratings	Change
1	3	^	Python	15.33%	+4.47%
2	1	v	C	14.08%	-2.26%
3	2	v	Java	12.13%	+0.84%
4	4		C++	8.01%	+1.13%
5	5		C#	5.37%	+0.93%
6	6		Visual Basic	5.23%	+0.90%
7	7		JavaScript	1.83%	-0.45%
8	8		PHP	1.79%	+0.04%
9	10	^	Assembly language	1.60%	-0.06%
10	9	v	SQL	1.55%	-0.18%

Рис. 1.2. Индекс TIOBE

1.6. Python поддерживает ICE

ICE (Internet Communications Engine) – объектная система, использующая механизм RPC. ICE создана под влиянием технологии CORBA. Но при этом Ice намного компактнее и проще, чем CORBA. Python поддерживает обе технологии – ICE и CORBA.

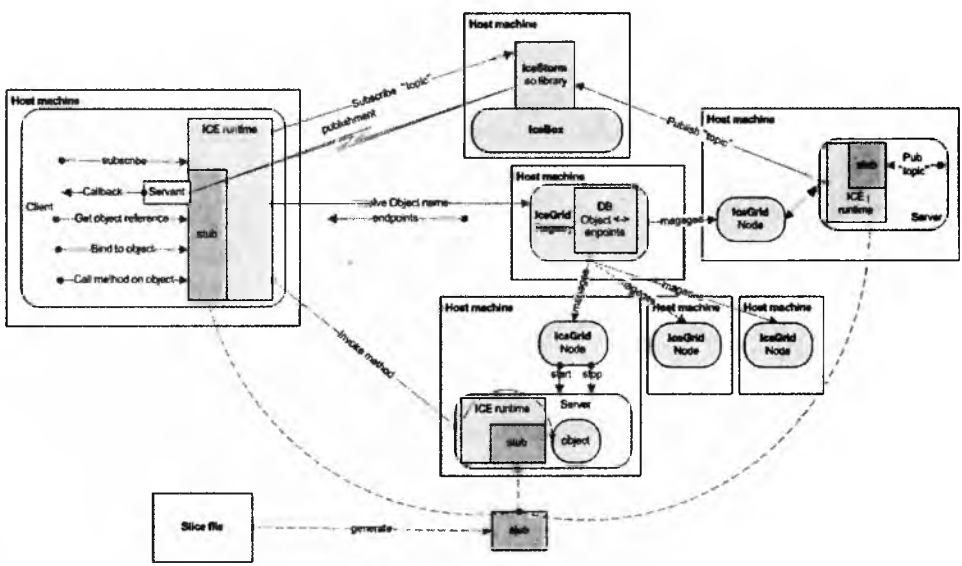
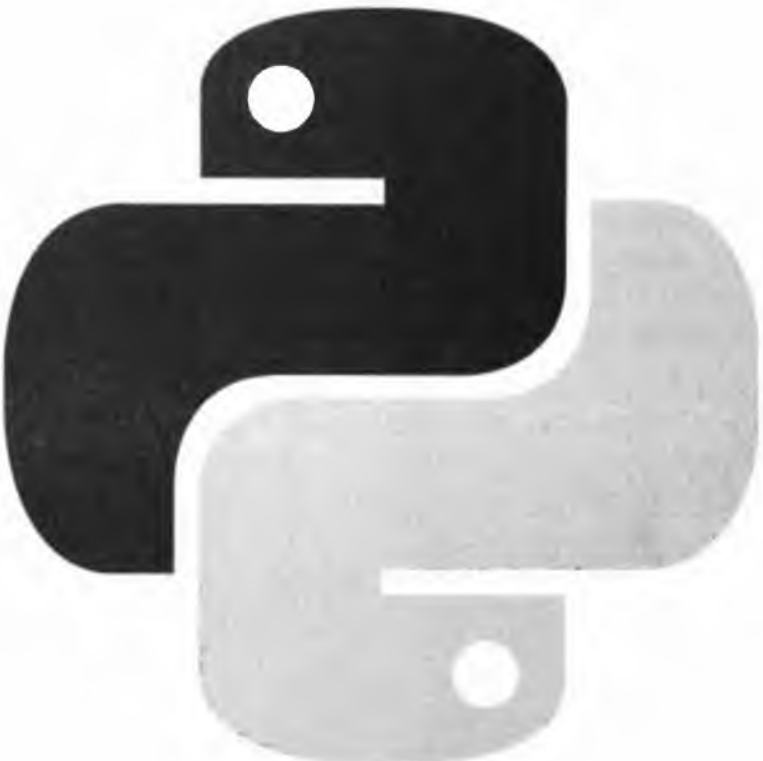


Рис. 1.3. Технология ICE

1.7. Сколько это стоит?

Интерпретатор Python абсолютно бесплатен и распространяется свободно. Чтобы использовать его, вам не нужно ничего платить, но зато вы можете абсолютно свободно продавать свои программы, написанные на Python – это позволяет лицензия, по которой распространяется этот интерпретатор.

Такие условия лицензии способствуют популярности этого языка программирования. Ведь для старта вам не нужно ничего и никому платить. Вы можете использовать Python, как для обучения программированию, так и для создания коммерческих программ. В отличие от других языков программирования, где за использование среды программирования нужно выложить кругленькую сумму, например, за Visual Studio придется выложить кругленькую сумму, а стоимость некоторых сторонних компонентов превышает 1500 евро (например, DevExpress). Конечно, сама Microsoft для физических лиц рекомендует бесплатные варианты, например, Visual Studio Community, а также редактор Visual Studio Code, который, кстати, можно с успехом использовать для написания скриптов на Python.



Глава 2.

УСТАНОВКА И НАЧАЛО РАБОТЫ

2.1. Загрузка Python

Обратите внимание, что загружать Python лучше всего с официального сайта – во избежание всякого рода вредоносного кода, который может быть внедрен на неофициальных сайтах в тех или иных целях.

Посетите сайт Python и скачайте версию, подходящую для вашей системы:

https://www.python.org/downloads/

2.2. Установка Python в Windows

В случае с Windows вам нужно скачать файл python-3.10.2-amd64.exe – это последняя на момент написания этих строк версия. Версии, начиная с 3.9, больше не поддерживают Windows 7 – ни с SP1, ни без него. Нужна как минимум Windows 10. Также больше нет поддержки 32-битных систем.

Примечание. Для удобства читателей, у которых стоят более старые версии Python, весь код примеров из данной книги будет работать в любой версии Python 3.x, кроме примеров из приложения 1, для выполнения которых необходима версия 3.5 или более новая.

Установка проходит просто – как любой другой программы (рис. 2.1). Вы можете нажать кнопку **Install Now** и тогда Python будет установлен в ваш пользовательский каталог, а можете нажать кнопку **Customize installation** и тогда у вас появится возможность выбора другого каталога.

Рис. 2.1. Первый экран инсталлятора Python

Далее нужно выбрать опциональные функции – нужно решить, нужна ли вам документация, тестовый набор и т.д. Можно установить все подряд, а потом разбираться. В случае обучения, когда вы еще точно не знаете, что вам понадобится, а что нет, лучше устанавливать полный комплект. Тем более, что места на диске полный набор занимает совсем немного.

Далее нужно выбрать расширенные опции. Лучше выбрать **Install for all users**, чтобы Python был доступен для всех пользователей. В этом случае Python будет установлен в каталог C:\Python310, а не в пользовательский каталог. Также лучше включить параметр **Add Python to environment variables**, который почему-то выключен по умолчанию.

Осталось только нажать кнопку **Install** для начала установки.

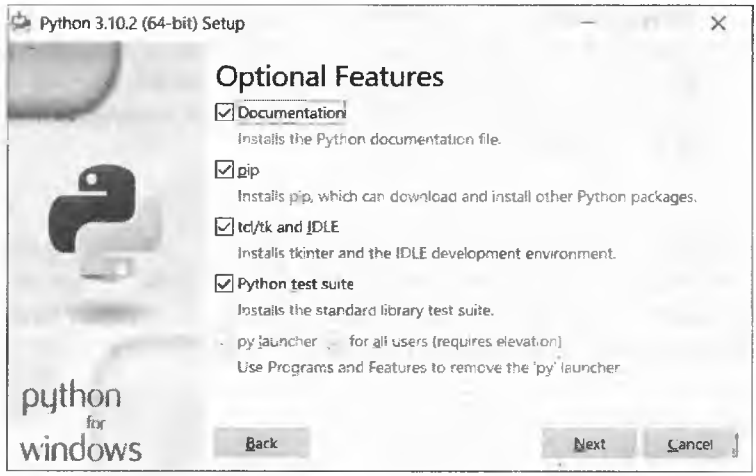


Рис. 2.2. Опциальные функции

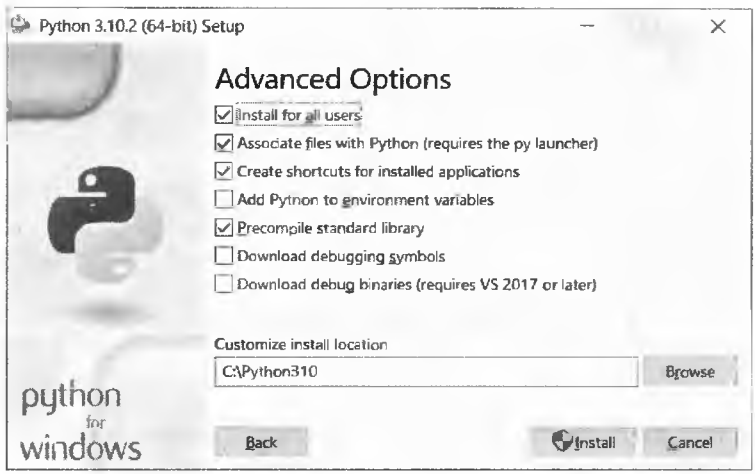


Рис. 2.3. Расширенные опции инсталлятора

Исключение разве что составляет каталог установки. Обычные программы по умолчанию устанавливаются в C:\Program Files\<Название программы>, а Python устанавливается в каталог C:\PythonNN, где NN – номер версии, например, C:\Python35.

При выборе компонентов Python я бы порекомендовал включить компонент **Add python.exe to Path**, который почему-то выключен по умолчанию. После этого вы сможете ввести просто команду *python* в командной строке без указания полного пути, например:

```
python <имя_py-файла>
```

В результате вы сможете запускать Python-программы без среды IDLE, что удобно, если вы хотите писать программы не только из соображений обучения языку программированию, но и для каких-то собственных нужд.

2.3. Установка Python в других операционных системах

На сайте Python есть возможность загрузить интерпретатор для других операционных систем. Так, на следующей страничке доступна версия для macOS X:

https://www.python.org/downloads/mac-osx/

Никаких ограничений по версиям Python нет: для macOS X доступны те же версии, в частности 3.10.2, что и для Windows.

Пользователям Linux повезло больше всех, так как в большинстве случаев интерпретатор Python установлен по умолчанию. Проверить его расположение можно командой:

```
denis@hosting:~$ which python
/usr/bin/python
```

В данном случае она сообщает, что Python установлен и находится в /usr/bin. Что делать, если Python не установлен? Учитывая, что Python в Linux используется всесторонне (многие модули настройки системы написаны на нем), то такая ситуация (когда этот интерпретатор не установлен) – что-то из области научной фантастики. Но если это так, то для его установки нужно установить пакет **python**, что можно сделать с помощью менеджера пакетов (см. документацию к вашему дистрибутиву).

Для других, более экзотических систем, можно найти информацию по установке Python на следующей страничке:

https://www.python.org/download/other/

На данный момент Python можно установить в следующих операционных системах:

- BeOS;
- IBM i;
- MorphOS;
- MS-DOS;
- OS/2;
- RISC OS;
- Series 60 (Nokia);
- VMS;
- Windows CE/Pocket PC;
- Solaris;
- HP-UX.

Глава 3.

СРЕДА IDLE

3.1. Запуск среды

Вместе с Python поставляется среда программирования IDLE Shell, в которой вы можете создавать свои Python-приложения. Запустить среду можно как обычное Windows-приложение, воспользовавшись средством поиска, встроенным в главное меню. На рис. 3.1 показана среда со стандартной темой оформления.

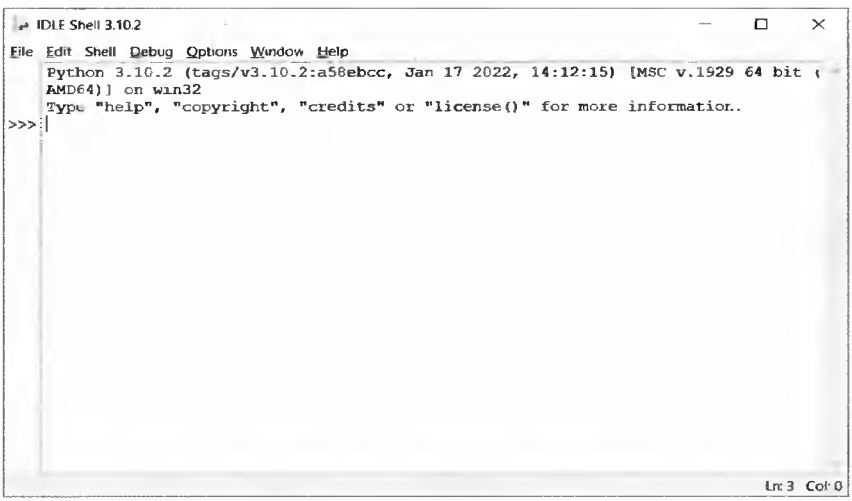


Рис. 3.1. Среда программирования IDLE

Давайте рассмотрим возможности этой среды.

3.2. Ввод программы и подсказки. Пример использования функции *print*

Несмотря на кажущуюся простоту IDLE Shell оснащена функцией подсказки при вводе кода – как у "полноценных" сред программирования (рис. 3.2). Введите название функции или метода, и вы получите подсказку по параметрам, которые нужно передать этой функции или методу.

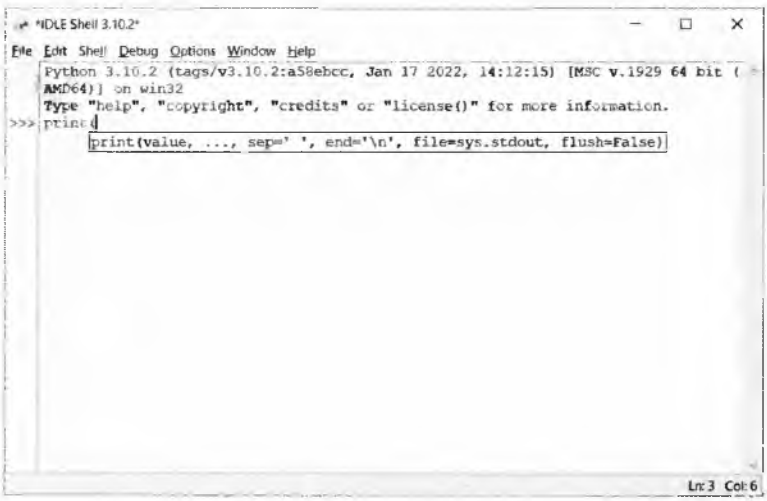


Рис. 3.2. Подсказка при вводе кода

Введите print("Hello") и нажмите **Enter**. Вы только что написали свою первую программу и получили результат ее выполнения – строчку "Hello".

Функция print() выводит на консоль текст, помещенный вовнутрь скобок и заключенный в кавычки. Если в скобках ничего нет, то будет выведена пустая строка.

> **Внимание!** Python чувствителен к регистру символов. Все названия функций пишутся строчными буквами, поэтому print() – это правильно, а Print() или PRINT() – нет.

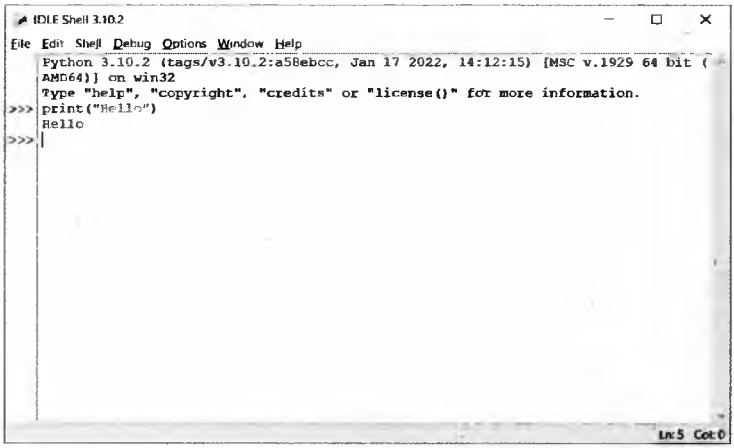


Рис. 3.3. Ваша первая программа

3.3. Генерируем ошибку. Пример сообщения об ошибке

Давайте посмотрим, как IDLE сообщает об ошибках. Для этого введите функцию Print("Hello") и нажмите **Enter**. Такой функции нет (см. примечание ранее), поэтому в ответ вы увидите:

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    Print("Hello")
NameError: name 'Print' is not defined
```

Python сообщает, что в *строке 1* в нашем модуле есть ошибка: название "Print" неопределенно. Другими словами, интерпретатор дает понять, что не знает, какую функцию ему вызвать, поскольку функции с названием "Print" нет.

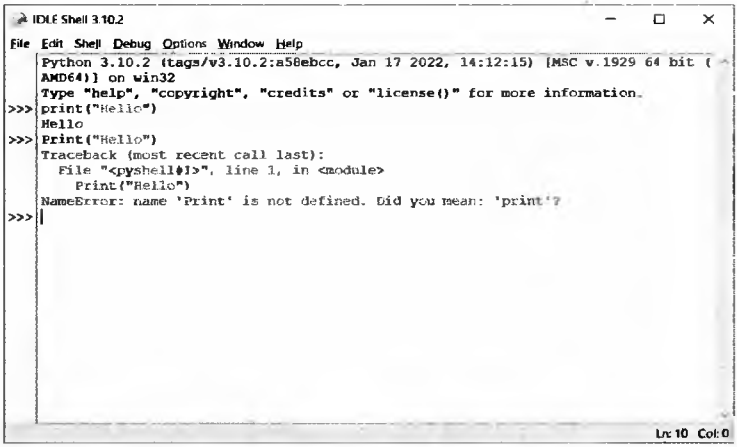


Рис. 3.4. Ошибка

3.4. Подсветка синтаксиса

Среда IDLE, как и ее старшие собратья, обеспечивает *подсветку синтаксиса*. Это означает, что слова на экране отображаются разными цветами. Такое явление упрощает понимание того, что именно вы вводите. Каждое слово по определенному правилу окрашивается в определенный цвет.

Так, имена функций окрашиваются в фиолетовый цвет, строковые значения – в зеленый. Если Python не может распознать, что именно вы ввели, то это слово никак не окрашивается. При светлой теме оформления оно будет напечатано черным, при темной – белым.

Результат работы программы интерпретатор выводит на экран шрифтом голубого цвета – так вы можете отделить код от его результата визуально.

Если вы проделали пример с функциями print() и Print(), то заметили, что первая функция была окрашена в фиолетовый, а вторая – никак не окрашена. Подсветка синтаксиса позволяет помочь понять, что вы что-то делаете не так и исправить ошибку еще до запуска кода. На первых порах подсветка синтаксиса станет вашим незаменимым помощником, ведь она делает код понятным с первого взгляда и позволяет сразу увидеть ошибки, если таковые имеются.

3.5. Изменение цветовой темы

Среда IDLE позволяет изменять цвет оформления элементов. Выбрать цветовую тему можно в настройках: Options, Configure IDLE. Далее нужно перейти на вкладку **Highlighting** и выбрать тему оформления. На рис. 3.5 показано, что выбрана тема IDLE Dark.

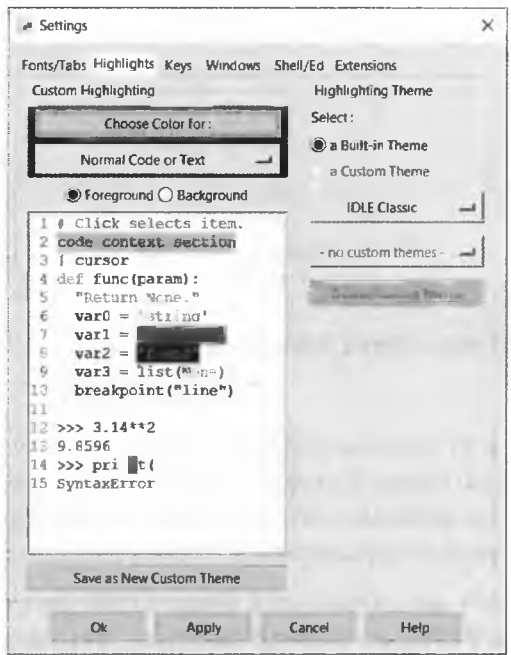


Рис. 3.5. Выбор темы оформления

При желании, вы можете создать собственную подсветку. Для этого выберите элемент, для которого вы хотите изменить цвет, а затем нажмите кнопку **Choose Color for**. Также можно воспользоваться переключателями **Foreground** (цвет шрифта) и **Background** (цвет фона).

Как только результат вам понравится, нажмите кнопку **Save as New Custom Theme**. Затем включите в группе **Highlighting Theme** переключатель **a Custom Theme** и выберите свою тему оформления.

На вкладке **Fonts/Tab** можно установить другие параметры шрифта – выбрать другую гарнитуру, установить другой размер. По умолчанию используется шрифт Courier New размером 10 пунктов.

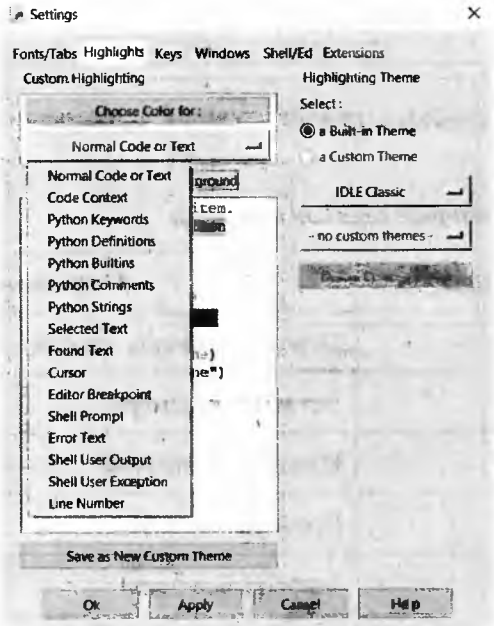


Рис. 3.6. Изменение цветовой темы

Рис. 3.7. Выбор другого шрифта

3.6. Горячие клавиши

Среда IDLE поддерживает горячие клавиши, приведенные в таблице 3.1.

Таблица 3.1. Некоторые горячие клавиши

Комбинация клавиш	Описание
Home	Переход к началу строки
Ctrl + I	Вставка по центру
Alt + U	Изменение отступа
Alt + X	Проверка модуля
Ctrl + Q	Закрыть все окна
Alt + F4	Закрыть окно (выход)
Ctrl + C, Ctrl + X, Ctrl + V	Стандартные клавиши управления буфером обмена
Ctrl + Backspace	Удалить слово слева
Ctrl + Delete	Удалить слово справа
Ctrl + F	Поиск
Alt + F3	Поиск в файлах
Ctrl + F3	Найти в выделенном
Ctrl + G	Найти снова
Alt + q	Форматировать параграф
Alt + g	Перейти к строке
Ctrl + C	Прервать выполнение (когда оно запущено)
Alt + m	Открыть модуль

Ctrl + N	Открыть новое окно
Ctrl + O	Открыть окно с файлом
Ctrl + P	Распечатать окно
Ctrl + Z	Отменить последнее действие
Esc	Снять выделение
Ctrl + H	Замена в тексте
F5	Запустить модуль на выполнение
Ctrl + F6	Перезапустить оболочку
Alt + Shift + S	Сохранить копию окна как файл
Ctrl + Shift + S	Сохранить окно как файл
Ctrl + A	Выделить все
Tab	Сделать отступ
F6	Просмотреть перезапуск

Внимание! Комбинации клавиш не работают? Обратите внимание на язык ввода. Он должен быть английским. Если вы нажимаете Ctrl + V, а язык ввода – русский, то Python видит комбинацию клавиш Ctrl + М и поэтому она не срабатывает! Подход не очень правильный, но какой есть. Именно поэтому я предпочитаю использовать сторонний редактор для редактирования Python-программ, а не редактор среды IDLE. В последний можно загрузить уже готовую программу, например, для ее отладки. В следующей главе будет показано, как настроить текстовый редактор Atom на выполнение сценариев Python прямо из редактора. Как редактор, Atom на голову, нет, на две головы выше, чем встроенный редактор IDLE, единственное, чего у него нет по умолчанию – возможности запуска программ Python, но это дело поправимое и в следующей главе будет показано, как это сделать.

3.7. Отладчик

В среду IDLE встроен отладчик, вызываемый командой **Debug, Debugger**. Процесс отладки приложения изображен на рис. 3.8. Управлять процессом отладки можно с помощью кнопок **Go, Step, Over, Out, Quit** – как и в других отладчиках.

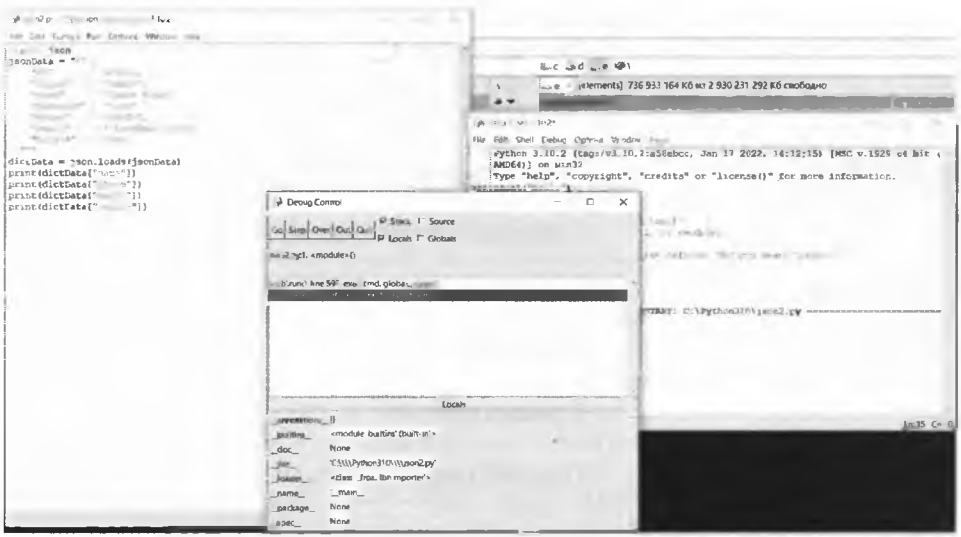


Рис. 3.8. Отладчик IDLE

В следующей главе мы рассмотрим создание программы, ее сохранение и запуск.

Глава 4.

ПРОГРАММА «ПРИВЕТ, МИР!»

Настало время написать вашу первую программу. Она будет очень проста и, по сути, мы уже "написали" ее в прошлой главе. Но нам нужно рассмотреть некоторые связанные с написанием программ моменты, а именно – сохранение программы, ее открытие, запуск вне оболочки IDLE и многое другое.

4.1. Сценарный режим

Ранее уже была рассмотрена простейшая программа, состоящая из всего одной функции print() – она выводила строку "Hello". Тогда мы нажимали **Enter** и интерпретатор сразу же выполнял нашу программу.

Это не очень удобно, если ваша программа состоит из более чем одной строки. Ведь первая строка будет выполнена еще до того, как вы введете следующую.

Именно поэтому все программы, состоящие из более чем одной строки, принято писать в сценарном режиме. Для этого выберите команду меню **File**, **New File** или нажмите Ctrl + N. Далее вы увидите окно сценарного режима, в котором можно вводить команды (операторы).

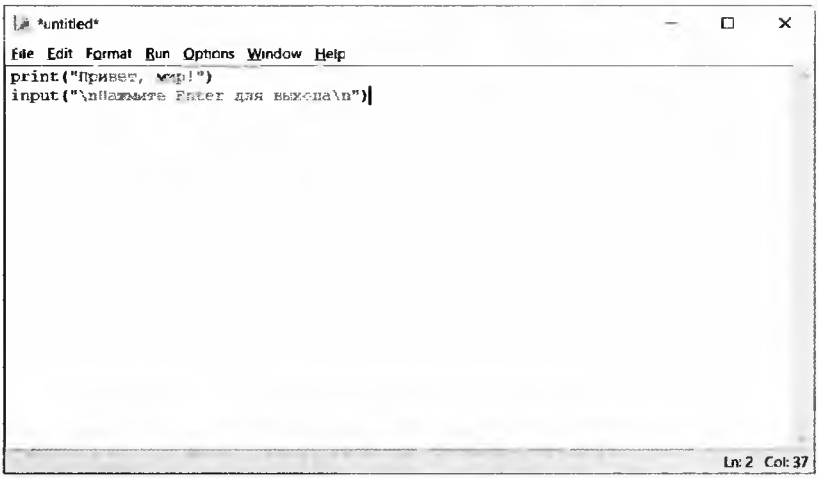


Рис. 4.1. Сценарный режим

В таком режиме вы можете ввести всю программу сразу. На рисунке 4.1 показано, что наша программа состоит из двух вызовов двух функций – сначала вызывается функция print() и мы выводим строку "Привет, Мир!" на экран, а затем – вызывается функция input(), которая обычно используется для организации пользовательского ввода в переменную, но поскольку мы переменную не указали, то выполнение программы будет приостановлено, пока пользователь не нажмет **Enter**. Если нужно сохранить пользовательский ввод в переменную, то конструкция будет выглядеть так:

```
userName = input('Как тебя зовут? ')
```

4.2. Комментарии в программе

Комментарии в программе начинаются символом #. Все, что после этого символа считается комментарием и не выполняется интерпретатором:

```
# Выводим приветствие
print("Привет, Мир!")
# Ждем, пока пользователь нажмет Enter
input("\nНажмите Enter для выхода\n") # Это тоже комментарий
```

Рис. 4.2. Комментарии в программе

Комментарии выделяются красным цветом шрифта. Комментарии можно писать не только, начиная с новой строки, но и в уже существующих строках, например:

```
min = 0       # минимум
max = 100     # максимум
```

Не забывайте комментировать программу по мере написания. Это существенно облегчит ее сопровождение – вам или другому программисту, когда со временем программу придется модифицировать. Даже если ее писали вы, то через несколько месяцев вы сами можете не вспомнить, для чего используется та или иная переменная или функция.

4.3. Сохранение и открытие программы

Понятно, что программу после ее написания нужно сохранить – если, конечно, вы хотите использовать ее повторно. Для этого выберите команду меню **File, Save** или нажмите Ctrl + S.

После этого откроется окно сохранения программы, в котором нужно будет ввести название файла. На рис. 4.3 показано, что было введено имя 1.py и программа была сохранена в каталоге Python по умолчанию.

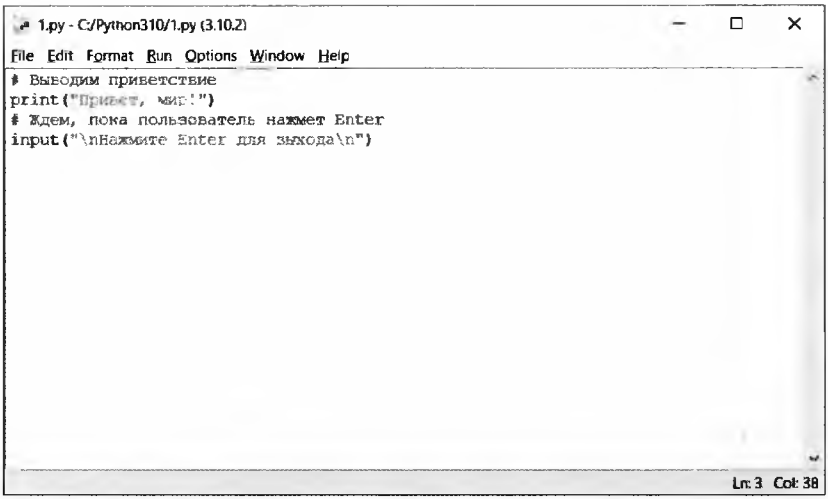


Рис. 4.3. Программа сохранена

Для открытия ранее сохраненной программы выберите команду **File, Open** или нажмите Ctrl + O. Далее выберите ру-файл и нажмите кнопку **Открыть**. После этого вы сможете отредактировать код программы или сразу запустить ее, нажав F5.

4.4. Запуск программы

Итак, ваша программа готова, и вы хотите ее запустить. Для этого просто нажмите **F5** или выберите команду меню **Run, Run Module**. Результат выполнения программы приведен на рис. 4.4.

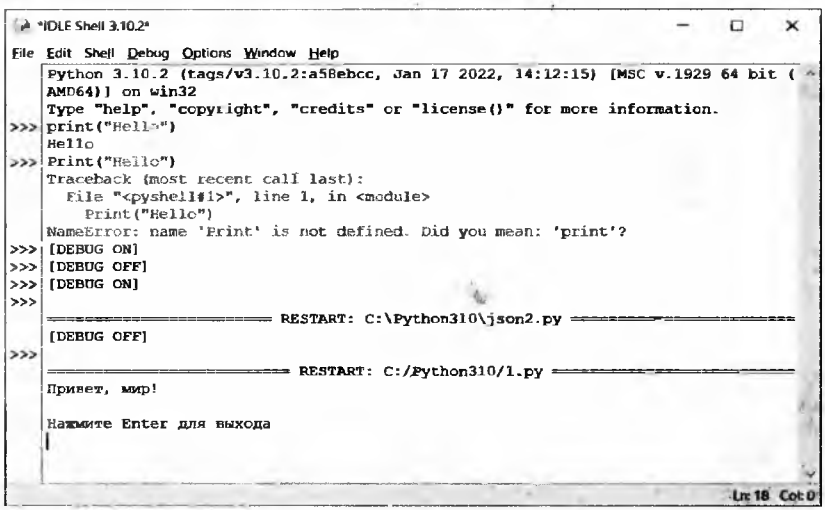


Рис. 4.4. Наша программа запущена

4.5. Запуск программы вне IDLE

Запустить Python-программу можно и просто в командной строке – без вызова IDLE. Для этого нужно вызвать Python так:

```
python <имя py-файла>
```

Например:

```
python 1.py
```

Такая команда будет работать только, если у вас Python добавлен в переменную окружения PATH. Если вы не выбрали соответствующий компонент при установке интерпретатора и не можете изменить PATH самостоятельно, нужно указывать полный путь к интерпретатору:

```
C:\Python34\python.exe 1.py
```

Рис. 4.5. Запуск Python-программы в Windows

В Linux можно создать командный исполняемый файл, который сразу будет выполняться системой. В начало вашего ру-файла добавьте строку:

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

Это путь к интерпретатору Python, узнать который можно командой *which python*. Обратите внимание, что между # и ! не должно быть пробелов и данная инструкция должна быть первой строкой в файле.

После этого данный файл нужно сделать исполняемым:

```
chmod +x <имя ру-файла>
```

Далее можно запускать просто сам ру-файл, например:

```
./1.py
```

Также можно поместить ру-файл (который сделан исполнимым) в каталог в пути поиска PATH и вызывать его без указания каталога:

```
sudo cp 1.py /usr/bin
1.py
```

Кажется, что в Linux гораздо больше свободы по запуску Python-файлов. Но в Windows можно создать bat-файл, запускающий тот или иной py-файл, например, вот содержимое файла 1.bat:

```
C:\Python34\python.exe C:\Python34\1.py
```

Мы указываем полный путь к Python и полный путь к программе. После чего мы сохраняем данную строку в файле 1.bat. Если поместить этот файл в путь поиска программ PATH и запускать просто так:

```
1
```

4.6. Использование нестандартного редактора

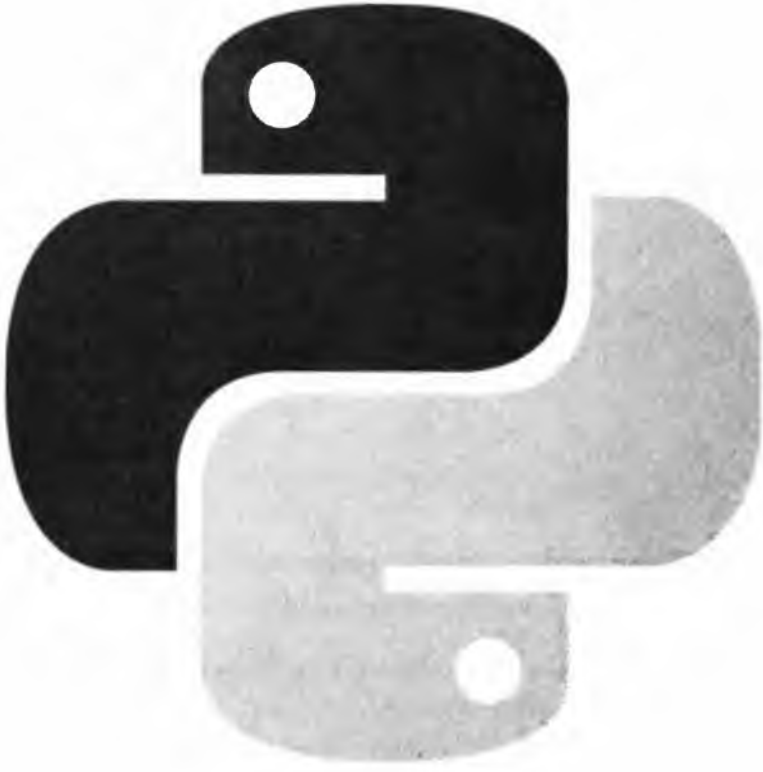
Редактор среды IDLE может показаться вам простоватым, если вы использовали другие редакторы. Никто не заставляет вас использовать штатный редактор. При желании можно выбрать любой другой редактор. На рис. 4.6 показан редактор Visual Studio Code, в который загружена программа, написанная на Python.

Рис. 4.6. Текстовый редактор Visual Studio Code

Visual Studio Code от Microsoft – очень простой и удобный редактор кода на все случаи жизни. При желании в нем можно не только писать код на Python, но и выполнять его, в том числе в режиме отладки. На рис. 4.7 показан как раз такой вариант – запуск скрипта в режиме отладки. Главное, чтобы при установке Python были "прописаны" переменные окружения – иначе среда попросту не сможет запустить интерпретатор Python.

Рис. 4.7. Запуск скрипта в режиме отладки

Ну чтож, вот мы и переходим к следующей части книги, в которой будут рассмотрены переменные и типы данных.



ЧАСТЬ II.

ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ

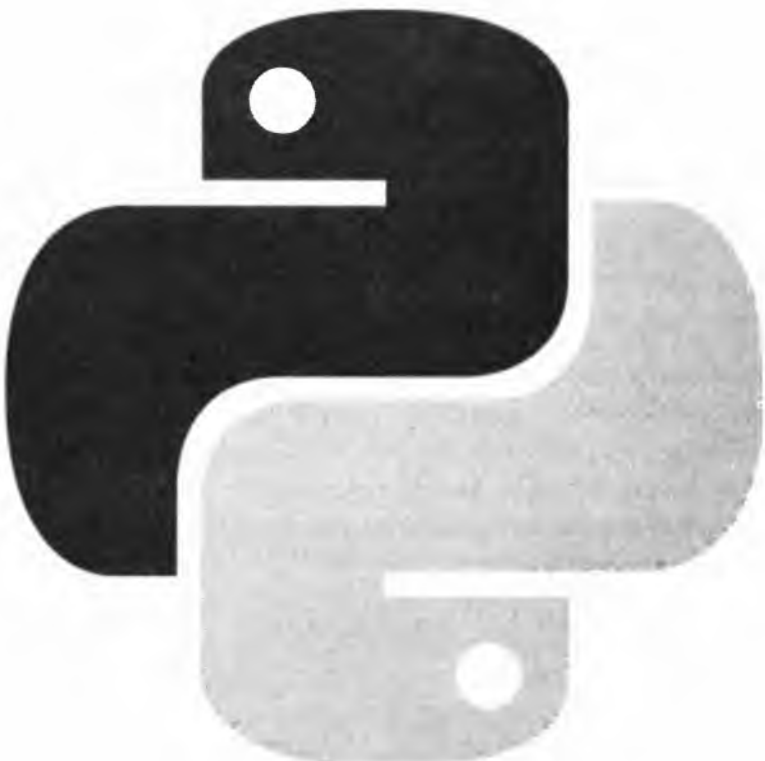
Теперь, когда вы знаете, как создать и запустить программу, можно приступить к изучению синтаксиса языка. Прежде, чем мы перейдем к рассмотрению переменных, вы должны знать, что типизация в Python динамическая, то есть тип переменной определяется только во время выполнения. Данное отличие может поначалу сбивать с толку, особенно, если вы программировали на языках, где требуется сначала указать тип переменной, а потом уже присваивать ей значение (C, Pascal и др.). В то же время языки с динамической типизацией тоже не редкость – типичный пример PHP, где не нужно объявлять тип переменной перед присвоением значения. Тип будет определен автоматически – по присвоенному значению. С одной стороны, так проще. С другой – это требует от программиста постоянно следить за данными, которые он присваивает переменной, поскольку одна и та же переменная в разные моменты времени может содержать данные разных типов.

В Python имеются встроенные типы: булевый, строка, Unicode-строка, целое число произвольной точности, число с плавающей запятой, комплексное число и некоторые др. Из коллекций в Python встроены: список, кортеж (неизменяемый список), словарь, множество и др. Все значения являются объектами, в том числе функции, методы, модули, классы.

Все объекты делятся на ссылочные и атомарные. К атомарным относятся ***int, long*** *(в версии Python 3 любое число является* ***int****, так как, начиная с этой версии, нет ограничения на размер), complex и некоторые другие.*

При присваивании атомарных объектов копируется их значение, в то время как для ссылочных копируется только указатель на объект, таким образом, обе переменные после присваивания используют одно и то же значение. Ссылочные объекты бывают изменяемые и неизменяемые. Например, строки и кортежи являются неизменяемыми, а списки, словари и многие другие объекты — изменяемыми. Кортеж в Python является, по сути, неизменяемым списком. Во многих случаях кортежи работают быстрее списков, поэтому если вы не планируете изменять последовательность, то лучше использовать кортежи.

Как показывает практика, большая часть обработки данных требует именно обработки строк, именно поэтому мы сначала рассмотрим строки, а затем другие типы данных. Однако строки – это настолько важная тема, что в следующей главе мы рассмотрим только основы строк, а детальной работе со строками посвящена вся четвертая часть книги.



Глава 5.

ОСНОВЫ РАБОТЫ СО СТРОКАМИ

В этой главе будет произведено знакомство со строками. Вы узнаете, как вводить и выводить строки, как осуществлять их слияние и повтор, как использовать внутри строк управляющие последовательности и в качестве бонуса познакомитесь с модулем WinSound, используемым для генерации звука в Windows.

5.1. Выбор кавычек

Строка – *это последовательность символов.* В Python строковые значения принято заключать в кавычки – двойные или одинарные. Компьютеру все равно, главное, чтобы использовался один и тот же тип открывающейся и закрывающейся кавычки, например:

```
print("Привет")
print('Привет')
```

Эти операторы выведут одну и ту же строку. При желании можно, чтобы строка содержала кавычки обоих типов:

```
print("Привет, 'мир'!")
```

Здесь внешние кавычки (двойные) используются для ограничения строкового значения, а внутренние выводятся как обычные символы. Внутри этой строки вы можете использовать сколько угодно одинарных кавычек.

Можно поступить и наоборот – для ограничения использовать одинарные кавычки, тогда внутри можно будет использовать сколько угодно двойных кавычек:

```
print('Привет, "мир"!')
```

Использовав кавычки одного типа в роли ограничителей, вы уже не сможете пользоваться ими внутри строки. Это целесообразно, ведь второе по порядку вхождение открывающей кавычки компьютер считает концом строки.

Функции print() можно передать несколько значений, разделив их запятыми:

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

Иногда такой прием используют, чтобы сделать код более читабельным.

5.2. Завершающий символ при выводе

По умолчанию функция print() после вывода заключенного в кавычки значения осуществляет перевод на новую строку. Это означает, что новый вывод print() будет произведен с новой строки. Иногда это не нужно делать. Задать новый завершающий символ можно с помощью параметра *end*:

```
print("Привет", end=" ")
print("мир")
```

Обратите внимание, что после слова "Привет" нет пробела. В качестве завершающего символа мы используем *пробел*. Благодаря этому, как только

будет напечатано слово "Привет" будет выведен пробел. Поскольку в качестве завершающего символа у нас используется символ, отличный от \n (это и есть перевод на новую строку), то строки "Привет" и "мир" будут находиться на одной строке, а не на разных. В итоге мы получим вывод:

```
Привет мир
```

5.3. Тройные кавычки. Пример вывода многострочного текста

Иногда есть большой фрагмент текста, который нужно вставить в программу как есть, и вывести в неизменном виде. Конечно, для этого лучше использовать файлы – записать текст в файл, потом в программе прочитать текст из файла и вывести его. Но не все программисты хотят усложнять программу – и если программа несложная, то можно весь код хранить в одном файле, чтобы ничего не потерялось.

Для вывода текста как есть используются тройные кавычки. В листинге 5.1 мы рассмотрим небольшую программу, выводящую инструкцию по использовании абстрактной программы. Когда мы будем изучать функции, данный вывод можно будет оформить в отдельную функцию, а пока разберемся, как работают тройные кавычки.

Листинг 5.1. Вывод многострочного текста прямо из программы

```
print("""
Использование: program -if <имя_файла> [-of <имя файла>]

-if: задает имя входного файла
-of: задает имя выходного файла. Если файл не задан, то используется
     стандартный вывод

""")
# Ждем, пока пользователь нажмет Enter
input("\nНажмите Enter для выхода\n") # Это тоже комментарий
```

Текст, заключенный между парой тройных кавычек (""" текст """), выводится как есть – сохраняется форматирование, переносы строк и т.д. Тройные кавычки существенно облегчают вывод многострочного текста (рис. 5.1).

Рис. 5.1. Использование тройных кавычек

5.4. Экранированные последовательности

Экранированные последовательности (они же Esc-последовательности, управляющие последовательности) позволяют вставлять в строки специальные символы. Esc-последовательность состоит из символа \ и еще какого-то символа. Самые популярные экранированные последовательности приведены в таблице 5.1.

Таблица 5.1. Экранированные последовательности

Последовательность	Описание
\n	Новая строка
\t	Табуляция (отступ)
\a	Звук системного динамика

\\	Выводит символ \
\'	Выводит одинарную кавычку (апостроф)
\"	Выводит двойную кавычку

5.5. Пример: программа "Имя v.0.0.1"

Цель данной небольшой программы продемонстрировать следующие вещи:

- Использование управляющих последовательностей
- Ввод данных с клавиатуры в переменную
- Вывод значения переменной

Код программы приведен в листинге 5.2.

Листинг 5.2. Код программы "Имя"

```
print("\t\t\t Имя v.0.0.1");
name = input("Как тебя зовут? ")
print("\a")
print("Привет, ", name)
```

Первая строка выводит три отступа (табуляция) перед названием программы. Посмотрите на рис. 5.2 – название программы появляется не с новой строки.

Вторая строка выводит приглашение ввода и читает ввод пользователя с клавиатуры. Ввод пользователя будет помещен в переменную *name*.

Третья строка должна генерировать звук системного динамика, но в Windows вместо этого выводится символ кружка и никакого звука не слышно. Цель звука – привлечение внимание пользователя – мы хотели заставить компьютер зазвучать перед тем, как поздороваемся с пользователем.

Последняя строка выводит приветствие – строку "Привет " и введенное пользователем имя.

Рис. 5.2. Программа имя в действии

5.6. Пример: генерирование звука в Windows. Программа "Имя v.0.0.2"

Предыдущий пример работал хорошо за исключением генерирования звука. Мало того, звука системного динамика не было, так еще и выводился ненужный нам символ. Можно было бы просто удалить третью строку, но мы легких путей не ищем.

Звук в Windows можно сгенерировать с помощью модуля **winsound**. Нужно импортировать этот модуль с помощью оператора *import*, задать частоту (в Герцах) и длительность (в миллисекундах) звучания звука, а затем вызвать метод *Beep*. Все очень просто. Вторая версия программы "Имя" приведена в листинге 5.3.

Листинг 5.3. Программа "Имя" с модулем *winsound*

```
print("\t\t\t Имя v.0.0.2\n");
name = input("Как тебя зовут? ")

import winsound
```

```
Freq = 2500                 # частота 2500 Гц
Dur = 1000                  # длительность 1000 мс = 1 с
winsound.Beep(Freq,Dur)

print("Привет, ", name)
```

Что изменилось в этой программе? Прежде всего, вместо print("\a") добавлено целых 4 строчки, обеспечивающих звук. Вы можете изменить частоту и длительность по своему усмотрению. В остальном – это та же программа (если не считать изменения номера версии и добавленного перехода на новую строку в первой строчке – \n).

5.7. Конкатенация строк

Сцепление, слияние или ***конкатенация строк*** – *это процесс объединения нескольких строковых значений в одну строку*. Для сцепления используется оператор +.

Рассмотрим небольшой пример:

```
s1 = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. "
s2 = "Proin porta, libero sit amet porttitor tempor, sapien diam vulputate "
s3 = "libero, quis porttitor nulla leo auctor felis."

s4 = s1 + s2 + s3
print(s4)

print("\n\n\n")

print(s1 + s2 + s3)
```

Итак, у нас есть три строки. Мы соединяем их с помощью оператора + и помещаем в строку s4.

После этого мы выводим строку s4. Далее мы выводим результат слияния строк прямо в print() – он будет таким же. Поэтому если вам нужно только вывести объединение строки (или передать его какой-то другой функции), то незачем использовать еще одну переменную и зря расходовать память.

Рис. 5.3. Результат слияния строк

5.8. Повторения строк

Не знаю, как вам, но мне в нашей программе **Имя** так и хотелось под названием вывести примерно 80 звездочек (*) – так будет немного красивее. Но как это сделать, ведь с циклами мы еще не знакомы. Выход есть – это повторы строк в Python. 80 звездочек тоже не хочется вводить – ведь их придется считать и что делать, если завтра мне захочется не 80, а 60 звездочек?

Специально для этих целей в Python есть операция повторения строки. Для ее обозначения используется символ *, после которого следует количество повторений. Код приведен в листинге 5.4, а результат выполнения – на рис. 5.4.

Листинг 5.4. Программа "Имя v.0.0.3"

```
print("\t\t\t Имя v.0.0.3");
print("*" * 80)
name = input("Как тебя зовут? ")

import winsound
Freq = 2500                 # частота 2500 Гц
Dur = 1000                  # длительность 1000 мс = 1 с
```

```
winsound.Beep(Freq,Dur)

print("Привет, ", name)
```

Рис. 5.4. Выведено 80 звездочек под названием программы

5.9. Строковые методы

Ранее было отмечено, что переменных всех типов являются объектами. А поскольку есть объекты, то есть и методы (табл. 5.2)

Таблица 5.2. Строковые методы

Метод	Описание
upper()	Возвращает строку, символы которой приведены к верхнему регистру. Исходная строка не изменяется
lower()	Возвращает строку, символы которой приведены к нижнему регистру. Исходная строка не изменяется

swapcase()	Возвращает строку, в которой регистр символов обращен
capitalize()	Возвращает строку, в которой первая буква прописная, остальные – строчные
strip()	Возвращает строку, из которой убраны все пробельные символы (пробелы, символы пустых строк, табуляция) в начале и в конце
replace(old, new [,max])	Возвращает строку, в которой вхождения строки *old* заменены строкой *new*. Необязательный параметр *max* устанавливает наиболее возможное количество замен
find(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
rfind(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
index(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError
rindex(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает ValueError
split(символ)	Разбиение строки по разделителю
isdigit()	Состоит ли строка из цифр
isalpha()	Состоит ли строка из букв
isalnum()	Состоит ли строка из цифр или букв
islower()	Состоит ли строка из символов в нижнем регистре

isupper()	Состоит ли строка из символов в верхнем регистре
isspace()	Состоит ли строка из неотображаемых символов (пробел, символ перевода страницы ('\f'), "новая строка" ('\n'), "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и "вертикальная табуляция" ('\v'))
istitle()	Начинаются ли слова в строке с заглавной буквы
upper()	Преобразование строки к верхнему регистру
lower()	Преобразование строки к нижнему регистру
startswith(str)	Начинается ли строка S с шаблона str
endswith(str)	Заканчивается ли строка S шаблоном str
join(список)	Сборка строки из списка с разделителем S
center(width, [fill])	Возвращает отцентрованную строку, по краям которой стоит символ fill (пробел по умолчанию)
count(str, [start],[end])	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
expandtabs([tabsize])	Возвращает копию строки, в которой все символы табуляции заменяются одним или несколькими пробелами, в зависимости от текущего столбца. Если TabSize не указан, размер табуляции полагается равным 8 пробелам

lstrip([chars])	Удаление пробельных символов в начале строки
rstrip([chars])	Удаление пробельных символов в конце строки
strip([chars])	Удаление пробельных символов в начале и в конце строки
partition(шаблон)	Возвращает кортеж, содержащий часть перед первым шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий саму строку, а затем две пустых строки
rpartition(sep)	Возвращает кортеж, содержащий часть перед последним шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий две пустых строки, а затем саму строку
swapcase()	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
title()	Первую букву каждого слова переводит в верхний регистр, а все остальные в нижний
zfill(width)	Делает длину строки не меньшей width, по необходимости заполняя первые символы нулями
ljust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом fillchar
rjust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя первые символы символом fillchar
*format(*args, **kwargs)*	Форматирование строки

Рассмотрим примеры использования строковых методов:

```
test = "привет"
print(test.upper())          # выведет ПРИВЕТ
print(test)                  # выведет привет
test = test.upper()
print(test)                  # выведет ПРИВЕТ
print(test.lower())          # выведет привет
test = test.swapcase()       # test = "привет"
print(test.capitalize())     # выведет Привет

test = "У Саши есть 10 арбузов"
print(test.replace("арбузов", "яблок"))
```

Обратите внимание, что строковые методы не изменяют строку, к которой они применяются. Они возвращают видоизмененную копию исходной строки. Если вам нужно изменить исходную строку, значит, вам нужно присвоить получившийся результат в исходную переменную.

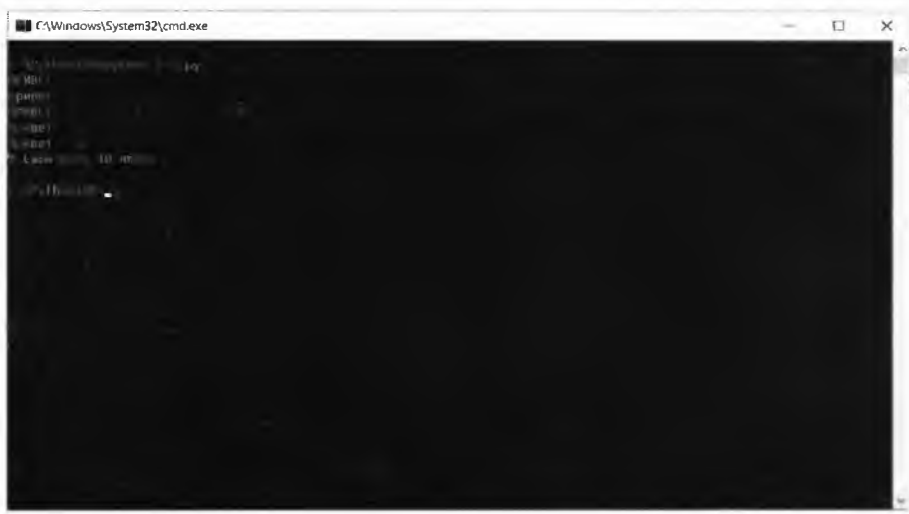


Рис. 5.5. Результат применения строковых методов

На этом мы заканчиваем знакомство со строками. Но мы еще поговорим о них в другой части этой книги, а пока переходим к работе с числами.

Глава 6.

РАБОТА С ЧИСЛАМИ

Числа – немаловажный элемент любой компьютерной программы. Можно сказать, что ни одна мало-мальски полезная программа не обходится без применения чисел. Даже в нашей, по сути, бесполезной программе "Имя" мы использовали числа – для задания количества повторов символов, частоты и продолжительности звучания.

6.1. Числовые типы данных

В Python доступно несколько числовых типов данных. На практике чаще всего используются два типа – *целые* (*int*) и *дробные* (*float*) числа. У целых чисел отсутствует дробная часть – думаю, это понятно.

6.2. Математические операторы

В Python поддерживаются математические операторы, приведенные в таблице 6.1.

Таблица 6.1. Математические операторы в Python

Оператор	Действие
+	Сложение

-	Вычитание
*	Умножение
/	Обычное деление
//	Деление с остатком
%	Остаток от деления

С помощью Python и математических операторов вы можете превратить свой компьютер в очень дорогой калькулятор. Думаю, всем понятно, как работают эти операторы, поскольку все это – курс математики из младших классов.

Рассмотрим небольшой пример:

```
print("5 + 4 = ", 5 + 4)
print("5 - 4 = ", 5 - 4)
print("5 * 4 = ", 5 * 4)
print("5 / 4 = ", 5 / 4)
print("5 // 4 = ", 5 // 4)
print("5 % 4 = ", 5 % 4)
```

Результат выполнения этого примера приведен на рис. 6.1.

Рис. 6.1. Использование математических операторов

6.3. Пример: вычисление времени в пути

Напишем небольшую программу, вычисляющую время автомобиля в пути. Пользователь должен будет ввести расстояние, которое нужно проехать, а также планируемую среднюю скорость автомобиля.

Листинг 6.1. Вычисление времени в пути

```
dist = 0       # Расстояние, которое нужно проехать
speed = 0      # Средняя скорость авто, км /ч

dist = int(input("Расстояние, которое нужно проехать: "))
speed = int(input("Планируемая средняя скорость: "))

time = dist * 60 / speed

print("Будет затрачено ", time, " минут")
```

Посмотрим, что есть в нашей программе. Первым делом мы инициализируем две переменные – *dist* и *speed*. Python не требует обязательной инициализации переменной. Мы сделали это лишь затем, чтобы добавить комментарий и знать, для чего используется та или иная переменная.

Далее мы получаем расстояние и среднюю скорость:

```
dist = int(input("Расстояние, которое нужно проехать: "))
speed = int(input("Планируемая средняя скорость: "))
```

Обратите внимание: мы используем преобразование типа и явно указываем, что прочитанное значение должно быть типа *int*.

Затем мы вычисляем время движения автомобиля по формуле:

```
time = dist * 60 / speed
```

60 здесь – количество минут в одном часе. После того, как время вычислено, мы его выводим.

Рис. 6.2. Результат работы программы

6.4. Пример: вычисление расхода топлива

Данный пример демонстрирует работу с дробными числами. Ранее мы вычисляли время в пути и вводили два целых параметра. Теперь мы будем также вводить два параметра, но они с большей долей вероятности могут быть дробными.

Листинг 6.2. Вычисления расхода топлива

```
consum = 0          # Средний расход 10.5 л/100 км
dist = 0            # Расстояние, км

consum = float(input("Введите средний расход л./100 км: "))
dist = float(input("Введите расстояние, км: "))

result = consum * dist / 100

print("Будет затрачено ", result, " литров")
```

Принцип программы такой же, как в предыдущем случае, но мы хотим получить дробные значения, поэтому мы используем функцию float(), которая приводит строковое значение к дробному.

Внимание! Обратите внимание, что в качестве разделителя целой и дробной части используется точка, а не запятая! То есть, если вы введете 10.5, программа будет работать, а если вы введете 10,5, то получите сообщение об ошибке:

```
Traceback (most recent call last):
  File "C:/Python310/9.py", line 4, in <module>
    consum = float(input("Введите средний расход л./100 км: "))
ValueError: could not convert string to float: '10,5'
```

Данное сообщение говорит о том, что невозможно конвертировать строковое значение "10,5" в float-значение.

6.5. Выбор правильного типа данных

В предыдущих примерах мы явно применяли int() и float() для приведения прочитанного с ввода значения к числовому типу. А что будет, если не выполнять приведения типа? Давайте посмотрим. Напишем программу, подсчитывающую стоимость содержания автомобиля.

Листинг 6.3. Стоимость содержания автомобиля

```
service = input("Введите стоимость технического обслуживания: ")
fuel = input("Стоимость топлива: ")
tax = input("Налоги и пошлины: ")
tuning = input("Тюнинг: ")
insurance = input("Стоимость страховки: ")

total = service + fuel + tax + tuning + insurance

print("Всего: ", total)
```

Вывод изображен на рис. 6.4. Явно не то, что мы хотели. По умолчанию все введенные значения считаются строковыми, и интерпретатор просто склеил строки в одну большую строку.

Именно поэтому нам нужно явно указывать тип прочитанного значения. Исправим ошибку (рис. 6.5).

Рис. 6.4. Стоимость содержания автомобиля. Ошибка!

```
service = float(input("Введите стоимость технического обслуживания: "))
fuel = float(input("Стоимость топлива: "))
tax = float(input("Налоги и пошлины: "))
tuning = float(input("Тюнинг: "))
insurance = float(input("Стоимость страховки: "))

total = service + fuel + tax + tuning + insurance

print("Всего: ", total)
```

Рис. 6.5. Стоимость содержания автомобиля. Правильная версия

Теперь, думаю, вы понимаете, зачем мы использовали int() и float() в предыдущих примерах.

6.6. Функция bit_count()

Начиная с Python 3.10, чтобы посчитать количество битов в двоичном представлении целого числа, можно вызвать int.bit_count(). Функция известна как *Population Count (popcount)*:

```
value = 42
print(bin(value))
# '0b101010'
print(value.bit_count())
# 3
```

Эту функцию несложно реализовать самому. Но наличие разных полезных маленьких функций — одна из причин, по которой Python так популярен: на первый взгляд все доступно из коробки.

6.7. Программа "Калькулятор"

Разработаем простейший калькулятор, то есть программу, умеющую выполнять над двумя вещественными числами арифметические операции (сложение, вычитание, умножение, деление) и завершающуюся по желанию пользователя.

Наш калькулятор будет работать так:

1. Запустить бесконечный цикл. Выход из него осуществлять с помощью оператора *break*, если пользователь вводит определенный символ вместо знака арифметической операции.
2. Если пользователь ввел знак, который не является ни знаком арифметической операции, ни символом-"прерывателем" работы программы, то вывести сообщение о некорректном вводе.
3. Если был введен один из четырех знаков операции, то запросить ввод двух чисел.

4. В зависимости от знака операции выполнить соответствующее арифметическое действие.
5. Если было выбрано деление, то необходимо проверить, не является ли нулем второе число. Если это так, то сообщить о невозможности деления.

Код программы приведен в листинге 6.4.

Рис. 6.6. Калькулятор в действии

Листинг 6.4. Калькулятор

```
print("*" * 15, " Калькулятор ", "*" * 10)
print("Для выхода введите q в качестве знака операции")
while True:
   s = input("Знак (+,-,*,/): ")
   if s == 'q': break
   if s in ('+','-','*','/'):
      x = float(input("x="))
      y = float(input("y="))
      if s == '+':
         print("%.2f" % (x+y))
      elif s == '-':
         print("%.2f" % (x-y))
      elif s == '*':
         print("%.2f" % (x*y))
      elif s == '/':
```

```
        if y != 0:
          print("%.2f" % (x/y))
        else:
          print("Деление на ноль!")
  else:
     print("Неверный знак операции!")
```

Посмотрим на вывод программы. Обратите внимание, как она реагирует на неверный ввод, например, если введено число вместо знака операции или был введен *0* вместо *y*:

```
*************** Калькулятор **********
Для выхода введите q в качестве знака операции
Знак (+,-,*,/): +
x=2
y=2
4.00
Знак (+,-,*,/): /
x=9
y=0
Деление на ноль!
Знак (+,-,*,/): 100
Неверный знак операции!
Знак (+,-,*,/): -
x=100
y=54
46.00
Знак (+,-,*,/): q
>>>
```

Глава 7.

ПЕРЕМЕННЫЕ

7.1. Объявление и инициализация переменных

Переменная *– это поименованная область памяти, содержащая какое-то значение.* Значения могут быть разных типов, например, строка, целое число, дробное число (float) и т.д. От типа переменной зависит, как именно она будет храниться в памяти.

Если вы программировали на других языках, то знаете, что переменную нужно сначала объявить, а затем инициализировать. Вот как это происходит на C:

```
int n;
int k = 0;
```

Первая строчка – это объявление переменной, при котором указывается тип переменной – *int*, вторая строчка – объявление и инициализация, то есть присваивание значения переменной.

Отличие Python от таких языков как C/C++/C#, Pascal, Java в том, что в нем не требуется объявление переменной. Python поддерживает динамическую типизацию, это означает, что тип переменной будет определяться автоматически – при присваивании значения. Операция присвоения значения осу-

ществляется с помощью оператора =, как и в других языках программирования:

```
n = 10
print(n)
n = "Марк"
print(n)
```

Обратите внимание: сначала переменной n присваивается целое значение 10, а затем строковое значение "Марк". С одной стороны, очень удобно – в других языках программирования у вас такое не получится. С другой стороны, отсутствие строгой типизации заставляет программиста следить за типом переменной самостоятельно. Представим, что вы сначала присвоили переменной значение n, чтобы использовать ее в качестве счетчика или индекса для обращения к массиву, а затем забыли и присвоили ей строковое значение. Когда вы попытаетесь использовать n в качестве индекса массива, вы получите сообщение об ошибке.

Что касается имен переменных, то есть два основных правила:

1. Имя переменной не должно начинаться с цифры.
2. Имя переменной может состоять только из цифр, букв и знаков подчеркивания.

Имена переменных могут быть даже написаны на русском языке, например:

```
имя = "Денис"
print(имя)
```

Никакой ошибки не будет, но настоятельно рекомендуется имена идентификаторов писать только латинскими буквами.

7.2. Использование переменных

Вы можете объявить переменную в любом месте программы, но до ее первого использования. После того, как переменной присвоено значение, ее можно использовать.

Например:

```
a = 5
b = 7
c = a + b
print(c)
print("Результат ", c)
```

Вы можете использовать переменные в различных операциях, например, в арифметических, если они содержат числовые значения. Переменные можно передавать в функции, например, в функцию print() для вывода значения переменной.

7.3. Рекомендации относительно использования переменных

Python довольно либеральный язык. С одной стороны – это хорошо, с другой стороны – не всегда. Слишком большая свобода ведет к отсутствию порядка. В этом разделе будут приведены некоторые рекомендации, позволяющие навести порядок в вашем коде и сделать его более удобным для чтения и поддержки в будущем.

Итак, рассмотрите следующие рекомендации:

- Хотя переменную можно объявить в любом месте программы, но до первого использования, рекомендуется объявлять переменные в начале программы. Там же можно произвести инициализацию переменных.
- Неплохо бы снабдить переменные комментариями, чтобы в будущем не забыть, для чего используется та или иная переменная.
- Имя переменной должно описывать ее суть. Например, о чем нам говорит переменная s? Это может быть все, что угодно и *сумма* (summ), и *счет* (score) и просто *переменная-счетчик*, когда вы вместо *i* почему-то используете *s*. Когда же вы называете переменную *score*, сразу становится понятно, для чего она будет использоваться.

- Придерживайтесь одной и той же схемы именования переменных. Например, если вы уже назвали переменную *high_score* (нижний регистр и знак подчеркивания), то переменную, содержащую текущий счет пользователя называйте *user_score*, но никак не userScore. С синтаксической точки зрения никакой ошибки здесь нет, но код будет красивее, когда будет использоваться одна схема именования переменных.
- Традиции языка Python рекомендуют начинать имя переменной со строчной буквы и не использовать знак подчеркивания в качестве первого символа в имени переменной. Все остальное – считается дурным тоном.
- Не создавайте слишком длинные имена переменных. В таких именах очень легко допустить опечатку, что не очень хорошо. Да и длинные имена переменных сложно "тянуть" за собой. Если нужно использовать в одной строке несколько переменных, то длинные названия будут выходить за ширину экрана. Максимальная рекомендуемая длина имени переменной – 15 символов.

Глава 8.

ПРАКТИЧЕСКИЙ ПРИМЕР: ПРОГРАММА «ПУТЕВОЙ КОМПЬЮТЕР»

8.1. Общие замечания

Ранее мы создали две простых программы, рассчитывающих время в пути, а также расход топлива. Сейчас мы попробуем на основании этих двух программ разработать программу "Путевой компьютер".

Программа будет учитывать не только время в пути и потребление топлива, но еще и остановки на дозаправки и плановые остановки. Ведь в длительном путешествии нужно делать остановки – без них никак, поэтому мы их должны также учитывать.

Что касается остановок для дозаправки, мы будем считать, что каждая такая остановка занимает 20 минут, а количество остановок будет зависеть от объема бака. Например, если нам понадобится 100 литров, а размер бака у нас 80 литров, то там нужно будет сделать одну остановку для дозаправки.

Наша программа будет демонстрировать все, что было изучено ранее:

- Работа с переменными, в том числе объявление и инициализация
- Арифметические операторы
- Ввод и вывод переменных
- Повторение строк

8.2. Версия 0.0.1

В принципе, смысл программы понятен. Давайте возьмем и напишем ее. Учитывая, что вы дочитали книгу до этого момента, вы сами можете написать такую программу без моих подсказок. Код версии 0.0.1 нашей программы приведен в листинге 8.1.

Листинг 8.1. "Путевой компьютер", версия 0.0.1

```
# Путевой компьютер

program = "Путевой компьютер, v.0.0.1"

# Переменные

stars = 80              # Количество звездочек
tabs = 5                # Количество отступов

dist = 0                # Расстояние, которое нужно проехать
speed = 0               # Средняя скорость авто, км /ч
time = 0                # Время в движении (за рулем)
total_time = 0          # Общее количество времени в пути

tank = 0                # Размер бака
consum = 0              # Средний расход л/100 км
dist = 0                # Расстояние в км.
refuels = 0             # Количество дозаправок
refuel_time = 20        # Время дозаправки
fuel = 0                # Сколько затрачено топлива
price = 0

breaks = 0              # Количество плановых остановок
break_time = 0          # Время каждой плановой остановки

# Выводим заголовок
print("\t" * tabs, program)
print("*" * stars)

# Ввод данных от пользователей
dist = int(input("Введите расстояние, км: "))
speed = int(input("Планируемая средняя скорость (целое число): "))
consum = float(input("Введите средний расход л./100 км (вещ. число): "))
tank = float(input("Объем бака, л: "))
price = float(input("Стоимость 1 литра топлива, р.: "))
breaks = float(input("Количество плановых остановок (без дозаправок): "))
```

```
break_time = float(input("Время каждой плановой остановки, минут: "))

# Производим вычисления
time = dist * 60 / speed                    # Время за рулем
fuel = consum * dist / 100                  # Всего затрачено топлива

refuels = fuel // tank
total_time = time + refuels * refuel_time + breaks * break_time

print("*" * stars)
print("\t" * tabs, "Результаты:")

print("Время за рулем, ч: ", time / 60)
print("Общее время в пути, ч:", total_time / 60)
print("Количество дозаправок: ", refuels)
print("Потрачено времени на дозаправку: ", refuels * refuel_time, " минут")
print("Израсходовано топлива, л.: ", fuel)
print("Стоимость топлива, р: ", fuel * price)
```

Первым делом мы описываем переменные. Можно было бы этого и не делать, но со временем, когда вам захочется усовершенствовать программу, вам сразу станет понятно, для чего используется та или иная переменная. А если вы не станете описывать переменные, вам нужно будет смотреть код и пытаться по нему понять, для чего используются переменные.

Запустим программу и попробуем поработать с ней. Первым делом введем проверочные данные и посмотрим, правильно ли программа работает (рис. 8.1).

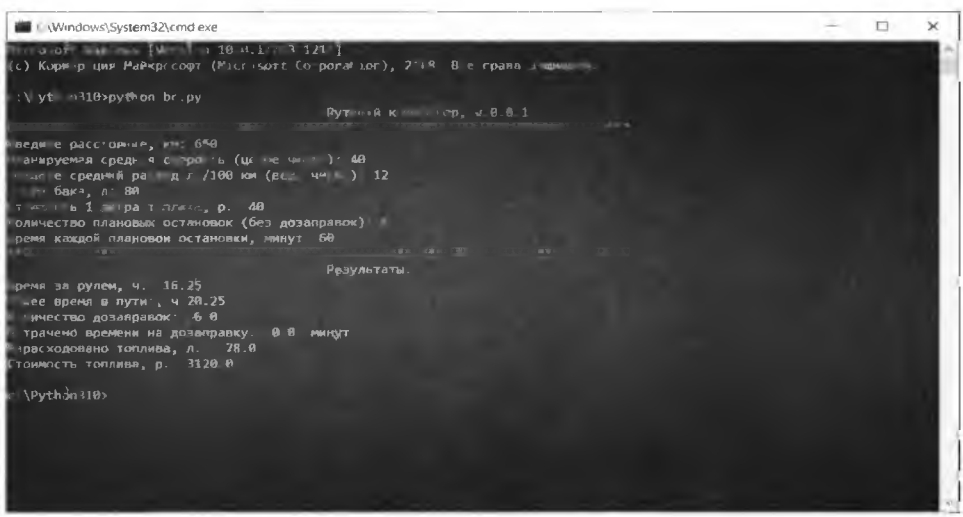


Рис. 8.1. Результат работы программы

Пусть мы хотим проехать 880 км со средней скоростью 90 км/ч, при этом объем бака составляет 80 литров (на момент выезда считаем, что заправлен полный бак), нам нужно сделать 2 плановых остановки по 15 минут каждая, а стоимость одного литра бензина составляет 40 рублей.

Время за рулем считаем так:

```
общая дистанция * 60 / скорость
```

В итоге мы получим время за рулем в минутах, для преобразования в часы нужно разделить его на 60, результат – 9.7 часов за рулем.

К этому времени нам нужно добавить время, потраченное на дозаправки, а также время плановых остановок. Количество дозаправок определяется просто: нужно затраченное топливо (которое вычисляется путем умножения расхода на дистанцию и деления на 100) разделить на объем бака. В нашем случае нужно 105.6 литров топлива, а объем бака – всего 80 литров. Понятно, что нам нужно сделать одну дозаправку.

```
time = dist * 60 / speed                    # Время за рулем
fuel = consum * dist / 100                  # Всего затрачено
топлива

refuels = fuel // tank
# Общее время
total_time = time + refuels * refuel_time + breaks * break_time
```

Обратите внимание: при вычислении количества дозаправок мы используем оператор //, а не /. Дозаправка – это целое число, она либо есть, либо ее нет. Не существует 0.5 или 0.3 дозаправки. Поэтому мы используем оператор //, чтобы получить количество целых частей.

Итак, к нашему времени нам нужно добавить 20 минут на одну дозаправку и 30 минут на плановые остановки: 586 минут + 20 + 30, итого 636 минут или 10.6 часа, что и показала нам наша программа.

Можно сделать вывод, что программа считает правильно, но лично мне не нравится вывод времени:

```
Время за рулем, ч:  9.777777777777777
Общее время в пути:, ч 10.61111111111111
```

Правильнее было бы выводить время в формате 9 часов 46 минут, а не 9.77 часов. Этот недостаток мы исправим в версии 0.0.2 программы.

8.3. Версия 0.0.2

В данной версии нам предстоит исправить небольшую ошибку программы, связанную с выводом затраченного времени.

Чтобы выводить время понятным для человека способом, нам нужно вычислить количество полных часов, а затем от общего количества минут отнять количество минут, содержащихся в полных часах.

Представим, что на поездку было потрачено 586 минут. Полных часов будет 9, а минут в полных часах – 540. Нам нужно отнять 540 от 586, и мы получим 46 минут. Следовательно, на поездку было затрачено 9 часов и 46 минут.

Приступим к реализации. Код будет таким:

```
drive_hours = time // 60
drive_mins = time - drive_hours * 60

total_hours = total_time // 60
total_mins = total_time - total_hours * 60
```

Мы добавили четыре дополнительных переменных, осталось обеспечить их вывод:

```
print("Время за рулем: ", drive_hours, " ч. ", drive_mins, " м.")
print("Общее время в пути: ", total_hours, " ч. ", total_mins, " м.")
```

Все как бы хорошо, но что мы видим в вывод:

```
Время за рулем:  9.0  ч.  46.66666666666663  м.
Общее время в пути:  10.0  ч.  36.66666666666663  м.
```

Все как бы хорошо, но лично мне хотелось видеть целые числа в выводе программы. В этом нам поможет уже знакомая функция приведения к типу int – int(). Полный код программы версии 0.0.2 приведен в листинге 8.2.

Рис. 8.2. "Путевой компьютер", окончательный вариант

Листинг 8.2. "Путевой компьютер", версия 0.0.2

```
# Путевой компьютер

program = "Путевой компьютер, v.0.0.2"

# Переменные

stars = 80                  # Количество звездочек
tabs = 5                    # Количество отступов

dist = 0                    # Расстояние, которое нужно проехать
speed = 0                   # Средняя скорость авто, км /ч
time = 0                    # Время в движении (за рулем)
total_time = 0              # Общее количество времени в пути
drive_hours = 0             # Часов в движении (полных)
drive_mins = 0              # Минут (остаток)
total_hours = 0             # Часов всего (полных)
total_min = 0

tank = 0                    # Размер бака
consum = 0                  # Средний расход л/100 км
dist = 0                    # Расстояние в км.
refuels = 0                 # Количество дозаправок
refuel_time = 20            # Время дозаправки
fuel = 0                    # Сколько затрачено топлива
price = 0
```

```
breaks = 0                  # Количество плановых остановок
break_time = 0              # Время каждой плановой остановки

# Выводим заголовок
print("\t" * tabs, program)
print("*" * stars)

# Ввод данных от пользователей
dist = int(input("Введите расстояние, км: "))
speed = int(input("Планируемая средняя скорость (целое число): "))
consum = float(input("Введите средний расход л./100 км (вещ. число): "))
tank = float(input("Объем бака, л: "))
price = float(input("Стоимость 1 литра топлива, р.: "))
breaks = float(input("Количество плановых остановок (без дозаправок): "))
break_time = float(input("Время каждой плановой остановки, минут: "))

# Производим вычисления
time = dist * 60 / speed                    # Время за рулем
fuel = consum * dist / 100                  # Всего затрачего топлива

refuels = fuel // tank
total_time = time + refuels * refuel_time + breaks * break_time

drive_hours = time // 60
drive_mins = time - drive_hours * 60

total_hours = total_time // 60
total_mins = total_time - total_hours * 60

print("*" * stars)
print("\t" * tabs, "Результаты:")

print("Время за рулем: ", int(drive_hours), " ч. ", int(drive_mins), " м.")
print("Общее время в пути: ", int(total_hours), " ч. ", int(total_mins), " м.")
print("Количество дозаправок: ", refuels)
print("Потрачено времени на дозаправку: ", refuels * refuel_time, " минут")
print("Израсходовано топлива, л.: ", fuel)
print("Стоимость топлива, р: ", fuel * price)
```

Вот теперь программа работает, как нужно и отображает то, что мы хотим видеть:

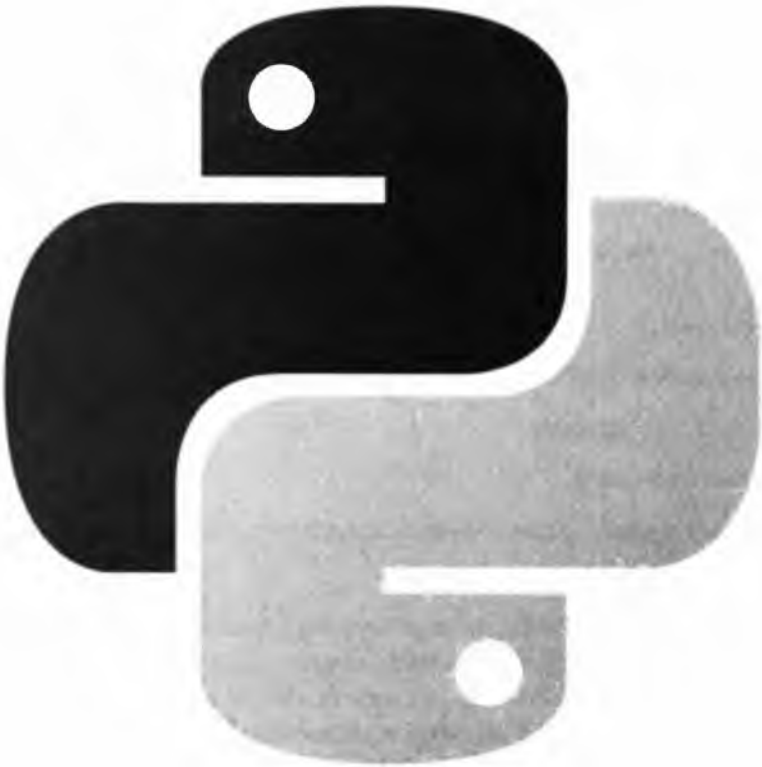
```
                    Путевой компьютер, v.0.0.2
************************************************************

Введите расстояние, км: 880
Планируемая средняя скорость (целое число): 90
```

```
Введите средний расход л./100 км (вещ. число): 12
Объем бака, л: 80
Стоимость 1 литра топлива, р.: 40
Количество плановых остановок (без дозаправок): 2
Время каждой плановой остановки, минут: 15
************************************************************
                        Результаты:

Время за рулем:  9  ч.  46  м.
Общее время в пути:  10  ч.  36  м.
Количество дозаправок:  1.0
Потрачено времени на дозаправку:  20.0  минут
Израсходовано топлива, л.:  105.6
Стоимость топлива, р:  4224.0
```

Далее по мере изучения новых возможностей Python мы будем дорабатывать нашу программу. Хотя и на данный момент – программа считает практически все, что нужно для расчета времени и затрат топлива при путешествии:

- Время за рулем
- Общее время в пути
- Количество дозаправок
- Стоимость топлива
- Учитывает время, потраченное на дозаправку и на остановки

Если поездка очень долгая, то наш бортовой компьютер также справится. Например, нужно остаться на ночлег, ночлег будет занимать 7 часов. Также вам нужно сделать еще 3 остановки по 20 минут каждая (элементарно купить воды, отдохнуть и т.д.). Итого укажите 4 остановки и время каждой – 120 минут (2 часа, а всего у нас уйдет на отдых 8 часов – ночлег и 3 остановки по 20 минут).



ЧАСТЬ III.

ОСНОВНЫЕ ОПЕРАТОРЫ

Третья часть книги посвящена основным операторам языка Python. Мы рассмотрим условный оператор, операторы циклов, научимся генерировать случайные числа и многое другое.

Глава 9.

УСЛОВНЫЙ ОПЕРАТОР *IF*

9.1. Сколько тебе лет

Ни одна серьезная программа не обходится без условного оператора *if*, позволяющего выбрать, какое действие нужно выполнить в зависимости от определенного условия. Давайте сразу рассмотрим пример (лист. 9.1).

Листинг 9.1. Сколько тебе лет

```
print("*" * 5, "Добро пожаловать!", "*" * 5)
age = int(input("Сколько тебе лет? "))
if age < 18:
  print("Извини, тебе нельзя использовать эту программу!")
```

Разберемся, как работает эта программа. Сначала пользователь вводит свой возраст. Далее мы сравниваем возраст (age) с 18. Если возраст меньше 18, то мы выводим сообщение о том, что использовать эту программу нельзя.

Рис. 9.1. Результат работы программы с условным оператором

Основная часть программы следующая:

```
if age < 18:
  print("Извини, тебе нельзя использовать эту программу!")
```

Здесь все предельно просто. Если условие истинно, то есть значение переменной *age* меньше 18, то выводится строка. Если нет, то управление переходит к следующему оператору.

9.2. Условия и операторы сравнения

При формировании конструкции *if* всегда задается условие, то есть выражение, которое может быть истинным или ложным. Чаще всего при задании условий используются операторы сравнения. В нашем случае мы использовали оператор < (см. табл. 9.1).

Таблица 9.1. Операторы сравнения

Оператор	Значение
<	Меньше
>	Больше

<=	Меньше или равно
>=	Больше или равно
==	Равно
!=	Не равно

Внимание! Условные операторы в Python могут сравнивать не только числа, но и строки, например audi < bmw, поскольку *audi* находится по алфавиту раньше, чем *bmw*. Но не все в Python можно сравнить. Объекты разных типов, для которых не определено отношение порядка, нельзя сравнить с помощью операторов <, <=, >, >=. Например, вы не можете сравнить число и строку. Если вы попытаетесь это сделать, получите огромное сообщение об ошибке.

9.3. Блоки кода и отступы

Давайте еще раз посмотрим на наш условный оператор:

```
if age < 18:
  print("Извини, тебе нельзя использовать эту программу!")
```

Обратите внимание, что вторая строка написана с отступом. Отступ превращает наш код в блок. Блок – это одна или несколько идущих подряд строк с одинаковым отступом. Блок – единая конструкция.

Блоки используются, когда в случае выполнения условия нужно выполнить несколько операторов:

```
if age < 18:
  print("Извини, тебе нельзя использовать эту программу!")
  print("Когда тебе исполнится 18, будем рады тебя видеть!")
```

На другом языке программирования блоки кода, как правило, заключают в фигурные скобки:

```
if ($age < 18) {
  echo "Извини, тебе нельзя использовать эту программу!";
  echo "Когда тебе исполнится 18, будем рады тебя видеть!";
}
```

В других языках программирования в скобках какие-либо отступы соблюдать не нужно, операторы разделяются точкой с запятой, а написать вы можете их хоть в одну строчку, лишь бы они были в одних фигурных скобках.

В Python программисту нужно следить за отступами. Но с другой стороны это приучает его к порядку и делает код удобным для чтения.

9.4. Оператор *if* с условием *else*

Наша программа выводит сообщение пользователю, если он младше 18 лет, но если пользователю уже есть 18 или он старше, программа ничего не сообщает. Исправим это:

```
print("*" * 5, "Добро пожаловать!", "*" * 5)
age = int(input("Сколько тебе лет? "))
if age < 18:
  print("Извини, тебе нельзя использовать эту программу!")
  print("Когда тебе исполнится 18, будем рады тебя видеть!")
else:
  print("Доступ открыт!")
```

В результате вывод программы в зависимости от ввода пользователя будет таким:

```
***** Добро пожаловать! *****
Сколько тебе лет? 11
Иэвини, тебе нельзя использовать эту программу!
Когда тебе исполнится 18, будем рады тебя видеть!
>>>

***** Добро пожаловать! *****
Сколько тебе лет? 19
Доступ открыт!
>>>
```

Общая форма оператора *if* выглядит так:

```
if условие:
  блок_который_будет_выполнен_если_условие_истинно
else:
  блок_который_будет_выполнен_если_условие_ложно
```

Напомню, что под блоком кода подразумевается одна или несколько строк кода с одинаковым отступом.

9.5. Несколько условий

А что делать, если нам нужно проверить несколько условий сразу? В этом случае поможет конструкция *elif*, позволяющая задать дополнительное условие. Рассмотрим пример кода:

```
mark = int(input("Какую оценку ты получил по математике? "))

if mark == 5:
    print("Молодец! Так держать!")
elif mark == 4:
    print("Хорошо, но могло бы быть и лучше!")
elif mark == 3:
    print("Нужно лучше стараться!")
elif mark <=2:
    print("Тебе необходим репетитор!")
```

После *elif* следует обычное условие, как будто бы перед нами обычный оператор *if*.

Общая форма оператора *if..elif..else* следующая

```
if <условие1>:
  <блок1>
elif <условие 2>
  <блок 2>
…
elif <условие N>:
```

```
  <блок N>
else:
  <блок N+1>
```

Рис. 9.2. Результат работы программы

Обратите внимание, что вы можете использовать конструкцию *else*, несмотря на наличие *elif*. Главное, чтобы *else* шла последней. Блок <N+1> будет выполнен, только если не сработали другие условия и не были выполнены другие блоки кода. После того, как выполнен один из блоков кода, например, при срабатывании условия K, то управление будет передано оператору, следующему за *if..elif..else*:

```
mark = int(input("Какую оценку ты получил по математике? "))

if mark == 5:
    print("Молодец! Так держать!")
elif mark == 4:
    print("Хорошо, но могло бы быть и лучше!")
elif mark == 3:
    print("Нужно лучше стараться!")
elif mark <=2:
    print("Тебе необходим репетитор!")

print("Удачи!")
```

Вывод программы будет такой:

```
Какую оценку ты получил по математике? 5
Молодец! Так держать!
Удачи!
```

Примечание. Если вы программировали на других языках, то вам наверняка знаком оператор *switch..case*. Смысл этого оператора в следующем: в *switch* задается выражение, значение которого сравнивается со значениями, заданными в блоках *case*. Если значение совпало, то выполняются операторы, указанные в этом блоке *case*. К сожалению, в Python нет такого оператора, и вам придется строить конструкции *if..elif..else*. Некоторые программисты предлагают использовать словари вместо *switch.. case*, но данный подход не универсальный и подойдет далеко не всегда.

9.6. Программа "Тесты"

Давайте напишем небольшую программу проверки компьютерных знаний, а именно тесты. Мы пока не знакомы ни с массивами, ни какими-либо другими структурами хранения данных, поэтому мы будем просто задавать вопросы и обрабатывать ответ пользователя, а потом выведем результат.

Листинг 9.2. Программа "Тесты"

```
mark = 0      # Оценка

print("""
      Сколько бит в одном байте
      1) 8
      2) 6
      3) 4
      4) 2
      """)
a = int(input("Ваш ответ: "))

if a == 1:
    mark = mark + 1

print("""
      Сколько байт в одном килобайте
      1) 1000
```

```
        2) 1024
        3) 1048
        4) 256
        """)
a = int(input("Ваш ответ: "))

if a == 2:
    mark = mark + 1

print("""
        Компания-разработчик Windows
        1) Mikrosoft
        2) Melkosoft
        3) Cybersoft
        4) Microsoft
        """)
a = int(input("Ваш ответ: "))

if a == 4:
    mark = mark + 1

print("""
        Самая "яблочная" операционная система
        1) AppleOS
        2) Linux
        3) macOS
        4) FreeBSD
        """)
a = int(input("Ваш ответ: "))

if a == 3:
    mark = mark + 1

print("""
        Символом какой операционной системы является пингвин
        1) Linux
        2) FreeBSD
        3) OS/2
        4) Windows
        """)
a = int(input("Ваш ответ: "))

if a == 1:
    mark = mark + 1

print("Ваша оценка: ", mark)
```

Внимание! Мы не производим никакой обработки ввода пользователя. Если пользователь вместо номера ответа введет строку, например, "Windows" вместо 4 – в последнем вопросе, то он получит сообщение об ошибке, а выполнение программы будет прервано. Чтобы такого не произошло, желательно настроить обработку исключительной ситуации. Обработка исключительных ситуаций – это очень важно в реальных программах. Ее также называют "защитой от дурака". Взять бы даже нашу программу. Ни одной умный человек не будет вводить строку, если нужно ввести числовой вариант ответа. Но есть люди не очень умные, а есть такие, кто специально обучен искать всякие баги в программах – тестеры. И те, и другие обязательно найдут данный просчет в вашей программе. Вы же не хотите, чтобы это произошло? Сейчас у нас программы простые и, по сути, демонстрационные. Но скоро вы начнете писать реальные программы, и там защита от дурака будет весьма кстати.

Программа поочередно задает вопросы, а затем, в случае правильного ответа просто прибавляет к переменной *mark* единицу. Всего 5 вопросов, поэтому максимально возможная оценка – 5.

Рис. 9.3. Программа "Тесты"

Программа "Тесты" в нынешнем виде очень простая и плохо подходит для тестирования:

- Если открыть код программы, можно увидеть невооруженным взглядом правильные ответы. Они никак не скрыты. Учитывая, что Python – это интерпретатор, а не компилятор, исходный код программы открыт всем желающим.
- Если нужно изменить текст вопроса, то придется редактировать код программы, а это не всегда хорошо.
- Если изменить количество вопросов, то также придется редактировать код программы и даже изменять ее логику. Сейчас логика такова, что задается 5 вопросов, и оценка равна количеству правильных ответов. А как оценить человека, если вопросов 10, а ответил правильно на 7 из них?

В дальнейшем будет рассмотрена работа с файлами, также будет показано, как можно "законсервировать" в файл любой объект в Python. В результате консервации он будет помещен в файл в двоичном виде и не будет доступен всем желающим. Далее в этой книге мы перепишем программу "Тесты" так, что в двоичном файле она будет хранить два массива – первый будет содержать вопросы, а второй – правильные ответы. Поскольку данные в файле будут храниться в закодированном виде, то тестируемый не сможет открыть файл и просмотреть правильные варианты ответа. Он разве что сможет открыть саму программу, но в ней не будет правильных ответов. Чтобы вас заинтересовать, покажу рисунок, на котором изображен файл с вопросами и ответами. Как видите, даже при всем своем желании у пользователя не получится прочитать ни список вопросов, ни варианты ответов.

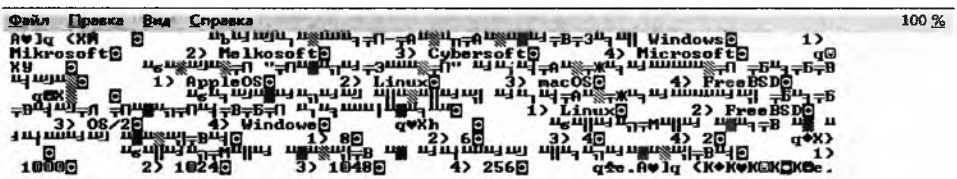


Рис. 9.4. Вот так выглядит файл с вопросами и ответами

Также у нас будет возможность задавать произвольное число вопросов (соответственно и ответов), что сделает программу более универсальной. Подробнее обо всем этом мы поговорим в дальнейшем. Не переключайтесь!

Глава 10.

ЦИКЛЫ

Если проанализировать все программы, то на втором месте после условного оператора будут операторы цикла. Используя цикл, вы можете повторить операторы, находящиеся в теле цикла. Количество повторов зависит от типа цикла – можно даже создать бесконечный цикл. В этом и есть некоторая опасность циклов – если не предусмотреть условие выхода из цикла, то может произойти зацикливание программы, когда тело цикла будет выполняться постоянно.

10.1. Цикл *while*. Программа "Угадай лучший автомобиль"

Чтобы продемонстрировать работу цикла *while* мы напишем простейшую программу, предлагающую пользователю угадать лучший автомобиль (листинг 10.1).

Листинг 10.1. Угадай лучший автомобиль

```
# Угадай лучший автомобиль
```

```
print("Угадай лучший автомобиль")
print("Название автомобиля нужно вводить строчными буквами")

response = ""
while response != "toyota":
  response = input("Какой авто самый лучший? ")

print("Ура! Вы это сделали!")
```

Основная часть программы:

```
while response != "toyota":
  response = input("Какой авто самый лучший? ")
```

Итак, у нас есть переменная *response*. Цикл *while* (пока) будет выполняться до тех пор, пока значение мереной *response* не будет равно "toyota". Обратите внимание: мы предусмотрели условие выхода из цикла:

```
response != "toyota":
```

Как и в случае с оператором *if*, тело цикла может состоять из нескольких операторов, заключенных в один блок. Блоком считаются строки, следующие друг за другом, с одинаковым отступом.

Переменная, которую мы проверяем в условии выхода из цикла, называется *управляющей*. Как правило, до начала цикла нужно ее инициализировать. Мы установили в качестве ее значения пустую строку:

```
response = ""
```

Вывод программы будет таким:

```
Угадай лучший автомобиль
Название автомобиля нужно вводить строчными буквами
Какой авто самый лучший? bmw
Какой авто самый лучший? audi
Какой авто самый лучший? nissan
Какой авто самый лучший? toyota
Ура! Вы это сделали!
```

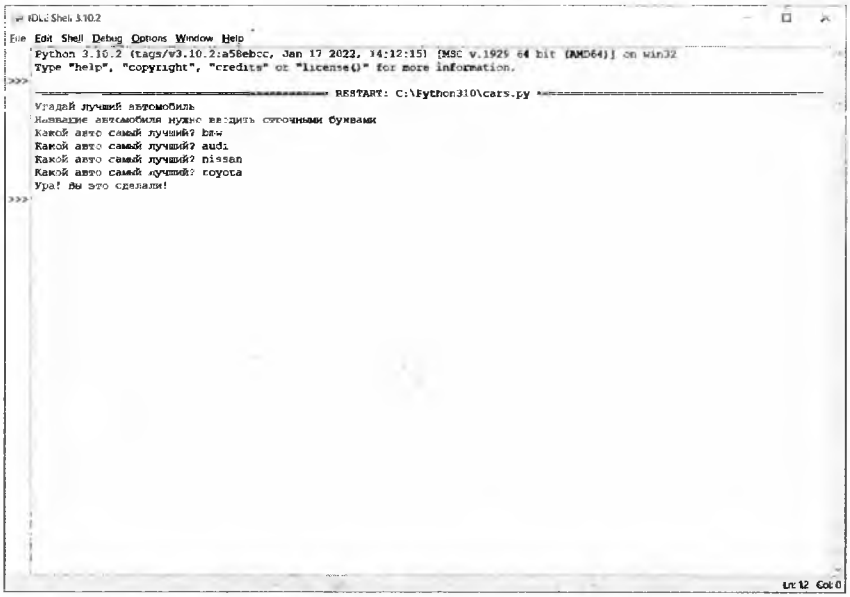


Рис. 10.1. Вывод программы

10.2. Бесконечные циклы

10.2.1. Бесконечный цикл по ошибке

Особое внимание уделите изменению значения управляющей переменной. Неправильно составленное условие может привести к бесконечному циклу. Рассмотрим пример зацикливания программы:

```
k = 10
while k > 5:
    print(k)
    k = k + 1
```

Здесь цикл будет выполняться, пока *k* больше 5. Изначально *k* у нас больше 5, далее значение *k* только увеличивается, поэтому мы получим бесконечный цикл – программа будет бесконечно увеличивать значение *k* и выводить его:

```
388
389
```

```
390
391
392
393
394
395
396
397
398
399
400
401
402
403
```

Прервать выполнение зацикленной программы можно с помощью комбинации клавиш Ctrl + C:

```
Traceback (most recent call last):
  File "C:/Python310/20.py", line 3, in <module>
    print(k)
  File "C:\Python310\lib\idlelib\PyShell.py", line 1344, in write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
>>>
```

Как исправить бесконечный цикл. Здесь нужно или редактировать условие или же процесс изменения значения управляющей переменной. Например, можно сделать декремент управляющей переменной:

```
k = k - 1
```

Тогда программа выведет 5 чисел и завершит свою работу:

```
10
9
8
7
6
```

Если разложить наш цикл на итерации, то получится табличка, приведенная ниже (табл. 10.1)

Таблица 10.1. Итерации цикла

Номер итерации	Значение *k*	Проверка условия	Действия
1	10	true	`print(k) # 10` `k = k - 1 # 9`
2	9	true	`print(k) # 9` `k = k - 1 # 8`
3	8	true	`print(k) # 8` `k = k - 1 # 7`
4	7	true	`print(k) # 7` `k = k - 1 # 6`
5	6	true	`print(k) # 6` `k = k - 1 # 5`
6	5	false	-

Можно было бы изменить и условие, например:

```
k = 10
while k < 15:
    print(k)
    k = k + 1
```

Тогда программа выведет:

```
10
11
12
13
14
```

Типичная ошибка новичков – многие вообще забывают изменять переменную в цикле. Например:

```
# Внимание! Код содержит ошибку!
k = 1
while k <= 10:
  print(k)
```

Очевидно, программист хотел, чтобы программа отобразила числа от 1 до 10, но забыл изменить значение *k* в теле цикла. Следовательно, программа будет выполняться бесконечно.

Подытожим. Чтобы не получить бесконечный цикл, необходимо:

1. Следить за начальным значением управляющей переменной
2. Проанализировать условие выхода из цикла
3. Следить за процессом изменения значения управляющей переменной: ей должны присваиваться такие значения, которые рано или поздно приведут к выходу из цикла

10.2.2. Намеренный бесконечный цикл. Операторы *break* и *continue*

Ради справедливости нужно отметить, что иногда бесконечные циклы создаются преднамеренно, поскольку того требуют условия задачи. Например, мы пишем какую-то программу, которая обрабатывает запросы извне, например, поступающие по сетевому сокету.

В этом случае проще написать так:

```
while True:
  блок_кода
```

Не нужно использовать пример, приведенный ранее (когда значение управляющей переменной не установлено) – так ваш код будет похож на ошибочный. А когда вы указываете while True:, то вы явно сообщаете, что хотите создать бесконечный цикл.

Как все-таки прервать цикл, например, если в теле цикла было получено сообщение прекратить работу программы? Для этого нужно использовать инструкцию *break*. Например:

```
while True:
  data = read_from_socket()
  if data == "quit":
    break
```

Вы обязательно должны предусмотреть возможность выхода из бесконечного цикла. В данном случае, если значение переменной *data* будет равно "quit", то выполнение цикла будет прервано.

Нужно обязательно предусмотреть возможность выхода из цикла, поскольку прерывание цикла по нажатию Ctrl + C, во-первых, считается дурным тоном, во-вторых, приводит к прерыванию всей программы, а не только цикла.

Как и в других языках программирования, в Python есть инструкция *continue*, позволяющая пропустить итерацию. Например:

```
k = 0
while k < 17:
    k = k + 1
    if k %5 == 0:
        continue
    print(k)
```

Программа выведет:

```
1
2
3
4
6
7
8
9
11
12
13
14
16
17
```

Как видите, в списке отсутствуют значения, кратные 5. Если остаток от деления на 5 равен 0, то мы просто переходим на следующую итерацию и пропускаем текущую. В этом коде, не смотря на его простоту, очень сложно допустить ошибку. Например, если изменять значение *k* уже после проверки на кратность оператором *if*, то можно получить бесконечный цикл:

```
# Внимание! Код содержит ошибку!
k = 0
while k < 17:
    if k %5 == 0:
        continue
    print(k)
    k = k + 1
```

Давайте посмотрим, что произойдет. Представим, что *k* уже равно 4. Поскольку $k < 17$, начнется выполнения тела цикла. Так как $k \% 5$ не равно 0, инструкция *continue* не будет выполнена. Цифра 4 будет выведена на экран, после чего значение *k* будет увеличено на 1 и станет равно 5.

Далее проверяется условие: значение $k < 17$, поэтому начинается выполнение тела цикла. В результате $k \% 5$ мы получаем 0 и пропускаем текущую итерацию. Но проблема в том, что значение *k* мы так и не увеличили и оно по-прежнему равно 5. Ситуация повторяется и так будет происходить, пока вы не нажмете Ctrl + C.

10.3. Истинные и ложные значения

Рассмотрим вот такое условие:

```
if score:
```

Странно, ведь *score* не сравнивается ни с одним значением, как так? Сама переменная *score* выступает как условие. Если значение *score* будет равно 0, то это считается *ложным значением* (*false*). Любое другое значение считается *истинным* (*true*). С тем же успехом мы могли бы написать:

```
if score > 0:
```

Но незачем делать код сложнее! Ведь можно сделать его проще!

10.4. Логические операторы *not*, *or*, *and*. Программа "Уровень доступа"

Логические операторы *not*, *or* и *and* представляют логические операции НЕ, ИЛИ и И соответственно.

Логическая бинарная операция И (and) возвращает *true*, если оба операнда истинны:

```
if money and score:
```

Здесь подразумевается, если деньги и счет отличны от 0, то условие будет истинным.

Логическая бинарная операция ИЛИ (or) возвращает *true*, если один из операндов равен *true*:

```
if money or score:
```

Если одна из переменных, содержит значение, отличное от 0, то условие будет истинным.

Логическая унарная операция отрицания NOT возвращает истину, если операнд был ложным и наоборот. Вот как можно бесконечно запрашивать ввод пароля, пока он не будет введен:

```
password = ""
while not password:
  password = input("Password: ")
```

Данные логические операции можно использовать для составления более сложных условий в циклах и условных операторах *if*. Рассмотрим небольшой пример (листинг 10.2). Данная программа запрашивает логин и пароль и на основании этих данных определяет уровень доступа.

Листинг 10.2. Определение уровня доступа

```
level = 0       # Уровень доступа

login = ""
while not login:
  login = input("Login: ")

password = ""
while not password:
  password = input("Password: ")

if login == "root" and password == "tabk":
    level = 10
elif login == "den" and password == "secret":
    level = 5

if level:
    print("Привет, ", login)
    print("Твой уровень доступа: ", level)
else:
    print("Доступ закрыт!")
```

По умолчанию уровень доступа равен 0. Если это так, то доступ закрыт. Если уровень отличается от 0, то программа выводит приветствие и сообщает уровень доступа.

Сначала мы в двух циклах *while* запрашиваем логин и пароль. Благодаря наличию *not* мы будем запрашивать логин и пароль до тех пор, пока они не будут введены.

Далее мы сравниванием введенные значения с определенными константами и определяем уровень доступа.

Рассмотрим вывод программы:

```
==========================
Login: den
Password: secret
Привет,  den
Твой уровень доступа:  5
>>>
==========================
Login: root
Password: 1234
Доступ закрыт!
>>>
```

В первом случае были введены правильные логин и пароль. Программа сообщила уровень доступа. Во втором случае логин был введен правильно, а пароль – нет. Программа сообщила, что доступ закрыт.

10.5. Цикл со счетчиком

Как и в других языках, в Python есть цикл со счетчиком. Это цикл *for*. Преимущество этого цикла в том, что он практически не может привести к зацикливанию (если не считать особо сложных случаев с нарушенной логикой программы), поскольку рано или поздно счетчик закончится.

Внешний вид самого цикла немного озадачит, если вы программировали на других языках. Он больше похож на цикл *foreach*, чем на классический цикл *for*.

Но обо всем по порядку. Начнем с самого простого примера – вывод чисел определенного диапазона:

```
for i in range(10):
    print(i, end=" ")

print()

for i in range(0, 50, 5):
    print(i, end=" ")

print()

for i in range(10, 0, -1):
    print(i, end=" ")
```

Вывод будет таким:

```
0 1 2 3 4 5 6 7 8 9
0 5 10 15 20 25 30 35 40 45
10 9 8 7 6 5 4 3 2 1
```

Первый цикл *for* выводит значения от 0 до 10. Мы указываем только верхнюю границу. Если нижняя граница не указана, то подразумевается, что это 0.

Второй цикл *for* работает от 0 до 5, а увеличение счетчика происходит сразу на 5 единиц. Поэтому мы увидим числа 0, 5, 10, 15 и т.д. – кратные 5.

Третий цикл работает от 10 до 0, уменьшение счетчика происходит на единицу (-1). Поэтому числа будут выведены в обратном порядке.

Функция range() возвращает последовательность цифр. Если ей передать в качестве аргумента положительное число, то последовательность будет охватывать числа от 0 до переданного аргумента (включая его).

Если передать функции range() три аргумента, как мы это сделали во втором и третьем случаях, то они будут рассматриваться как начало, конец счета и интервал. Начало – это первый элемент нашей последовательности чисел, а конечное значение в него не попадает, поэтому мы получили набор чисел `0 5 10 15 20 25 30 35 40 45` во втором случае.

Цикл *for* можно использовать не только для прохода по последовательности чисел. Возможен проход по любой последовательности элементов. Например, вот так можно пройтись по всем буквам:

```
for letter in word:
  print(letter)
```

Глава 11.

ГЕНЕРАТОР СЛУЧАЙНЫХ ЧИСЕЛ. ИГРА "УГАДАЙ ЧИСЛО"

11.1. Постановка задачи

Сейчас мы напишем игровую программу "Угадай число", которая будет демонстрировать следующее:

- Работу с генератором случайных чисел
- Использование цикла *while*
- Прерывание итерации

Работа с циклами была рассмотрена в предыдущей главе, а сейчас, так сказать, мы теорию закрепим практикой.

Алгоритм работы будет такой:

- В цикле мы "загадываем" случайное число от 1 до 10
- Затем просим пользователя отгадать это число
- Если число правильное, мы выводим соответствующее сообщение и увеличиваем значение переменной *score*

Также будет показано, как исправить логическую ошибку в программе.

11.2. Работа с генератором случайных чисел

Для подключения генератора случайных чисел нужно импортировать модуль *random*:

```
import random
```

Далее нужно вызвать функцию randint(), передав ей начальное и конечное значение. Возвращенное случайное число будет лежать в диапазоне между ними:

```
random.randint(1, 10)
```

В модуле *random* также есть функция randrange(), возвращающая случайное целое число в промежутке от 0 до преданного в качестве параметра значения (но, не включая само значение), то есть вызов randrange(10) вернет числа от 0 до 9 включительно.

Как по мне, то проще использовать randint(), чем randrange(). Но это смотря, что вам нужно.

11.3. Код программы

Код программы, действительно, очень прост (листинг 11.1).

Листинг 11.1. Код программы "Угадай число"

```
import random

print("*" * 10, "Угадай число", "*" * 10)

print("Компьютер будет загадывать число от 1 до 10. Попробуй
угадать число. Для выхода введи 0")
answer = 1;
score = 0;
i = 0

while answer:
    i = i + 1
    rand = random.randint(1, 10)
```

```
    answer = int(input("Число загадано. Попробуй отгадать: "))
    if answer == rand:
        score = score +1
        print("Правильно! Всего ты отгадал чисел ", score, "
из ", i)
    else:
        print("Неправильно :(")

print("До встречи в следующий раз")
```

Программа ничего сверхъестественного не делает. В цикле *while* она проверяет введенное пользователем значение. Если оно совпадает со сгенерированным в начале итерации случайным значением, значит, выводится соответствующее сообщение и увеличивается значение переменной *score*. Параллельно мы ведем счетчик итераций, чтобы знать, сколько попыток совершил пользователь (переменная *i*).

Посмотрим на вывод программы:

```
********** Угадай число **********
Компьютер будет загадывать число от 1 до 10. Попробуй угадать число.
Для выхода введи 0
Число загадано. Попробуй отгадать: 7
Неправильно :(
Число загадано. Попробуй отгадать: 3
Неправильно :(
Число загадано. Попробуй отгадать: 4
Неправильно :(
Число загадано. Попробуй отгадать: 1
Неправильно :(
Число загадано. Попробуй отгадать: 9
Правильно! Всего ты отгадал чисел  1  из  5
Число загадано. Попробуй отгадать: 7
Неправильно :(
Число загадано. Попробуй отгадать: 6
Неправильно :(
Число загадано. Попробуй отгадать: 9
Неправильно :(
Число загадано. Попробуй отгадать: 8
Неправильно :(
Число загадано. Попробуй отгадать: 2
Неправильно :(
Число загадано. Попробуй отгадать: 3
Неправильно :(
Число загадано. Попробуй отгадать: 5
```

```
Неправильно :(
Число загадано. Попробуй отгадать: 6
Неправильно :(
Число загадано. Попробуй отгадать: 9
Правильно! Всего ты отгадал чисел  2  из  14
Число загадано. Попробуй отгадать: 0
Неправильно :(
До встречи в следующий раз
```

11.4. Исправление логической ошибки в программе

Все бы хорошо, но в нашей программе есть одна логическая ошибка и как минимум одна недоработка. Для выхода пользователь должен ввести 0. Но посмотрите, что происходит при этом.

Программа считает 0 … еще одним вариантом, но никак не признаком выхода, поэтому она сообщает, что введенный вариант неправильный. Но он и не может быть правильным, поскольку случайные числа генерируются в диапазоне от 1 до 100.

Исправить эту ошибку можно, если добавим конструкцию:

```
if answer == 0:
  break
```

Данную инструкцию нужно добавить в самое начало тела цикла. Если пользователь введет 0, выполнение будет прервано. Также было бы неплохо, чтобы программа выводила статистику по окончанию игры:

```
print("Всего ты отгадал чисел ", score, " из ", i)
```

Измененный код приведен в листинге 11.2.

Листинг 11.2. Окончательный вариант

```
import random

print("*" * 10, "Угадай число", "*" * 10)
```

```
print("Компьютер будет загадывать число от 1 до 10. Попробуй угадать
число. Для выхода введи 0")

answer = 1;
score = 0;
i = 0

while answer:
    if answer == 0:
      break

    rand = random.randint(1, 10)
    answer = int(input("Число загадано. Попробуй отгадать: "))
    if answer == rand:
        score = score +1
        print("Правильно! ")
    else:
        print("Неправильно :(")
    i = i + 1

print("Всего ты отгадал чисел ", score, " из ", i)
print("До встречи в следующий раз")
```

Вывод программы будет следующим:

```
********** Угадай число **********
Компьютер будет загадывать число от 1 до 10. Попробуй угадать
число. Для выхода введи 0
Число загадано. Попробуй отгадать: 9
Неправильно :(
Число загадано. Попробуй отгадать: 8
Правильно!
Число загадано. Попробуй отгадать: 0
Неправильно :(
Всего ты отгадал чисел  1  из  3
До встречи в следующий раз
```

Вот теперь все правильно и работает как нужно!

Примечание. При чтении данных мы не производим проверку их корректности. Если пользователь введет строку вместо числа, то выполнение программы будет установлено, а на консоли будет изображено сообщение об ошибке. Для обработки таких ситуаций используются блоки try..except, о которых мы поговорим в дальнейшем.

ЧАСТЬ IV.

ПРАКТИЧЕСКАЯ РАБОТА СО СТРОКАМИ

Ранее мы рассматривали только основы строк. Строки – это настолько важная тема, что пришло время посвятить всю четвертую часть книги детальной работе со строками.

Глава 12.

ОПЕРАТОРЫ И ФУНКЦИИ ДЛЯ РАБОТЫ С ПОСЛЕДОВАТЕЛЬНОСТЯМИ И СТРОКАМИ

12.1. Функция len(). Длина текста

Строки являются одним из типов последовательностей. Элементами строки являются отдельные символы. Для работы с последовательностями любого рода, в том числе и строками в Python есть несколько полезных функций и операторов, которые позволяют узнать длину последовательности, узнать, есть ли в ней определенный элемент. В этой главе мы познакомимся с этими средствами языка Python.

Начнем мы с функции len(), которая возвращает длину текста. Напишем небольшую программу, которая демонстрирует ее работу (листинг 12.1).

Листинг 12.1. Длина текста

```
ltext = 0        # Длина текста

print("Демонстрация функции len()")

text = input("Введите произвольный текст: ")

print("Длина введенного текста: ", len(text))
```

Сначала мы читаем введенный пользователем текст в переменную *text*, а затем выводим результат функции len(text) на экран (рис. 12.1)

Рис. 12.1. Длина текста

12.2. Оператор *in*

При работе с последовательностями довольно полезным является оператор *in*, позволяющий проверить, есть ли тот или иной элемент в последовательности или нет. Например:

```
if "a" in message:
  print("Искомый символ есть в сообщении")
else
  print("Искомого символа нет в сообщении")
```

12.3. Анализатор слова

Сейчас мы напишем программу, которая:

- Сообщает длину слова
- Сообщает количество гласных и согласных (для русского текста) букв
- Демонстрирует разрыв в строке кода (когда строка кода является слишком длинной)

- Демонстрирует использование метода lower() для приведения символа в нижний регистр
- Демонстрирует использование цикла *for* для прохода по строке (посимвольно)
- Демонстрирует использование оператора *in* в цикле *for*

Листинг 12.2. Анализатор слова

```
print("*" * 10, " Анализатор слова ", "*" * 10)
word = input("Введите слово: ")

vowels = 0
consonants = 0

for i in word:
    letter = i.lower()
    #print(letter)
    if letter == "а" or letter == "о" or\
       letter == "и" or letter == "е" or\
       letter == "ё" or letter == "э" or\
       letter == "ы" or letter == "у" or\
       letter == "ю" or letter == "я":
        vowels += 1
    else:
        consonants += 1
print("Длина текста:", len(text))
print("Гласные %i Согласные %i" % (vowels, consonants))
```

Вывод программы будет таким:

```
**********  Анализатор слова  **********
Введите слово: привет
Длина текста: 6
Гласные 2 Согласные 4
```

Переменная *vowels* будет содержать количество гласных букв, а *consonants* – согласных букв. В цикле *for* мы проходимся по введенному слову (переменная *word*).

В теле цикла мы формируем переменную *letter* – это *i*, приведенная к нижнему регистру:

```
letter = i.lower()
```

Далее мы проверяем, относится ли буква к гласным или нет. Всего в русском языке 10 гласных букв – а, о, и, е, ё, ы, у, э, ю, я. Если будет обнаружена гласная буква, то переменная *vowels* будет увеличена на 1, в противном случае будет увеличена переменная *consonants*:

```
    if letter == "а" or letter == "о" or\
       letter == "и" or letter == "е" or\
       letter == "ё" or letter == "э" or\
       letter == "ы" or letter == "у" or\
       letter == "ю" or letter == "я":
        vowels += 1
    else:
        consonants += 1
```

Обратите внимание на **два момента**. **Первый** – как происходит разрыв строки кода. Если строка кода слишком длинная, что не помещается на экране, можно использовать символ \ для ее разрыва.

Второй – как мы увеличиваем счетчики гласных и согласных. Записи вида:

```
vowels += 1
consonants += 1
```

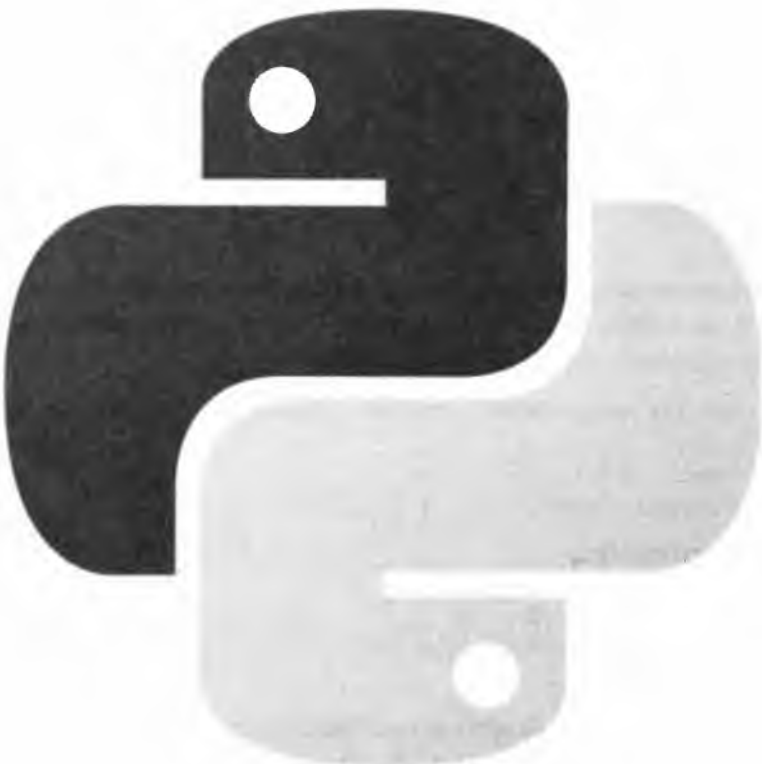
аналогичны операторам:

```
vowels = vowels + 1
consonants = consonants + 1
```

Кроме операторов += вы можете использовать и другие операторы:

- -=
- *=
- /=

Все эти операторы применяются аналогично оператору +=.



Глава 13.

ИНДЕКСАЦИЯ СТРОК

13.1. Введение в индексацию строк

Как было рассказано ранее, с помощью цикла *for* можно перебирать строку посимвольно, что позволяет обращаться к каждому отдельному символу строки. Это называется *последовательным доступом* к элементам.

Последовательный доступ можно сравнить с извлечением книг из ящика – сначала вы вытаскиваете верхние книги, затем книги, которые лежат в ящике ниже и т.д. Постепенно вы подбираетесь к последней книге. Но вы не можете достать из ящика книгу, которая лежит в четвертом ряду (сверху) и справа.

Для этого нужен *произвольный* или *прямой доступ*. В случае с ящиком его организовать сложно, но, к счастью, строки в Python – это не ящики. Для обращения к произвольному символу строки можно указать его порядковый номер в последовательности (индекс). Прямой доступ можно сравнить с книжкой полкой – вы можете взять и достать нужную книгу с полки без необходимости извлечения других книг.

13.2. Демонстрация прямого доступа к строке

Обратиться к произвольному элементу строки можно так:

```
строка[позиция]
```

При использовании прямого доступа к строке помните, что нумерация строк начинается с нуля, например:

```
word = "привет"
print(word[1])
```

Программа выведет символ "р", то есть второй по счету.

В качестве примера прямого доступа к элементам строки напишем программу доступа к случайному символу строки (лист. 13.1)

Листинг 13.1. Демонстрация произвольного доступа к строке

```
import random

word = input("Введите слово: ")

max = len(word)
min = -len(word)

for i in range(10):
    position = random.randrange(min, max)
    print("word[", position, "] = ", word[position])
```

Вывод программы показан на рис. 13.1.

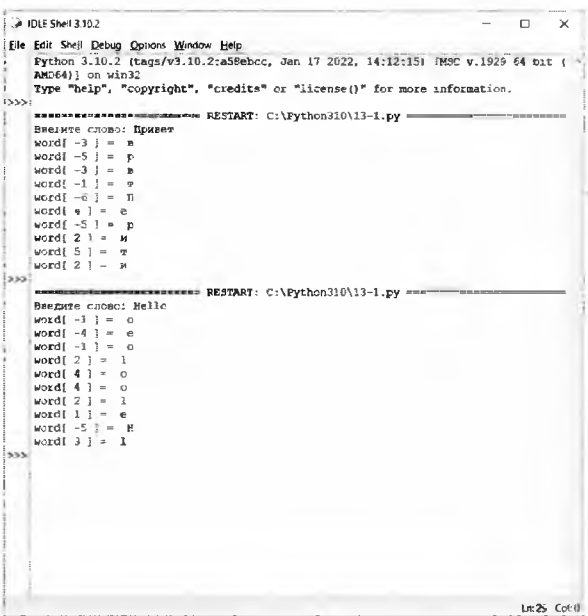


Рис. 13.1. Произвольный доступ к строке

13.3. Позиции с положительными и отрицательными номерами

Позиция, то есть индекс строки, может быть как положительной, так и отрицательным. Сначала поговорим о положительных индексах.

Как уже было отмечено, нумерация начинается с 0, поэтому у первого символа строки будет индекс 0, у второго – 1 и т.д. В целом все понятно, нужно помнить только, что нумерация начинается с 0.

Отрицательные индексы начинаются с конца строки – слева направо. Давайте продемонстрируем это.

Листинг 13.2. Положительные и отрицательные индексы

```
word = input("Введите слово: ")

max = len(word)
min = -len(word)

for i in range(min,max,1):
    print("word[", i, "] = ", word[i])
```

Посмотрим на программу. Пользователь вводил слово. Пусть это будет слово "Привет". Далее мы вычисляем минимум и максимум. Максимум равен длине введенного слова, минимум – отрицательной длине. В нашем случае максимум будет равен 6, минимум равен -6.

Далее в цикле *for* мы перебираем значения *i* – от -6 до 6. Вывод программы будет таким:

```
Введите слово: Привет
word[ -6 ] =  П
word[ -5 ] =  р
word[ -4 ] =  и
word[ -3 ] =  в
word[ -2 ] =  е
word[ -1 ] =  т
word[ 0 ] =  П
word[ 1 ] =  р
word[ 2 ] =  и
word[ 3 ] =  в
```

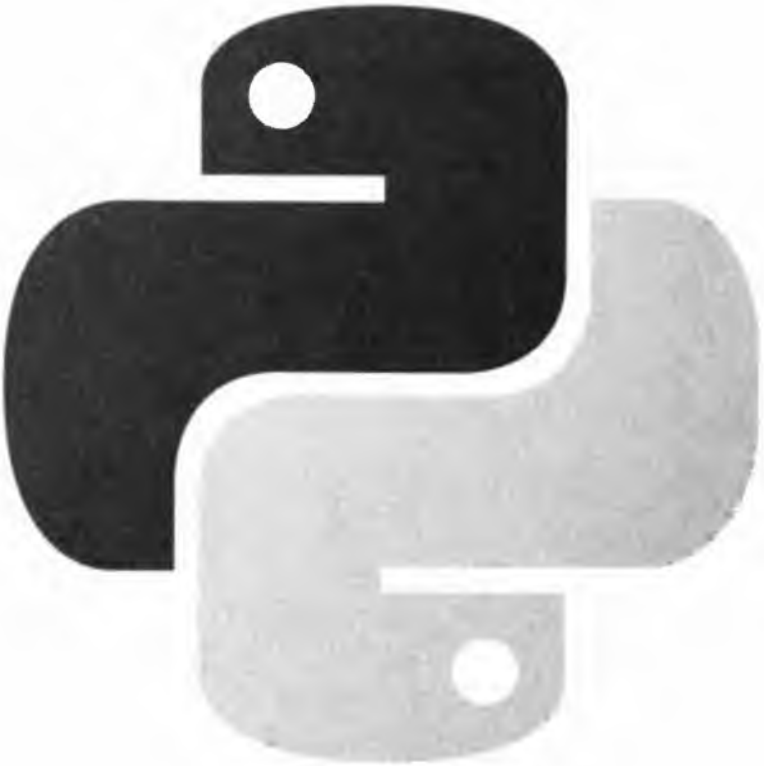
```
word[ 4 ] =  e
word[ 5 ] =  т
```

Нулевая позиция - первый символ строки, это будет буква "П", позиция 1 – буква "р" и т.д. Отрицательные индексы следуют слева направо, то есть -1 – это буква "т", то есть последняя буква строки, -2 – предпоследняя – буква "е". Индексу -6 будет соответствовать первая буква строки – "П".

Следующая таблица (табл. 13.1) демонстрирует индексы строки на примере слова "Привет"

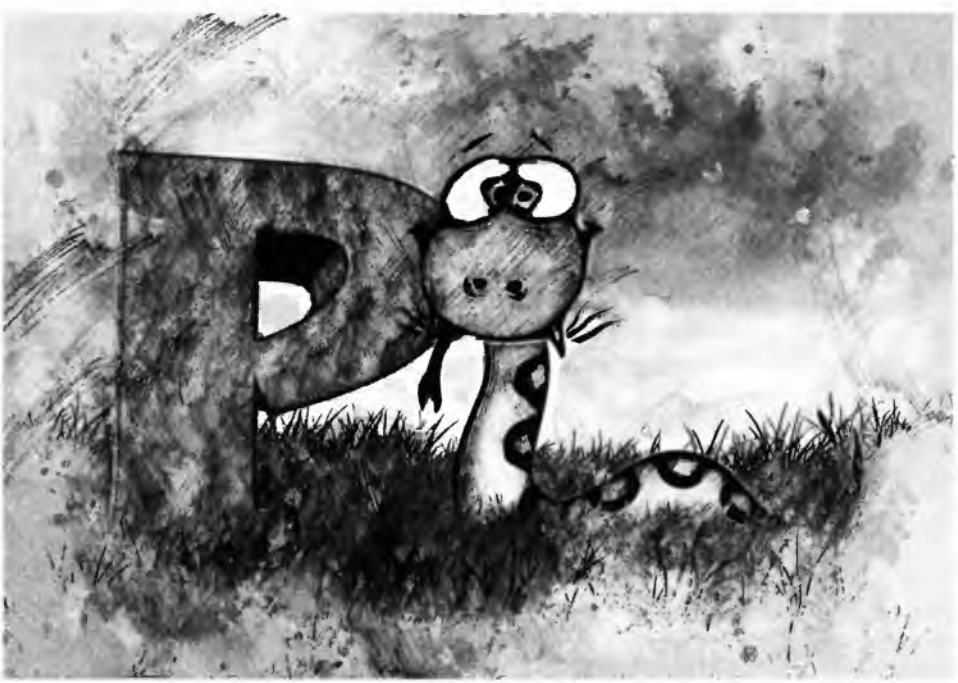
Таблица 13.1. Положительные и отрицательные индексы

-6	-5	-4	-3	-2	-1
П	**р**	**и**	**в**	**е**	**т**
0	1	2	3	4	5



Глава 14.

НЕИЗМЕНЯЕМОСТЬ СТРОКИ. ДЕМО-ПРОГРАММА «СРЕЗЫ»



14.1. Понятие неизменяемости строки

Все последовательности делятся на *изменяемые* и *неизменяемые*:

- Элементы изменяемой последовательности можно модифицировать;
- Элементы неизменяемой последовательности изменить нельзя.

Строки в Python – это пример неизменяемой последовательности.

Вы не можете изменить строку. Строка "Привет" всегда останется строкой "привет". Как-либо преобразовать ее невозможно. Но вы можете возразить, мол, как так. Если можно легко сделать так:

```
>>> message = "Привет"
>>> print(message)
Привет
>>> message = "Пока"
>>> print(message)
Пока
```

Ведь у нас же получилось изменить строку *message*! Ведь было значение "Привет", а стало – "Пока". На самом деле Python и не пытался менять строку "Привет". Он просто создал строку "Пока" и присвоил это значение переменной *message*.

Рисунок 14.1 наглядно демонстрирует то, что произошло сейчас.

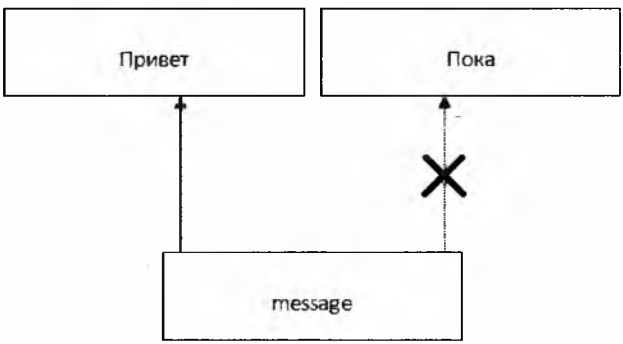


Рис. 14.1. Была создана новая строка, и это значение было присвоено переменной message

Неизменяемость строк в Python означает одну неприятную вещь: вы не можете присвоить новое значение символу строки, обращаясь к нему по индексу, как это можно сделать в других языках программирования.

Рассмотрим пример:

```
>>> message = "Hello"
>>> print(message[1])
e
>>> message[1] = "a"
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    message[1] = "a"
TypeError: 'str' object does not support item assignment
>>>
```

Как видите, у нас есть строка *message*. Вы можете обратиться к message[1] и прочитать это значение. Но вы не можете присвоить message[1] новое значение – получите длинное сообщение об ошибке. Интерпретатор сообщает, что тип *str* не поддерживает поэлементное назначение.

14.2. Создание новой строки

Рассмотрим создание новой строки на примере программы, которая принимает исходную строку и создает новую на основании введенной, но новая строка будет содержать только согласные буквы. Ранее мы уже писали похожую программу, но здесь мы применим немного другой способ определения того, является ли буква согласной или гласной.

Листинг 14.1. Создание новой строки

```
vowels = "аоиеёыуэюя"

text = input("Введите строку: ")

new_text = ""

for letter in text:
    if letter.lower() not in vowels:
        new_text += letter
        print("Создана новая строка:", new_text)

print()
print("Окончательный результат: ", new_text)
```

Программа работает так. Сначала мы формируем переменную *vowels*, содержащую гласные буквы русского языка. Затем мы проходимся по всем буквам введенного пользователем текста. Если буква (в нижнем регистре) отсутствует (*not in*) в строке *vowels*, то мы создаем новую строку путем добавления гласного символа к переменной *new_text*. Может казаться, что мы просто добавляем новый символ, но на самом деле мы каждый раз (при каждой итерации) создаем новую строку.

Вывод программы будет таким:

```
Введите строку: Привет всем!
Создана новая строка: П
Создана новая строка: Пр
Создана новая строка: Прв
Создана новая строка: Првт
Создана новая строка: Првт
Создана новая строка: Првт в
Создана новая строка: Првт вс
Создана новая строка: Првт всм
Создана новая строка: Првт всм!

Окончательный результат:  Првт всм!
```

Для определения, является ли символ гласным, мы использовали оператор *in*, а не оператор *if*, как в программе "Анализатор слова". Данный подход гораздо компактнее и правильнее, чем использование оператора *if*.

Интерпретатор Python очень скрупулезен в работе со строками и символами. Он придирается к регистру символов и "О" – это не "о". Поэтому мы в нашем коде сначала приводим символ к нижнему регистру, а затем уже ищем его в переменной *vowels*.

> **Внимание!** Необходимость изменения регистра (приведения его к нижнему или верхнему регистру) возникает довольно часто. Например, вы просите пользователя подтвердить действие и ввести Y. Пользователь может ввести *y* – он не будет вводить Y – лишний раз нажимать **Shift** не хочется. Конечно, можно заставить его это сделать, но гораздо проще перевести его ввод в нижний регистр и сравнить с 'y'.

Для добавления символа к уже существующей строке мы используем оператор +=:

```
new_text += letter
```

Еще раз напоминаем, что каждый раз при таком присваивании создается новая строка на базе уже существующей.

14.3. Срезы строк

Индексирование дает возможность выбирать не обязательно только один элемент из последовательности. Язык Python позволяет создать копии непрерывных подпоследовательностей. Такие копии называются срезами.

При выборе только одного элемента, например, word[1], создается срез, состоящий из одного элемента. Но вы можете выбрать несколько элементов строки, например, выбрать начало строки (несколько символов в начале), середину или конец. При желании можно создать срез-копию всей исходной последовательности.

По сути, срез строки – это получение подстроки. При получении среза вы задаете начальную и конечные позиции, например:

```
word = "привет"
print(word[1:3])          # выведет "ри"
```

Рассмотрим еще немного примеров:

```
>>> s = 'spameggs'
>>> s[3:5]
'me'
>>> s[2:-2]
'ameg'
>>> s[:6]
'spameg'
>>> s[1:]
'pameggs'
>>> s[:]
'spameggs'
```

Кроме того, можно задать шаг, с которым нужно извлекать срез.

```
>>> s[::-1]
'sggemaps'
>>> s[3:5:-1]
''
>>> s[2::2]
'aeg'
```

Давайте посмотрим, как срезы используются на практике. При работе с демо-программой ориентируйтесь на таблицу 14.1. Она подскажет вам индексы слова.

Таблица 14.1. Индексы слова

-6	-5	-4	-3	-2	-1
П	**р**	**и**	**в**	**е**	**т**
0	1	2	3	4	5

Листинг 14.2. Демо-программа Срезы

```
word = "привет"

print("Демо-программа Срезы")
print("Введите начальный и конечный индексы для среза слова
'привет'")

begin = None
while begin != "":
    begin = input("Начальная позиция: ")
    if begin:
        begin = int(begin)
        end = int(input("Конечная позиция: "))
        print(word[begin:end])
```

Вывод программы будет следующим (заодно посмотрите, как работают срезы):

```
Демо-программа Срезы
Введите начальный и конечный индексы для среза слова 'привет'
Начальная позиция: 1
Конечная позиция: 3
ри
Начальная позиция: 0
Конечная позиция: 3
при
Начальная позиция: 0
Конечная позиция: 4
прив
Начальная позиция: 4
Конечная позиция: 5
е
```

```
Начальная позиция: 2
Конечная позиция: 5
иве
Начальная позиция: -6
Конечная позиция: 6
привет
Начальная позиция: -6
Конечная позиция: 0

Начальная позиция: -1
Конечная позиция: -6

Начальная позиция: -6
Конечная позиция: -1
приве
Начальная позиция: -6
Конечная позиция: 0
```

Проанализируем программу. Первым делом мы присваиваем начальной позиции значение *None*:

```
begin = None
```

Данное ключевое слово в Python представляет пустое значение. Означает только одно – значение еще не присвоено. В условной интерпретации *None* рассматривается как *False*. В этой программе мы используем *None* для инициализации переменной *begin* – чтобы ее можно было использовать в условии цикла *while*.

В цикле пользователь вводит начальную и конечную позицию, которые потом используются так:

```
        print(word[begin:end])
```

Как и при индексировании, можно использовать отрицательные номера. При желании можно даже сочетать отрицательную нумерацию с положительной. Поэкспериментируйте с программой "Срезы", чтобы понять, как работают срезы в Python. Вполне возможно, сразу вы будете плохо ориентироваться в индексах, особенно, отрицательных. Но потратив минут 10 на работу

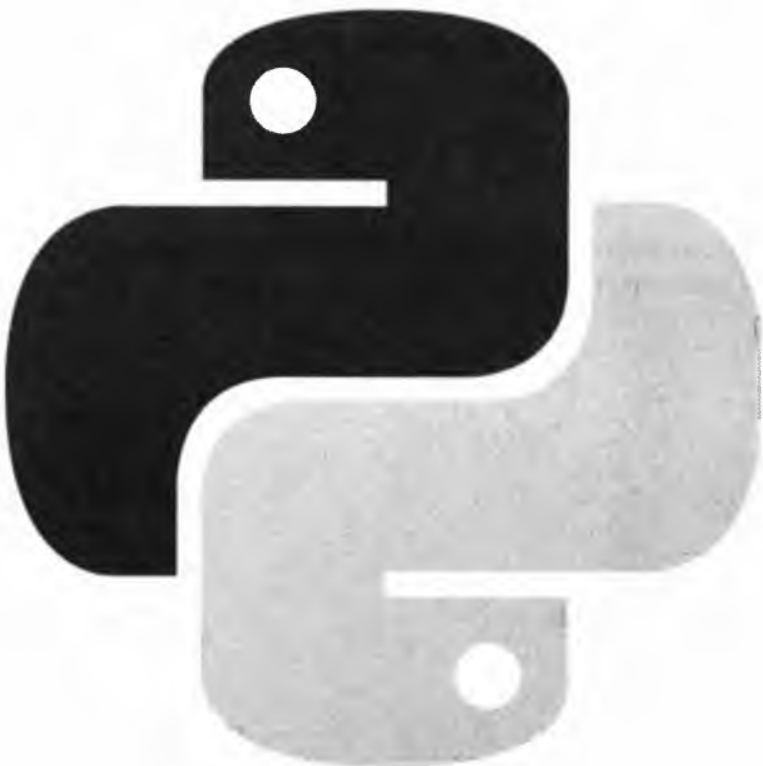
с демо-программой, вы сможете обходиться уже без нее – формирование срезов будет происходить в голове автоматически.

> **Примечание.** Если вызвать невозможный срез, в котором начальная позиция имеет больший номер, чем конечная или же заданы несуществующие индексы, то интерпретатор не выдаст ошибку, а просто вернет пустую последовательность.

При задании срезов можно использовать сокращенную запись, например:

```
word[:2]
word[1:]
```

Если не задана начальная позиция, то подразумевается начало строки, если не задана конечная, то подразумевается конец строки.



Глава 15.

КОРТЕЖИ

15.1. Что такое кортежи

Кортежи – это еще один из типов *последовательностей*. Но в отличие от строк, которые состоят только из символов, кортежи могут содержать элементы любой природы. В кортеже вы можете хранить, например, фамилии сотрудников, марки автомобилей, номера телефонов и т.д.

Самое интересное, что элементы кортежа не обязательно должны относиться к одному типу. При желании вы можете создать кортеж, содержащий, как строковые, так и числовые значения. Вообще кортеж может содержать все, что угодно – хоть звуковые файлы.

15.2. Создание кортежей

Создать кортеж можно так:

```
cars = ("Nissan", "Toyota", "Lexus")
drivers = ()      # Создает пустой кортеж
```

Первая строка создает непустой кортеж, состоящий из трех элементов. Вторая строка создает пустой кортеж.

15.3. Перебор элементов кортежа

Вывести содержимое кортежа можно функцией print():

```
print(cars)
```

Перебрать все элементы кортежа и что-то сделать с ними:

```
for item in cars:
    print(item)
```

15.4. Кортеж как условие

Кортеж можно использовать как условие, например:

```
if not cars:
  print("У вас нет автомобиля!")
```

Пустой кортеж интерпретируется как *ложное условие* (*False*), а кортеж, содержащий хотя бы один элемент – как *истинное*. Поскольку пустой кортеж интерпретируется как *False*, то условие *not cars* оказывается истинным, поэтому программа выведет строку "У вас нет автомобиля".

15.5. Функция len() и оператор *in*

К кортежам может применяться функция len(), возвращающая число элементов кортежа:

```
print("Сейчас у вас ", len(cars), " автомобилей")
```

Проверить существование элемента в кортеже можно так:

```
if "Nissan" in cars:
  print("В вашем гараже есть Nissan!")
```

15.6. Индексация и срезы кортежей

Кортежи поддерживают индексацию. Индексы назначаются так же, как и в случае со строками: первому элементу кортежа соответствует индекс 0, второму – 1 и т.д. Поддерживается и отрицательная индексация:

```
print("Ваш первый автомобиль: ", cars[0])
```

Программист может также создать срезы кортежа. Принцип такой, как и со строками.

15.7. Неизменность кортежей и слияния

О кортежах вам нужно знать еще две вещи. Первая – они, как и строки, неизменяемые:

```
>>> cars = ("nissan", "toyota")
>>> print(cars[0])
nissan
>>> cars[0] = "ford"
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    cars[0] = "ford"
TypeError: 'tuple' object does not support item assignment
>>>
```

То есть вы не можете присвоить другое значение элементу кортежа.

Вторая вещь – кортежи поддерживают слияния. *Слияние кортежей* еще называют *сцеплением*. Чтобы выполнить сцепление кортежей, нужно использовать оператор +.

15.8. Программа "Автомобили"

Программа "Автомобили" демонстрирует все операции с кортежами (листинг 15.1).

Листинг 15.1. Программа "Автомобили"

```
# Пустой кортеж, кортеж как условие

cars = ()
if not cars:
    print("У вас пока нет автомобиля")

# Вывод элементов кортежа

cars = ("nissan", "toyota", "lexus")

print("Выводим кортеж функцией print()")

print(cars)

print("Поэлементный перебор кортежа")

for item in cars:
    print(item)

# len() и in

print("Сейчас у вас ", len(cars), " автомобилей")

car = input("Поиск авто: ")

if car in cars:
    print("Автомобиль найден!")
else:
    print("Автомобиля ", car, " у вас нет")

# Слияние кортежей

new_cars = ("ford", "audi")

all_cars = cars + new_cars

print("Результат слияния: ")
```

```
print(all_cars)
```

Вывод программы будет следующим:

```
У вас пока нет автомобиля
Выводим кортеж функцией print()
('nissan', 'toyota', 'lexus')
Поэлементный перебор кортежа
nissan
toyota
lexus
Сейчас у вас  3  автомобилей
Поиск авто: ford
Автомобиля  ford  у вас нет
Результат слияния:
('nissan', 'toyota', 'lexus', 'ford', 'audi')
```

15.9. Распаковка кортежа в отдельные переменные

Представим, что у нас есть кортеж из N элементов, который вы хотите "распаковать" в набор из N переменных.

Любая последовательность может быть распакована в переменные с использованием простой операции присваивания. Требование только одно: чтобы число переменных соответствовало числу элементов в структуре. Например:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'Den', 50, 91.1, (2017, 12, 21) ]
>>> name, shares, price, date = data
>>> name
```

```
'Den'
>>> date
(2017, 12, 21)
>>> name, shares, price, (year, mon, day) = data
>>> name
'Den'
>>> year
2017
>>> mon
12
>>> day
21
>>>
```

Если будет несоответствие в числе элементов, то вы получите ошибку. Например:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
  >>>
```

Фактически, распаковка работает с любым объектом, который является итерируемым, а не только с кортежами или списками. К таким объектам относятся строки, файлы, итераторы и генераторы. Например:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

При распаковке иногда бывает нужно отбросить определенные значения. У Python нет специального синтаксиса для этого, но вы можете просто указать имя переменной для значений, которые нужно отбросить. Например:

```
>>> data = [ 'Den', 50, 91.1, (2017, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

Однако убедитесь, что имя переменной, которое вы выбираете, не используется для чего-то еще.

Ситуация усложняется, когда вам нужно распаковать N элементов из итерируемого объекта, который может быть длиннее, чем N, что вызывает исключение "*too many values to unpack*".

Для решения этой задачи может использоваться "звездочка". Например, предположим, что в конце семестра вы решаете отбросить первые и последние классы домашней работы и выполнить только их среднюю часть. Если классов только четыре, то можно распаковать все четыре, но, что если 24? Тогда все упрощает "звездочка":

```
def drop_first_last(grades):
  first, *middle, last = grades
  return avg(middle)
```

Рассмотрим и другой вариант использования. Предположим, что у вас есть записи, состоящие из имени пользователя и адреса электронной почты, сопровождаемые произвольным числом телефонных номеров. Вы можете распаковать эти записи так:

```
>>> record = ('Андрей', 'andrew@example.com', '25-333-26',
'888-12-11')
>>> name, email, *phone_numbers = user_record
>>> name
'Андрей'
>>> email
'andrew@example.com'
>>> phone_numbers
['25-333-26', ''888-12-11']
>>>
```

Стоит отметить, что переменная *phone_numbers* всегда будет списком, независимо от того, сколько телефонных номеров распаковано (даже если ни один). Таким образом, любой код, использующий *phone_numbers*, должен всегда считать ее списком или хотя бы производить дополнительную проверку типа.

Переменная со звездочкой может также быть первой в списке. Например, скажем, что у вас есть последовательность значений, представляющая объемы продаж вашей компании за последние 8 кварталов.

Если вы хотите видеть, как самый последний квартал складывается в средних по первым семи кварталам, вы можете выполнить подобный код:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

А вот как выглядит эта операция из интерпретатора Python:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Расширенная распаковка итерируемого объекта нестандартна для распаковки итерируемых объектов неизвестной или произвольной длины. Часто у таких объектов есть некоторый известный компонент или образец в их конструкции (например, "все, что после элемента 1 считать телефонным номером") и распаковка со звездочкой позволяет разработчику усиливать те образцы, чтобы получить соответствующие элементы в итерируемом объекте.

Стоит отметить, что *-синтаксис может быть особенно полезным при итерации по последовательности кортежей переменной длины. Например, возможно у нас есть последовательность теговых кортежей:

```
records = [
('foo', 1, 2),
('bar', 'hello'),
('foo', 3, 4),
```

```
]

def do_foo(x, y):
  print('foo', x, y)

def do_bar(s):
  print('bar', s)

for tag, *args in records:
  if tag == 'foo':
    do_foo(*args)
  elif tag == 'bar':
    do_bar(*args)
```

Распаковка со звездочкой может также быть полезной, когда объединена с определенными видами операций обработки строк, например, при *разделении строки* (*splitting*). Например:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/
bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Иногда нужно распаковать значения и отбросить их. Вы не можете указать пустое место с * при распаковке, но вы можете использовать звездочку вместе с переменной _. Например:

```
>>> record = ('Den', 50, 123.45, (12, 18, 2017))
>>> name, *_, (*_, year) = record
>>> name
'Den'
>>> year
2017
>>>
```

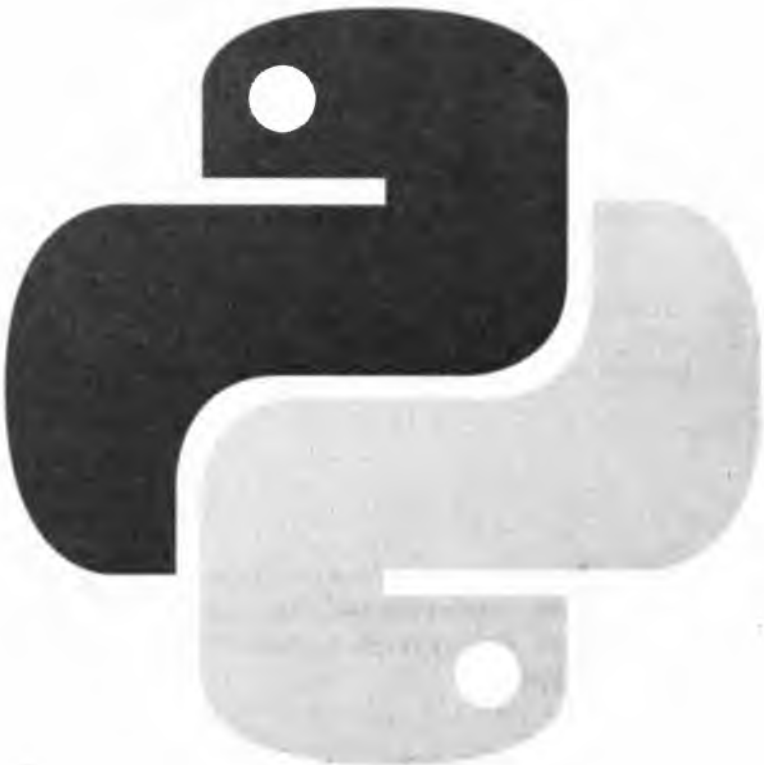
Есть определенная схожесть между звездообразными функциями распаковки и обработки списков различных функциональных языков. Например, если у вас есть список, вы можете легко разделить его на компоненты головы и хвоста. Вот пример:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

Можно было предположить написание функций, выполняющих такое разделение, в виде некоторого умного рекурсивного алгоритма. Например:

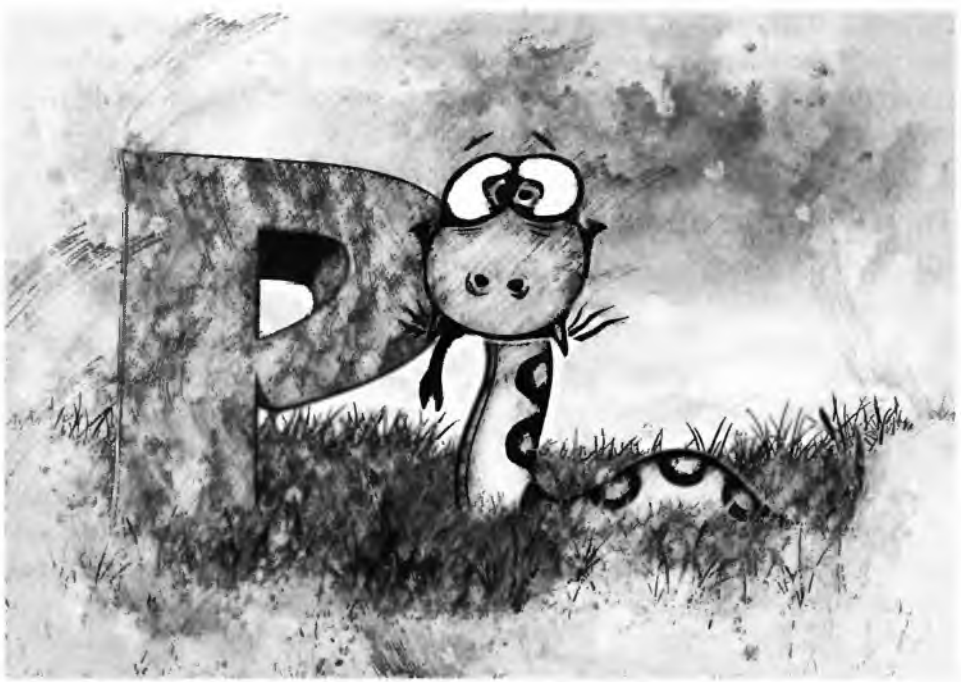
```
>>> def sum(items):
... head, *tail = items
... return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

Однако знайте, что рекурсия действительно не сильная функция Python из-за свойственного ей предела. Таким образом, этот последний пример приведен только из академического любопытства, на практике вы вряд ли будете использовать рекурсию в Python.



Глава 16.

ПРАКТИЧЕСКИЕ ПРИМЕРЫ РАБОТЫ СО СТРОКАМИ



В предыдущих главах мы познакомились со строками и рассмотрели основные операции. Данная глава – сборник практической примеры работы со строками. В ней вы найдете примеры кода на любой случай жизни (разумеется, если такой случай связан со строками).

16.1. Разделение строк с использованием разделителей

Представим, что у нас есть строка и нам ее нужно разделить на поля, используя разделители, например, пробелы. Чтобы задача была менее тривиальной, будем считать, что разделители и пробелы вокруг них противоречивы по всей строке.

В самых простых случаях можно использовать метод split() объекта строки. Но он не позволяет использовать несколько разделителей и учитывать возможное пространство вокруг разделителей. В нашем случае нужна большая гибкость, поэтому мы будем использовать метод **split()** из модуля **re**:

```
>>> line = 'asdf fjdk; afed, fjek,asdf, foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

Метод re.split() полезен, потому что вы можете определить многократные образцы для разделителя. например, как показано в решении, разделитель может быть или запятой (,) или точкой с запятой (;) или пробелом, сопровождаемым любым количеством дополнительных пробельных символов. Каждый раз, когда найден образец, все соответствия ему становятся разделителем между любыми полями, лежащими по обе стороны от соответствия. Результат – список полей, как и в случае с str.split().

При использовании re.split() нужно быть осторожным, образец регулярного выражения должен содержать группу захвата, заключенную в круглые скобки. Если используются группы захвата, то соответствующий текст также будет включен в результат. Например:

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf',
',', 'foo']
>>>
```

Получение символов разделителя может быть полезным в определенных контекстах. Например, возможно, вы нуждаетесь в символах разделителя, чтобы преобразовать выходную строку:

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']

>>> # Преобразуем строку с использованием тех же разделителей
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

Если вы не хотите получить символы разделителя в результате, но вам все еще нужны круглые скобки для группировки частей образца регулярного выражения, убедитесь, что вы используете группы не захвата, которая определяется как (?...). Например:

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

16.2. Использование маски оболочки

Этот пример показывает, как найти соответствие текста с использованием масок, которые часто используются при работе с Unix-оболочкой (к примеру, *.py, Dat[0-9]*.csv и т.д.).

Для решения поставленной задачи мы используем модуль **fnmatch**, предоставляющий две функции – fnmatch() и fnmatchcase(), которые могут использоваться для решения поставленной задачи. Использование функций очень простое:

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('rep.txt', '*.txt')
True
>>> fnmatch('rep.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')

True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'rep.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

Шаблоны fnmatch() используют те же правила, что и маски используемой файловой системы (которая зависит от используемой операционной системы в свою очередь). Пример:

```
>>> # B OS X (Mac)
>>> fnmatch('rep.txt', '*.TXT')
False
>>> # B Windows
>>> fnmatch('rep.txt', '*.TXT')
True
>>>
```

Если важен регистр символов, то используйте fnmatchcase(). Метод fnmatchcase() различает строчные и прописные символы, которые вы предоставляете:

```
>>> fnmatchcase('rep.txt', '*.TXT')
False
>>>
```

Функциональность fnmatch() находится где-то между функциональностью простых строковых методов и полной мощью регулярных выражений. Если вам нужно обеспечить простой механизм для разрешения подстановочных символов в операциях обработки данных, часто – это разумное решение.

> **Примечание.** Об улучшении производительности при работе со строками в Python 3.10. В Python 3.10 произведено улучшение производительности: оптимизация конструкторов str(), bytes() и bytearray(), которые должны стать примерно на 30% быстрее (фрагмент, адаптированный из примера в баг-трекере Python):

```
~ $ ./python3.10 -m pyperf timeit -q --compare-to=python "str()"
Mean +- std dev: [python] 81.9 ns +- 4.5 ns -> [python3.10] 60.0 ns +- 1.9
ns: 1.36x faster (-27%)
~ $ ./python3.10 -m pyperf timeit -q --compare-to=python "bytes()"
Mean +- std dev: [python] 85.1 ns +- 2.2 ns -> [python3.10] 60.2 ns +- 2.3
ns: 1.41x faster (-29%)
~ $ ./python3.10 -m pyperf timeit -q --compare-to=python
"bytearray()"
Mean +- std dev: [python] 93.5 ns +- 2.1 ns -> [python3.10] 73.1 ns +- 1.8
ns: 1.28x faster (-22%)
```

Другой более заметной оптимизацией (если вы используете аннотации типов) является то, что параметры функции и их аннотации вычисляются уже не во время исполнения, а во время компиляции. Теперь функция с аннотациями параметров создается примерно в два раза быстрее.

16.3. Совпадение текста в начале и конце строки

Пример показывает, как произвести поиск в начале и конце строки определенного шаблона текста, например, расширение имени файла, название протокола и т.д.

Самый простой способ заключается в использовании методов str.startswith() и str.endswith(). Например:

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

Если вам нужно проверить на соответствие нескольким вариантам, просто предоставьте кортеж возможных вариантов функциям startswith() или endswith():

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h'))
]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

16.4. Регулярные выражения

Довольно часто на практике возникает необходимость найти текст, соответствующий определенному шаблону. Если искомый текст является простым литералом, чаще проще использовать базовые строковые методы, например, str.find(),str.endswith(), str.startswith() и т.п. Например:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> # Точное совпадение
>>> text == 'yeah'
False
>>> # Совпадение в начале или в конце
>>> text.startswith('yeah')
```

```
True
>>> text.endswith('no')
False
>>> # Поиск позиции первого вхождения искомого текста в строку
>>> text.find('no')
10
>>>
```

Для более сложного соответствия используйте регулярные выражения и модуль **re**. Чтобы проиллюстрировать основную механику использования регулярных выражений, предположим, что вы хотите найти даты, определенные как цифры, например, "02/24/2022". Вот как это можно сделать:

```
>>> text1 = '02/24/2022'
>>> text2 = 'Feb 24, 2022'
>>>
>>> import re
>>> # Простое соответствие: \d+ означает соответствие 1 или более цифрам
>>> if re.match(r'\d+/\d+/\d+', text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
... print('yes')
... else:

... print('no')
...
no
>>>
```

Если нужно выполнить больше соответствий, используя тот же образец, имеет смысл скомпилировать образец регулярного выражения в объект. Например:

```
>>> datepat = re.compile(r'\d+/\d+/\d+')

>>> if datepat.match(text1):
... print('yes')
... else:
... print('no')
...
```

```
yes
>>> if datepat.match(text2):
... print('yes')
... else:
... print('no')
...
no
>>>
```

Метод match() всегда пытается найти совпадение в начале строки. Если вам нужно найти все вхождения, используйте метод findall(). Пример:

```
>>> text = 'Сегодня 11/27/2022. Следующий День рождения 3/13/2023.'
>>> datepat.findall(text)
['10/10/2022', '3/13/2023']
>>>
```

При определении регулярных выражений принято представлять группы захвата, заключая части образца в круглые скобки. Например:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

Группы захвата часто упрощают последующую обработку соответствующего текста, потому что содержание каждой группы может быть извлечено индивидуально. К примеру:

```
>>> m = datepat.match('02/24/2022')
>>> m
<_sre.SRE_Match object at 0x1005d2750>
>>> # Извлекаем содержимое каждой группы
>>> m.group(0)
'02/24/2022'
>>> m.group(1)
'02'
>>> m.group(2)
'24'
>>> m.group(3)
'2022'
>>> m.groups()
```

```
('02', '24', '2022')
>>> month, day, year = m.groups()
>>>
>>> # Поиск всех соответствий
>>> text
'Сегодня 02/24/2022. Следующий День рождения 3/13/2023.'
>>> datepat.findall(text)
[('2', '24', '2022'), ('3', '13', '2023')]
>>> for month, day, year in datepat.findall(text):
... print('{}-{}-{}'.format(year, month, day))
...
2022-2-24
2023-3-13
>>>
```

Метод findall() ищет текст и находит все соответствия, возвращая их как список. Если вы хотите найти соответствия итеративно, используйте метод finditer(). Например:

```
>>> for m in datepat.finditer(text):
... print(m.groups())
...
('2', '24', '2022')
('3', '13', '2023')
>>>
```

Обсуждение теории регулярных выражений выходит за рамки этой книги. Однако этот рецепт иллюстрирует абсолютные основы использования модуля **re**. Сначала нужно скомпилировать образец, используя re.compile(), а затем использовать методы, такие как match(), findall() или finditer().

При указании шаблонов, как правило, принято использовать обычные строки, такие как r'(\d+)/(\d+)/(\d+)'. В таких строках обратный слеш остается неинтерпретируемым, что может быть полезно в контексте регулярных выражений. Иначе вам придется использовать двойной обратный слеш, например, '(\\d+)/(\\d+)/(\\d+)'.

Знайте, что метод match() проверяет только начало строки. Возможно, это немного не то, что вы ожидаете. Например:

```
>>> m = datepat.match('11/27/2023abcdef')
>>> m
```

```
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2023'
>>>
```

Если вам нужно точное совпадение, убедитесь, что шаблон в конце содержит маркер $, например:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2023abcdef')
>>> datepat.match('11/27/2023')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

Наконец, если вы просто выполняете простой поиск текста, вы можете часто пропустить шаг компиляции и использовать функции уровня модуля в модуле re вместо этого. Например:

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('2', '24', '2022'), ('3', '13', '2023')]
>>>
```

Знайте, тем не менее, что если вы собрались произвести значительное соответствие или поиск по большому тексту, обычно имеет смысл сначала скомпилировать образец, чтобы использовать его снова и снова. Так вы получите значительный прирост производительность.

16.5. Поиск и замена текста

В самых простых случаях используйте метод str.replace(). Например:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

Для более сложных образцов используйте функцию/метод sub() в модуле **re**. В качестве примера представим, что нужно перезаписать даты вроде "2/24/2022" так, чтобы они выглядели так: "2022-2-24".

Вот пример того, как это можно сделать:

```
>>> text = 'Сегодня 2/24/2022. Следующий День рождения 3/13/2023.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Сегодня 2022-2-24. Следующий День рождения 2023-3-13.'
>>>
```

Первый аргумент sub() – это шаблон, которой должен быть найден в тексте, а второй аргумент – это шаблон замены. Цифры с обратными слешами (например, \3) используются для группировки чисел в образце.

Если вы собираетесь выполнить повторные замены с тем же образцом, задумайтесь о его компиляции для лучшей производительности. Например:

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Сегодня 2022-2-24. Следующий День рождения 2023-3-13.'
>>>
```

Для более сложных замен можно использовать callback-функцию замены. Например:

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Сегодня 24 Feb 2022. Следующий День рождения 13 Mar 2023.'
>>>
```

В качестве ввода в callback-функцию замены передается объект, возвращенный функциями match() или find(). Используйте метод .group() для извле-

чения определенных частей соответствия. Функция должна вернуть текст замены.

Если вы знаете, сколько замен было сделано в дополнение к получению текста замены, используйте вместо этого re.subn(). Например:

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Сегодня 2022-2-24. Следующий День рождения 2023-3-13.'
>>> n
2
>>>
```

Для выполнения нечувствительных к регистру операций над текстом вам нужно использовать модуль **re** и установить флаг re.IGNORECASE. Этот флаг нужно устанавливать отдельно для каждой операции над текстом. Например:

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

Последний пример показывает ограничение, что текст замены не будет соответствовать регистру в исходном тексте. Если вам нужно исправить это, вам, возможно, придется использовать функцию поддержки, например:

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
```

```
        else:
            return word
return replace
```

Вот пример использования этой последней функции:

```
>>> re.sub('python', matchcase('snake'), text, flags=re.
IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

16.6. Удаление нежелательных символов из строки

Часто требуется удалить нежелательные символы, например, пробелы в начале и конце строки.

Метод strip() может быть использован для удаления символов в начале или конце строки. Функции lstrip() или rstrip() осуществляет обрезку символов слева или справа соответственно.

По умолчанию эти методы удаляют пробельные символы, но вы можете задать любые другие символы. Например:

```
>>> # Обрезаем пробельные символы
>>> s = ' hello world \n'
>>> s.strip()
'hello world'

>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'
>>>

>>> # Удаляем заданные символы
>>> t = '-----hello====='
>>> t.lstrip('-')
'hello====='
```

```
>>> t.strip('-=')
'hello'
>>>
```

Различные strip()-методы обычно используются при чтении и очистке данных от дальнейших для дальнейшей обработки. К слову, вы можете использовать их, чтобы избавиться от пробелов, удалить кавычки и выполнить другие задачи. Знайте, что strip()-методы не применяются к любому тексту в середине строки. Например:

```
>>> s = ' hello world \n'
>>> s = s.strip()
>>> s
'hello world'
>>>
```

Если вам нужно сделать что-то во внутреннем пространстве, вам нужно использовать другую технику, например, использовать метод replace() или замену с использованием регулярных выражений. Вот как это выглядит:

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

Часто нужно объединить *строковые операции разделения* (*splitting*) с некоторым другим видом итеративной обработки, например, чтением строк данных из файла. Если это так, то вам пригодится использование генератора. Например:

```
with open(filename) as f:
     lines = (line.strip() for line in f)
     for line in lines:
     ...
```

Здесь выражение *lines = (line.strip() for line in f)* действует как своего рода преобразование данных. Это эффективно, потому что оно фактически снача-

ла не считывает данные ни в какой вид временного списка. Мы просто создает итератор, где все ко всем считанным строкам сразу применяется strip().

Для более расширенного стриппинга вам нужно использовать метод translate(). Для поучения дополнительной информации обратитесь к следующему рецепту.

16.7. Подставляем значения переменных в строку

В PHP вы можете указать имена переменных прямо в строке, и при выводе они будут заменены значениями этих переменных, например:

```
echo "Привет, $name";
```

В Python так сделать нельзя. Но мы можем попробовать решить поставленную задачу с использованием метода format(). Например:

```
>>> s = 'У пользователя {name} {n} сообщений.'
>>> s.format(name='Den', n=55)
'У пользователя Den 55 сообщений.'
>>>
```

Альтернативно, если значения подстановки действительно найдены в переменных, вы можете использовать комбинацию функций format_map() и vars():

```
>>> name = 'Den'
>>> n = 55
>>> s.format_map(vars())
'У пользователя Den 55 сообщений.'
>>>
```

Одна из особенностей vars() – то, что она работает с инстанциями. Например:

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Den',55)
>>> s.format_map(vars(a))
' У пользователя Den 55 сообщений.'
>>>
```

Недостаток функций format() и format_map() – то, что они плохо работают с отсутствующими значениями. Например:

```
>>> s.format(name='Den')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

Один из способов избежать этого – определить альтернативный класс словаря с методом __missing__(), как показано ниже:

```
class safesub(dict):
     def __missing__(self, key):
     return '{' + key + '}'
```

Теперь используем этот метод для обработки ввода для format_map():

```
>>> del n # Убедимся, что  n не определена
>>> s.format_map(safesub(vars()))
'У пользователя Den {n} сообщений.'
>>>
```

Если вы часто выполняете эти шаги в своем коде, вы можете скрыть процесс замены переменной небольшой служебной функцией, которая использует так называемый "взлом фрейма" (frame hack). Например:

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

Теперь мы можем делать вещи как это:

```
>>> name = 'Den'
>>> n = 55
>>> print(sub('Привет {name}'))
Привет Den
>>> print(sub('У вас {n} сообщений.'))
У вас 55 сообщений.
>>> print(sub('Предпочитаемый вами цвет - {color}'))
Предпочитаемый вами цвет - {color}
>>>
```

Отсутствие истинной интерполяции переменной в Python привело к множеству решений за эти годы. В качестве альтернативного решений вы также можете использовать иногда оператор форматирования %, например:

```
>>> name = 'Den'
>>> n = 55
>>> 'У %(name) %(n) сообщений.' % vars()
'У Den 55 сообщений.'
>>>
```

Вы также можете использовать шаблоны:

```
>>> import string
>>> s = string.Template('У $name $n сообщений.')
>>> s.substitute(vars())
'У Den 55 сообщений.'
>>>
```

Однако методы format() и format_map() более современны, чем любая из этих альтернатив, и вы должны предпочесть именно их. Одно из преимуществ использования format() – то, что вы также получаете все функции, связанные с форматированием строки (выравнивание, дополнение, форматирование чисел и т.д.), которые просто не возможные с альтернативами, например, с объектами Template.

Части этого рецепта также иллюстрируют несколько интересных расширенных функций. К примеру, малоизвестный метод __missing__(). В функции *safesub* этот метод используется, чтобы возвращать вместо отсутствующих

переменных их имена. Теперь вместо получения исключения *KeyError* вы просто получите в строке названия отсутствующих значений, что потенциально полезно при отладке.

ЧАСТЬ V.

СПИСКИ И СЛОВАРИ

В предыдущей части книги вы познакомились с кортежами. Кортеж – хороший способ хранения элементов разных типов, но поскольку кортеж неизменяем, использовать его не очень удобно. Списки снимают данное ограничение – они умеют все, что и кортежи, но предоставляют больше гибкости, поскольку программист может изменять значения элементов списка. Вы можете изменять значения, удалять элементы, добавлять новые элементы, а также сортировать список.

Также в этой части книги будут рассмотрены словари. Если список – это набор значений, то словарь – это набор пар значений. Начнем мы со списков как с более простой сущности.

Глава 17.

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ СПИСКОВ. ПРОГРАММА "ГАРАЖ"

17.1. Создание списка

Создать список можно точно также, как и кортеж, но вместо круглых скобок применяются квадратные:

```
cars = ["audi", "vw", "lexus"]
print("Посмотрим, что у нас в гараже: ")
for item in cars:
    print(item)
```

Вывод программы изображен на рис. 17.1.

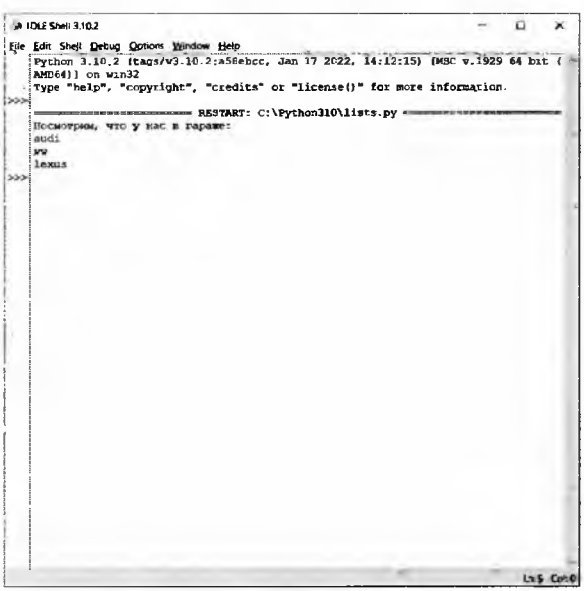


Рис. 17.1. Вывод списка

17.2. Функция len()

К списку можно применять функцию len():

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
print("В гараже ", len(cars), " автомобилей")
```

Вывод программы:

```
В гараже  5  автомобилей
```

17.3. Оператор *in*. Поиск в списке

Оператор *in* можно использовать для поиска по списку. Рассмотрим небольшой пример:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
car = input("Введите искомый автомобиль: ")
if car in cars:
    print("У вас есть такой автомобиль!")
else:
    print("У вас нет такого автомобиля :(")
```

Вывод программы:

```
Введите искомый автомобиль: vm
У вас нет такого автомобиля :(

Введите искомый автомобиль: vw
У вас есть такой автомобиль!
```

17.4. Индексация списков

Как в случае со строками и кортежами, списки индексируются. Ничего нового нет. Индексация начинается с 0 (то есть первый элемент списка имеет индекс 0), поддерживаются положительные и отрицательные индексы.

Пример работы с индексами:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]

start = -len(cars)
end = len(cars)

for i in range(start,end,1):
    print("cars[", i, "] = ", cars[i])
```

Вывод программы:

```
cars[ -5 ] =  audi
cars[ -4 ] =  vw
cars[ -3 ] =  lexus
cars[ -2 ] =  gtr
cars[ -1 ] =  m5
cars[ 0 ] =  audi
cars[ 1 ] =  vw
cars[ 2 ] =  lexus
cars[ 3 ] =  gtr
cars[ 4 ] =  m5
```

Сначала мы получаем начало (-len()) и конец (len()) диапазона. Потом проходимся по списку и указываем в качестве индекса переменную *i*, которая изменяется от *start* до *end* с приростом в 1.

Из вывода видно, что первому элементу списка присвоен индекс 0. Также к нему можно обратиться по индексу -len(cars), который в нашем случае равен -5.

17.5. Срезы списков

Списки поддерживают срезы. Принцип такой же, как и с кортежами: в квадратных скобках указываются начальный и конечный индексы среза:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]

start = int(input("Начальный индекс среза: "))
end = int(input("Конечный индекс среза: "))
```

```
print(cars[start:end])
```

Вывод программы:

```
Начальный индекс среза: 1
Конечный индекс среза: 4
['vw', 'lexus', 'gtr']
```

17.6. Сцепление списков

Как и кортежи, списки поддерживают сцепление. Для сцепления списков используется тот же оператор +:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
japan_cars = ["toyota", "nissan"]

cars = cars + japan_cars

print(cars)
```

Вывод программы:

```
['audi', 'vw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
```

17.7. Изменяемость списка. Присваиваем новые значения элементам списка

В отличие от кортежей, элементы списка можно изменять, то есть вы можете присвоить новое значение элементу, зная его индекс. Рассмотрим пример:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
japan_cars = ["toyota", "nissan"]
```

```
cars = cars + japan_cars

print(cars)

cars[0] = "bentley"

print(cars)
```

Здесь мы сначала выводим список *cars*, затем изменяем его первый элемент и снова выводим список *cars*. Вывод будет таким, как мы и ожидаем – в качестве первого элемента будет "bentley":

```
['audi', 'vw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
['bentley', 'vw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
```

При желании можно присваивать значение целому срезу списка, например:

```
print(cars)
cars[1:2] = ["bentley", "bmw"]
print(cars)
```

Вывод будет следующим:

```
['audi', 'vw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
['audi', 'bentley', 'bmw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
```

17.8. Удаление элементов списка

Списки поддерживаю удаление элементов, и даже удаление целых срезов. Например:

```
del cars[0]
print(cars)
del cars[:2]
print(cars)
```

Вывод:

```
['vw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
['gtr', 'm5', 'toyota', 'nissan']
```

17.9. Методы списков

17.9.1. Добавление и удаление элементов списка

Метод append() позволяет добавить элемент в список:

```
cars.append("kamaz")
```

Метод append() добавляет элемент в конец списка. Если нужно добавить элемент в позицию *i*, то нужно использовать метод insert():

```
cars.insert(i, "kamaz")
```

Для удаления используется метод remove:

```
cars.remove("kamaz")
```

Работает он так: если будет встречен элемент с заданным значением, то он будет удален. Удаляется только первый элемент, то есть, если в нашем списке есть несколько элементов со значением "kamaz", то будет удален только первый из них.

Если нужно удалить все элементы с заданным значением, то сначала нужно вычислить количество элементов списка, имеющих такое значения. В этом поможет метод count():

```
cars.count("kamaz")
```

Для удаления можно также использовать метод pop(). Он возвращает значение i-го элемента и удаляет его из списка:

```
cars.pop(i)
```

17.9.2. Сортировка – метод sort()

Метод sort() сортирует элементы списка по возрастанию. Если передать ему необязательный параметр *reverse*, равный *True*, то список будет отсортирован по убыванию.

Метод reverse() обращает порядок элементов списка. Обратите внимание: не сортирует элементы списка в обратном порядке, а обращает порядок.

Рассмотрим пример использования этих методов:

```
cars = ["audi", "vw", "lexus", "gtr", "m5"]
japan_cars = ["toyota", "nissan"]

cars = cars + japan_cars

print("Исходный список: ")
print(cars)
print("Сортируем список: ")
cars.sort()
print(cars)
print("Добавлем элемент kamaz: ")
cars.append("kamaz")
print(cars)
print("Обращаем порядок списка: ")
cars.reverse()
print(cars)
```

Вывод программы будет таким:

```
Исходный список:
['audi', 'vw', 'lexus', 'gtr', 'm5', 'toyota', 'nissan']
Сортируем список:
['audi', 'gtr', 'lexus', 'm5', 'nissan', 'toyota', 'vw']
Добавлем элемент kamaz:
['audi', 'gtr', 'lexus', 'm5', 'nissan', 'toyota', 'vw', 'kamaz']
Обращаем порядок списка:
['kamaz', 'vw', 'toyota', 'nissan', 'm5', 'lexus', 'gtr', 'audi']
```

Изначально наш список неотсортирован. Мы выполнили сортировку методом sort(). Далее мы добавили в конец списка элемент "kamaz". После этого

наш список опять превратился в неотсортированный. Метод reverse() не сортирует список, а просто изменяет порядок следования элементов на обратный.

17.9.3. Метод index()

Метод index() возвращает номер первого элемента списка, который содержит заданное значение:

```
cars.index("kamaz")
```

17.10. Программа "Гараж"

Программа "Гараж" демонстрирует практически все операции над списком:

- Добавление и удаление элементов
- Вывод списка
- Сортировка списка
- Поиск элемента в списке

Листинг 17.1. Программа "Гараж"

```
cars = []

print("*" * 10, " Гараж v.0.0.1 ", "*" * 10)

responce = 1
while responce:
    print("""Выберите действие:
           1 - Добавить автомобиль
           2 - Удалить автомобиль
           3 - Вывести список автомобилей
           4 - Найти автомобиль
           5 - Отсортировать гараж
           0 - Выйти""")
    responce = int(input(">> "))
    if responce == 1:
        car = input("Введите название авто: ")
        cars.append(car)
```

```
elif responce == 2:
    car = input("Введите название авто: ")
    cars.remove(car)
elif responce == 3:
    print(cars)
elif responce == 4:
    car = input("Введите название авто: ")
    if car in cars:
        print("Такой автомобиль есть в гараже!")
    else:
        print("Такого автомобиля нет в гараже!")
elif responce == 5:
        cars.sort()
        print("Сортировка выполнена!")
else:
        print("До скорых встреч!")
```

Программа работает так. В цикле мы выводим подсказку-меню. Пользователь вводит свой выбор, программа выполняет действия в зависимости от введенного номера действия. Предполагается, что пользователь будет вводить только цифры от 0 до 5. Обработка некорректного ввода не добавлялась для упрощения кода.

Рассмотрим несколько скриншотов. Изначально список авто пуст (рис. 17.2).

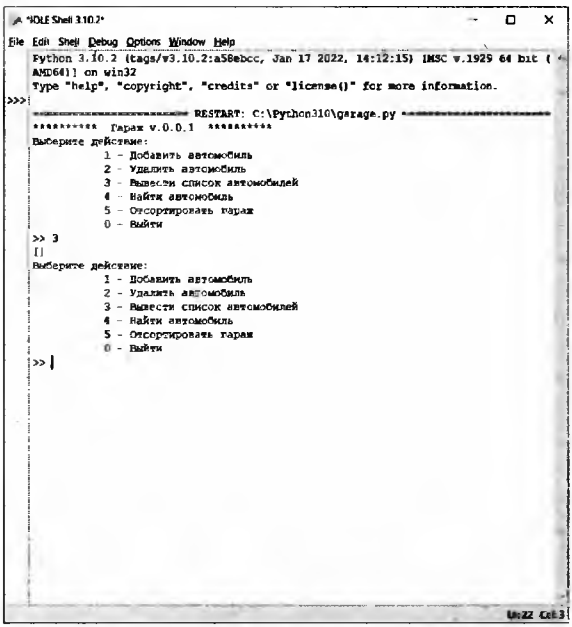


Рис. 17.2. В гараже нет авто

Программа просто выводит пустые скооки. В реальнои программе лучше бы сделать такую обработку:

```
if len(cars) > 0:
  for car in cars:
     print(car)
else:
  print("В гараже нет авто!")
```

Но у нас демо-программа для начинающих разработчиков, которые должны понимать, что [] соответствует пустому списку.

Далее добавим несколько автомобилей (рис. 17.3).

Рис. 17.3. Добавление автомобилей и вывод списка

Попробуем добавить и удалить "лишний" автомобиль:

Рис. 17.4. Добавление и удаление "kamaz"

Отсортируем список:

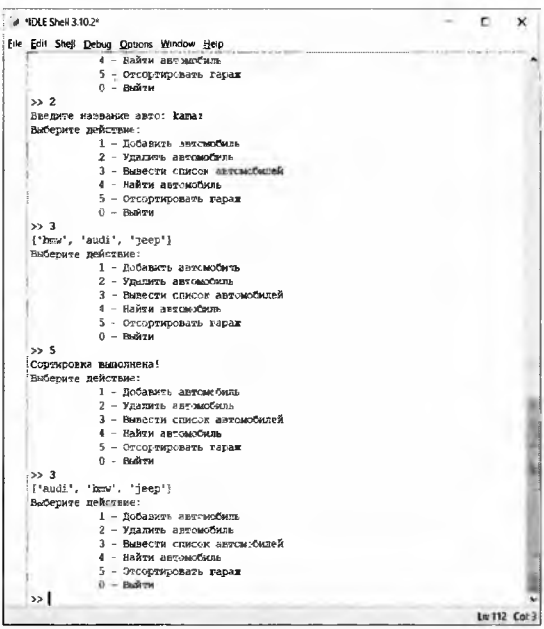


Рис. 17.5. Сортировка выполнена

Проверим поиск:

Рис. 17.6. Поиск авто

Поскольку мы удалили элемент "kamaz", то его не будет в списке, а вот элемент "jeep" есть в нашем гараже.

Проверим обработку выхода:

Рис. 17.7. Выход произведен

17.11. Списки vs кортежи

Мы только что рассмотрели, как работать со списками в Python. Возникает закономерный вопрос – когда лучше использовать списки, а когда – кортежи? Понятно, что списки – лучше кортежей, поскольку можно изменять элементы списка.

Но не спешите отказываться от кортежей. У них есть следующие преимущества:

- Кортежи работают быстрее. Система знает, что кортеж не изменится, поэтому его можно сохранит так, что операции с его элементами будут выполняться быстрее, чем с элементами списка. В небольших программа эта разница в скорости никак не проявит себя. Но при работе с большими последовательностями разница будет ощутимой.
- Неизменяемость кортежей позволяет использовать их как константы.
- Кортежи можно использовать в отдельных структурах данных, от которых Python требует неизменимых значений.
- Кортежи потребляют меньше памяти. Рассмотрим пример:

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> b = [1, 2, 3, 4, 5, 6]
>>> a.__sizeof__()
36
>>> b.__sizeof__()
44
```

- Кортежи можно использовать в качестве ключей словаря:

```
>>> d = {(1, 1, 1) : 1}
>>> d
{(1, 1, 1): 1}
>>> d = {[1, 1, 1] : 1}
Traceback (most recent call last):
  File "", line 1, in
    d = {[1, 1, 1] : 1}
TypeError: unhashable type: 'list'
```

Глава 18.

СЛОВАРИ. ПРОГРАММА «СЛОВАРЬ»

В отличие от кортежей и списков, которые хранят наборы элементов, словари хранят наборы пар элементов. Словари в Python можно сравнивать с классическими словарями, где каждому слову соответствует его определение, то есть образуется пара "слово: определение".

18.1. Создание словаря

Словарь определяется в фигурных скобках:

```
dict = {
    "apple" : "яблоко",
    "bold" : "жирный",
    "bus" : "автобус",
    "cat" : "кошка",
    "car" : "автомобиль"}
```

Пары *ключ : значение* разделяются двоеточием.

18.2. Доступ к значениям словаря

Вывести весь словарь можно так:

```
print(dict)
```

Правда, такой вывод сгодится разве что для отладки программы, а не для конечного результата пользователю:

```
{'cat': 'кошка', 'car': 'автомобиль', 'bus': 'автобус', 'bold':
'жирный', 'apple': 'яблоко'}
```

Пример вывода *значения* определенного *ключа* словаря:

```
print(dict["bus"])
```

А вот как можно вывести все ключи и значения словаря в более или менее удобном для пользователя формате:

```
for item in dict:
    print(item, " => ", dict[item])
```

Вывод будет таким:

```
cat  =>  кошка
car  =>  автомобиль
bus  =>  автобус
bold  =>  жирный
apple  =>  яблоко
```

Как видите, доступ к значениям словаря похож на индексирование, только здесь мы вместо индекса используем ключ, а не номер элемента.

Если обратиться к ключу, которого нет в словаре, то будет сгенерировано сообщение об ошибке:

```
Traceback (most recent call last):
  File "C:/Python310/dict.py", line 20, in <module>
    print(dict["bqus"])
KeyError: 'bqus'
```

18.3. Оператор *in*. Программа "Словарь v.0.0.1"

Оператор *in* можно использовать как в цикле *for* (что уже было показано ранее) – для перебора всех элементов словаря, так и для проверки наличия определенного элемента в словаря. Пример:

```
if "bus" in dict:
    print(dict["bus"])
else:
   print("Слова bus нет в словаре!")
```

Поскольку при обращении к несуществующему элементу словаря генерируется ошибка, то перед обращением неплохо бы проверить его наличие с помощью оператора *in*. Напишем простейшую программу поиска по словарю.

Листинг 18.1. Словарь

```
dict = {
    "apple" : "яблоко",
    "bold" : "жирный",
    "bus" : "автобус",
    "cat" : "кошка",
    "car" : "автомобиль"}

print("=" * 15, "Словарь", "=" * 15)

word = ""
while word != "q":
    word = input("Введите слово или q для выхода: ")
    if word != "q":
        if word in dict:
            print(dict[word])
        else:
            print("Нет такого слова в словаре")
```

Программа осуществляет поиск по словарю. Посмотрим, как она организована. Сначала мы определяем переменную *word*. Цикл *while* будет работать, пока эта переменная не равна "q".

В цикле пользователю предлагается ввести слово. Если слово не равно "q", то мы начинаем поиск по словарю. Если слово найдено, мы выводим его значение, если нет – то строку "Нет такого слова в словаре".

Если пользователь введет "q", то в цикле мы ничего не делаем, а при следующей итерации цикл будет прекращен.

Посмотрим на вывод программы (рис. 18.1). Программа работает корректно – если ввести слова, которого нет в словаре, не будет выведено сообщение об ошибке, а пользователь получит возможность ввести следующее слово.

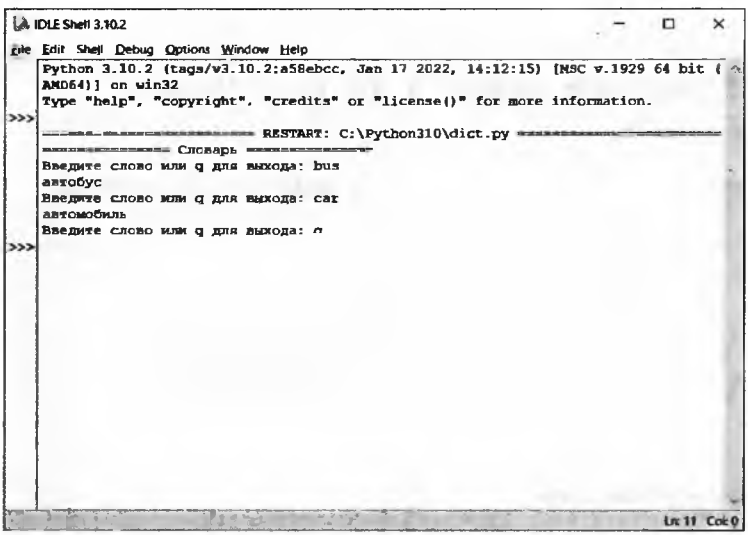


Рис. 18.1. Вывод программы

18.4. Метод get()

Метод get() представляет собой безопасный способ извлечения информации из словаря. При использовании этого метода вы можете не беспокоиться, что будет сгенерирована ошибка при обращении к несуществующему ключу. Более того, данный метод позволяет задать значение по умолчанию, которое будет возвращено в случае, если слово не будет найдено в словаре:

```
print(dict.get("test", "Нет такого слова в словаре!"))
```

18.5. Добавление пары "ключ-значение"

Добавить пару ключ-значение в процессе выполнения программы можно так:

```
словарь[ключ] = значение
```

Пример:

```
dict["test"] = "тест"
```

18.6. Удаление пары "ключ-значение"

Удалить пару "ключ-значение" можно с помощью оператора *del*:

```
del dict["bus"]
```

18.7. Методы словаря

Объект словаря поддерживает следующие методы:

- get(ключ [, значение]) – возвращает значение по ключу, если ключ в словаре не определен, будет возвращено указанное значение.
- keys() – возвращает набор всех ключей словаря.
- values() – возвращает набор всех значений словаря.
- items() – возвращает набор всех пар в словаре, каждая пара – это кортеж из двух элементов. Первый – это ключ, второй – это соответствующее ему значение.

18.8. "Словарь v 0.1"

Продолжим разработку нашего **Словаря**. Попробуем модифицировать исходную программу так, чтобы она поддерживала добавление и удаление элементов словаря, а также некоторые другие возможности.

Листинг 18.2. "Словарь v 0.1"

```
# Словарь заполнен по умолчанию
dict = {
    "apple" : "яблоко",
    "bold" : "жирный",
    "bus" : "автобус",
    "cat" : "кошка",
    "car" : "автомобиль"}

print("=" * 15, "Словарь v 0.1", "=" * 15)

# Справка. Будет выведена по команде h
help_message = """
s - Поиск слова в словаре
a - Добавить новое слово
r - Удалить слово
k - Вывод всех слов
d - Вывод всего словаря
h - Отображение этой подсказки
q - Выход
"""

choice = ""
while choice != "q":
    choice = input("(h - справка)>> ")
    if choice == "s":
        word = input("Введите слово: ")
        res = dict.get(word, "Нет такого слова!")
        print(res)
    elif choice == "a":
        word = input("Введите слово: ")
        value = input("Введите перевод: ")
        dict[word] = value
        print("Слово добавлено!")
    elif choice == "r":
        word = input("Введите слово: ")
        del dict[word]
```

```
        print("Слово ", word, " удалено")
    elif choice == "k":
        print(dict.keys())
    elif choice == "d":
        for word in dict:
            print(word, ": ", dict[word])
    elif choice == "h":
        print(help_message)
    elif choice == "q":
        continue;
    else:
        print("Нераспознанная команда. Введите h для справки")
```

В отличие от программы **Список**, здесь у нас будет словарь, заполненный данными по умолчанию. Сделано это, чтобы сразу можно было проверить работоспособность программы и не заполнять словарь данными сразу после запуска.

Основной цикл программы:

```
choice = ""
while choice != "q":
    choice = input("(h - справка)>> ")
```

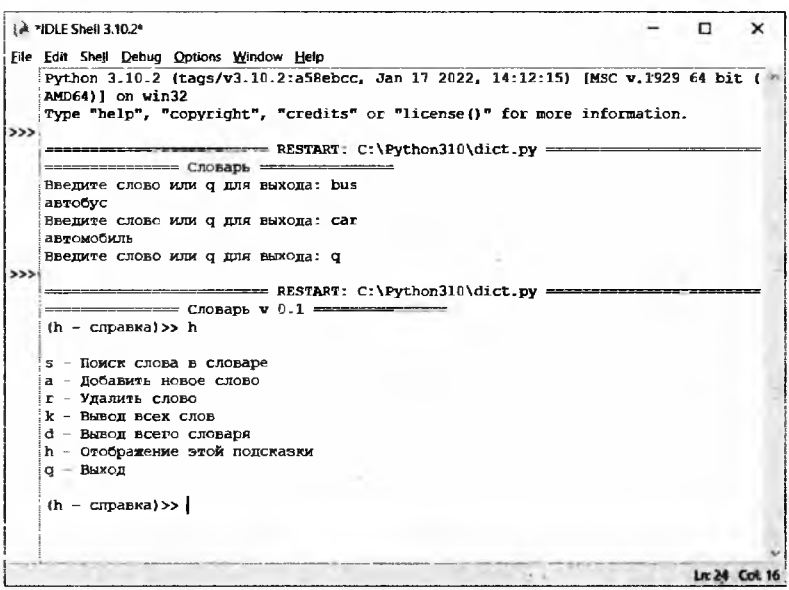


Рис. 18.2. Словарь, вывод справки

При каждой итерации мы выводим подсказку (h – справка)>> и читаем ввод пользователя. Справка, а именно доступные команды отображаются по команде *h* (рис. 18.2).

Поиск слова в словаре мы производим с помощью метода get():

```
if choice == "s":
    word = input("Введите слово: ")
    res = dict.get(word, "Нет такого слова!")
    print(res)
```

Добавление осуществляется так:

```
elif choice == "a":
    word = input("Введите слово: ")
    value = input("Введите перевод: ")
    dict[word] = value
    print("Слово добавлено!")
```

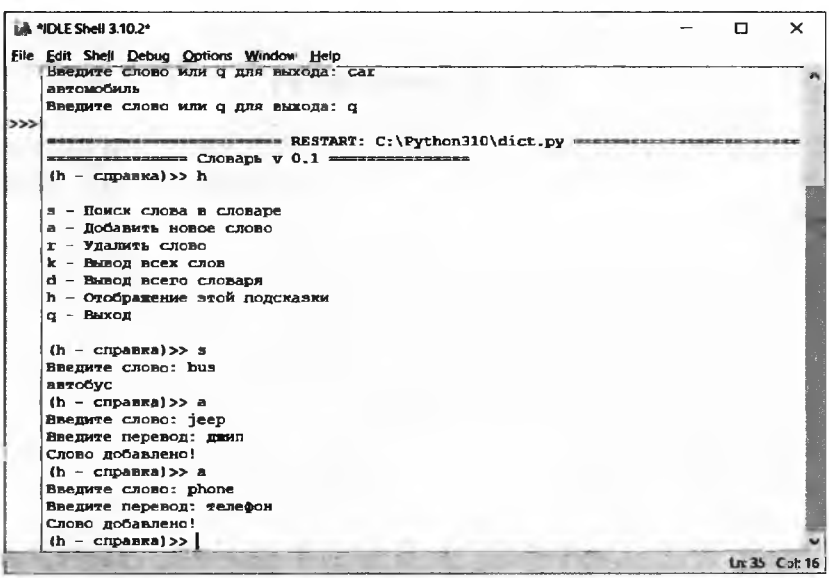


Рис. 18.3. Поиск в словаре и добавление новой пары

Вывод словаря осуществляется в удобном для человека формате:

```
elif choice == "d":
    for word in dict:
        print(word, ": ", dict[word])
```

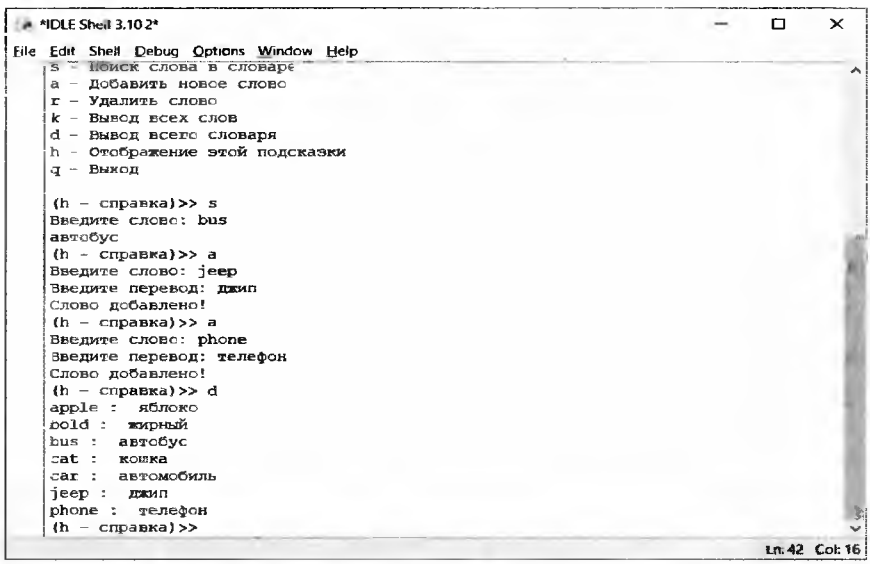


Рис. 18.4. Вывод словаря

А вот вывод всех слов подойдет разве что для отладки. При желании вы можете модифицировать код, чтобы он выводил список слов в удобном для человека формате:

```
elif choice == "k":
    print(dict.keys())
```

Вывод будет таким:

```
(h - справка)>> k
dict_keys(['bus', 'apple', 'cat', 'bold', 'phone', 'car'])
```

При вводе неизвестной команды программа выводит соответствующее сообщение, а при вводе *q* происходит выход из программы:

```
elif choice == "q":
    continue;
else:
    print("Нераспознанная команда. Введите h для справки")
```

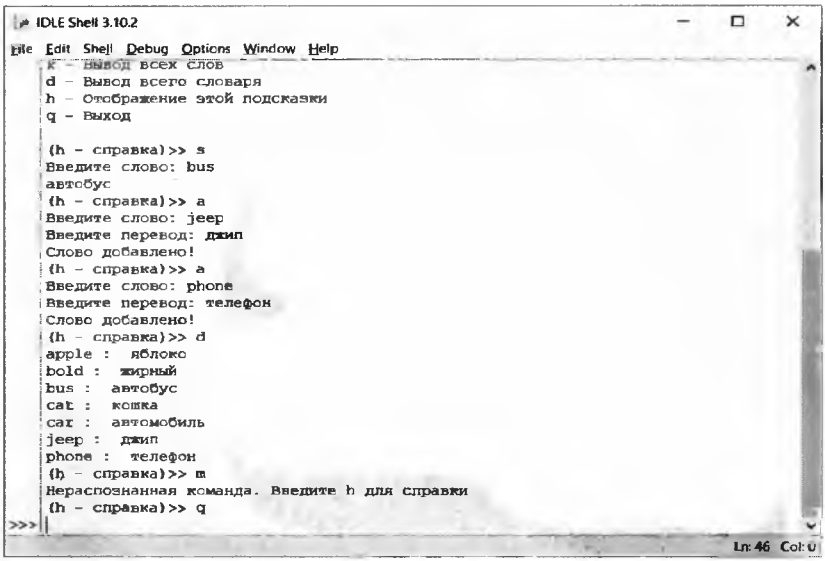
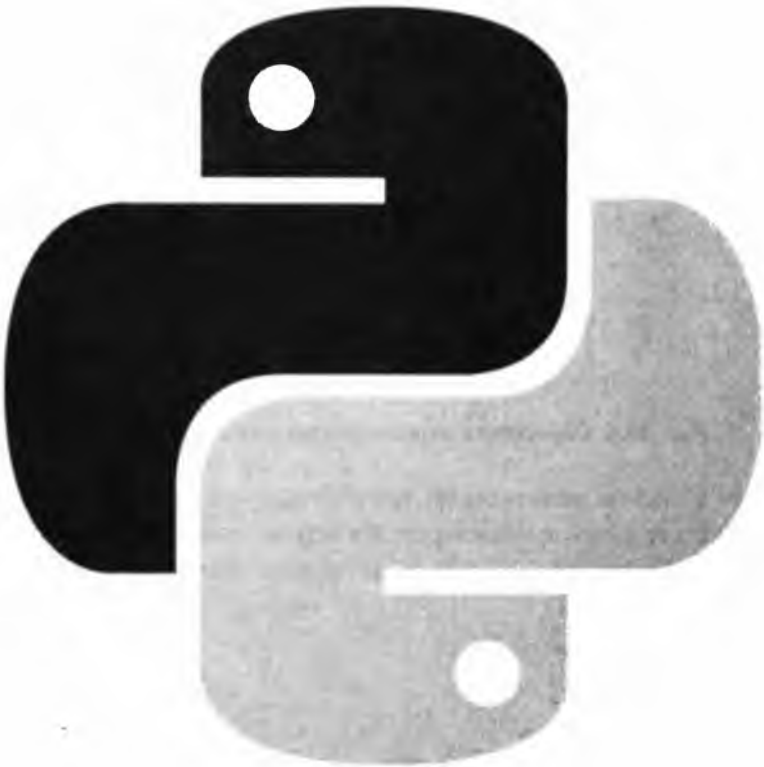


Рис. 18.5. Обработка нераспознанной команды и выход

При вводе "q" мы вызываем *break*, чем инициируем переход на следующую итерацию. Далее в цикле *while* будет проверено значение и произведен выход из цикла. В принципе, можно было бы использовать *break*, чтобы сразу прервать работу цикла.



ЧАСТЬ VI.

ФУНКЦИИ

Ни одна серьезная программа не обходится без функций. В любом языке программирования имеются подпрограммы, которые служат для экономии кода программы и удобства программиста. В Python подпрограммы называются функциями. Функции, как в любом другом языке программирования, могут возвращать значения. Заметьте: могут возвращать, а могут и нет – все зависит от того, какую функцию вы разрабатываете. Вы можете написать функцию, добавляющую переданные ей аргументы в текстовый файл или в базу данных, но не выводящую никакого текста и не возвращающую никаких значений.

Функциям в Python можно передавать параметры (аргументы). Примечательно, что при этом можно создавать функции с произвольным числом параметров и функции с параметрами по умолчанию, что облегчает использование функций.

Функция может возвращать любой тип данных с помощью оператора ***return****, но, как уже было отмечено, использование этого оператора необязательно.*

Глава 19.

ВВЕДЕНИЕ В ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ

19.1. Создание функции

Объявление функции начинается со служебного слова *def*:

```
def <имя_функции>():
```

Данная строка сообщает интерпретатору, что следующий блок кода – функция. По сути, функция – это блок выражений. Каждый раз, когда интерпретатор встретит команду имя_функции(), он выполнит данный блок выражений. Строка с оператором *def* и следующий за ней блок выражений называется *объявлением функции*.

Встретив объявление функции, интерпретатор не будет ее выполнять, он просто отметит, что такая функция есть и все. Выполнение функции начнется, когда интерпретатор встретит команду имя_функции().

Теперь рассмотрим пример объявления функции (листинг 19.1).

Листинг 19.1. Пример объявления функции

```
def help():
    print("""Доступные команды:
          -h - справка
          -a - добавить
          -d - удалить
          -p - вывести""")

print("Эта строка будет отображена раньше")

help()
```

Мы объявили функцию help(), выводящую справку по использованию программы. Обратите внимание, наша функция состоит всего из одного вызова функции print() и она ничего не обращает – оператора *return* у нее нет. Это то, о чем было сказано ранее – функция может возвращать значения, а может и не возвращать. ***Подпрограммы**, которые не возвращают никакого значения, в некоторых языках называются **процедурами**.* В Python отдельной терминологии нет. Подпрограмма, независимо от того, возвращает она значение или нет, называется функцией.

Обратите внимание: не смотря на то, что функция была объявлена раньше следующего за ней вызова функции print(), строка "Эта строка будет отображена раньше" будет показана раньше, чем подсказка, выводимая функцией help().

Это потому, что интерпретатор при объявлении функции не выполняет ее, а выполнение функции начинается, когда будет встречен оператор ее вызова. Надеюсь, это понятно.

Рис. 19.1. Подсказка по использованию программы выведена функцией help()

Прежде, чем перейти к следующему разделу, нужно поговорить об именах функций. Используйте те же правила, что и в названиях переменных. Позаботьтесь, чтобы название функции отображало ее суть. Например, функция sum() должна возвращать сумму, а не произведение.

Функцию можно объявить в любом месте программу, но до первого ее использования – как и в случае с переменными. Если вызвать функцию до ее первого использования, то вы получите примерно такое сообщение об ошибке:

```
Traceback (most recent call last):
  File "C:/Python310/30.py", line 1, in <module>
    help1()
NameError: name 'help' is not defined
```

19.2. Параметры и возвращаемые значения

Теперь усложним нашу задачу и напишем функцию, которая бы принимала параметры и возвращала какое-то значение. Напишем функцию sum(), которая принимает два параметра и возвращает сумму этих параметров:

```
def sum(a, b):
    return a+b
```

Вызвать функцию можно так:

```
print(sum(2,2))
```

В результате будет выведено:

```
4
```

Оператор *return* может возвращать, как результат вычисления выражения, так и значение переменной. При желании можно было бы объявить функцию так:

```
def sum(a, b):
    res = a + b
    return res
```

Но в простых случаях, как правило, проще использовать первый вариант – так код будет еще компактнее.

Рассмотрим еще один пример. Ранее мы писали пример кода, который должен запросить у пользователя определенный ввод и он будет запрашивать ввод до тех пор, пока пользователь не введет *y* или *n*. Сейчас мы можем сделать код основной программы компактнее благодаря использованию функции. Например:

```
def yn(message):
    resp = None
    while resp not in ("y", "n"):
        resp = input(message).lower()
    return resp
```

Тогда код основной программы станет компактнее:

```
answer = yn("Форматировать диск? [y/n] ")
print("Ваш выбор: ", answer)
```

Вывод программы:

```
Форматировать диск? [y/n] y
Ваш выбор:  y
```

19.3. Документирование функций

Очень полезно документировать функции по мере их написания. Пройдет время, и даже вы не сможете вспомнить, как работает та или иная функция, какие параметры она должна принимать. Конечно, в простом случае достаточно взглянуть на ее код и все станет понятно, но есть ситуации, когда

функции содержат сотни строк кода. В таких ситуациях на помощь приходит документирование.

В Python есть особый механизм, который называется документирующими строками. Такие строки представляют собой строку в тройных кавычках. В блоке кода документирующая строка должна обязательно идти первой по порядку. Рассмотрим пример:

```
def warning(message):
    """ Выводит сообщение, заданное параметром message,
    обрамленное символами * для привлечения внимания """
    print("*" * 10, message, "*" * 10)
```

Данная строка воспринимается как многострочный комментарий и никак не обрабатывается интерпретатором, тем более не выводится на экран и не возвращается в качестве значения функции.

При желании документировать функцию можно и с помощью комментариев. Но использование документирующих строк элегантнее и удобнее.

19.4. Повторное использование кода

Основное назначение функций – это повторное использование кода. Представим, что у нас есть какой-то код, который нужно вызывать несколько раз в основной программе – либо в неизменном виде, либо с небольшими изменениями. Тогда целесообразно данный код оформить в виде функции – без параметров или с параметрами (если код с небольшими изменениями – тогда параметры будут играть роль тех самых небольших изменений).

Это так называемое повторное использование кода. Вы можете взять, например, разработанную нами ранее функцию yn() и использовать ее в любом месте программы – где вам это нужно, меняя лишь сообщение:

```
answer = yn("Форматировать жесткий диск? [y/n] ")
answer = yn("Отправить письмо теще? [y/n] ")
answer = yn("Заспамить сервер конкурента? [y/n] ")
```

Преимущества повторного использования кода следующие:

- Оно позволяет повысить производительность труда программиста. На проект, который использует уже существующий код, будет затрачено меньше времени, меньше усилий.
- Оно позволяет улучшить качество программ. Вы можете взять и тщательно протестировать какой-то фрагмент кода и оформить его в виде функции. После этого в других проектах вы можете использовать эту функцию, а не изобретать колесо заново. При этом вы можете быть уверенным, что этот код работает безупречно, и при этом еще сэкономите время на тестирование.
- Оно позволяет повысить эффективность ПО. Здесь все просто: если есть хороший код, который эффективно работает, то целесообразно использовать его, а не писать новый код.

Как повторно использовать код? В нашем случае, пока вы еще не знакомы с модулями, нужно просто взять и скопировать функцию из одной программы и вставить в другую программу. А в других частях нашей книги вы познакомитесь с модулями и сможете создать свой модуль, содержащий все написанные ранее вами функции. После создания модуля его можно будет подключать к любой Python-программе, и вызывать уже существующие в нем функции вместо написания новых для каждого проекта.

19.5. Улучшение в проверке типов в версии 3.10

Если вы используете проверку типов, то должны знать, что Python 3.10 включает много улучшений в проверке типов, среди них оператор объединения типов, синтаксис которого теперь чище.

```
# Функция, которая принимает `int` или `float`
# Раньше:
def func(value: Union[int, float]) -> Union[int, float]:
    return value

# Теперь:
def func(value: int | float) -> int | float:
    return value
```

Кроме того, это простое улучшение не ограничивается только аннотациями типа, оно может применяться с функциями isinstance() и issubclass():

```
isinstance("hello", int | str)
# True
```

19.6. Практический примеры

19.6.1. Программа для чтения RSS-ленты

Формат новостной ленты RSS довольно популярен во всем мире, особенно на новостных сайтах и всевозможных форумах. Многие пользователи предпочитают читать RSS-ленту, а не заходить на сайт. Почему? Да потому что в RSS-ленту не попадает реклама и прочий не нужный пользователю контент – он видит только текст новости и некоторые другие служебные данные (дату и время публикации, имя автора и т.д.)

Сейчас мы попробуем написать программу, которая будет читать RSS новостную ленту (лист. 19.2).

Листинг 19.2. Программа для чтения RSS

```
def rss_reader(url):
    from urllib.request import urlopen
    from xml.etree.ElementTree import parse

    # Загружаем RSS-ленту и парсим ее
    u = urlopen(url)
    doc = parse(u)
    # Извлекаем и выводим интересующие теги
    for item in doc.iterfind('channel/item'):
        title = item.findtext('title')
        date = item.findtext('pubDate')
        link = item.findtext('link')

        print(title)
        print(date)
        print(link)
        print()

rss_reader('http://www.dkws.org.ua/phpbb2/rss.php')
```

Итак, у нас есть функция rss_reader(), которой нужно передать адрес новостной ленты. Посмотрим, что происходит внутри функции. Первым делом импортируются модули **urlopen** и **parse**. Первый нужен для открытия удаленного документа, второй – для разбора (парсинга XML-формата, в котором и распространяется новостная лента).

Далее в переменной *u* мы читаем новостную ленту, адрес которой задается параметром *url*. После мы выполняем парсинг этой ленты, и результат сохраняем в *doc*.

Переменная *doc* содержит уже "разобранную" новостную ленту. Функция xml.etree.ElementTree.parse() парсит весь XML-документ в объект документа. Вы можете использовать методы вроде find(), iterfind() и findtext() для поиска определенных XML-документов. Аргументы к этим функциям – имена определенных тегов, вроде *channel/item* или *title*.

При определении тегов вы должны принять полную структуру документа во внимание. Каждая операция *find* работает относительно начального элемента. Аналогично, имя тега, которое вы передаете каждой операции, тоже указывается относительно начального элемента. В примере вызов к doc.iterfind('channel/item') находит все элементы "item", которые находятся внутри элемента "channel". "doc" представляет верхнюю часть документа (элемент "rss"). Более поздние вызовы item.findtext() будут иметь место относительно найденных элементов "item".

Далее в цикле мы просто находим и выводим элементы title, pubDate, link. Как правило, элементы с такими названиями есть в большинстве случаев, но вы можете просмотреть код RSS-ленты и изменить названия элементов, если в коде используются другие.

Вывод программы изображен на рис. 19.2.

Рис. 19.2. Вывод программы

19.6.2. Обратный отсчет

Напишем функцию для обратного отсчета. В качестве параметра она принимает количество секунд, затем выводит оставшееся время в формате *минуты:секунды* на консоль.

Листинг 19.3. Обратный отсчет

```
import time

def countdown(time_sec):
    while time_sec:
        mins, secs = divmod(time_sec, 60)
        timeformat = '{:02d}:{:02d}'.format(mins, secs)
        print(timeformat, end='\r')
        time.sleep(1)
        time_sec -= 1

    print("stop")

countdown(10)
```

По окончанию отсчета будет выведено слово "stop". Работает наша функция так:

- Метод divmod() принимает два числа и возвращает пару чисел (кортеж), состоящую из их частного и остатка.
- end='\r' перезаписывает вывод для каждой итерации.
- Значение *time_sec* уменьшается в конце каждой итерации.

Глава 20.

ПОДРОБНО ОБ АРГУМЕНТАХ. АНОНИМНЫЕ ФУНКЦИИ

Ранее был рассмотрен самый примитивный способ передачи параметров – для упрощения материала. Python позволяет использовать именованные аргументы и задавать значения по умолчанию для аргументов. Также в Python возможно использование анонимных функций – небольших функций, которые применяются тогда, когда не хочется или не нужно создавать полноценную функцию.

20.1. Именованные аргументы. Программа "Привет"

Давайте рассмотрим функцию *hello*, использующую самый простой способ передачи параметров:

```
def hello(name, city):
  print("Привет, ", name, "! Сегодня мы едем в ", city)
```

Ничего сложного. Мы просто определили два параметра – *name* и *city*. Первый – имя, второй город. Такие параметры называются позиционными, поскольку строго определен порядок их следования. Значения функция принимает в том же порядке, что и указаны позиционные параметры.

Рассмотрим вызов функции:

```
print("Вадик", "Краснодар")
```

Вывод будет таким:

```
Привет, Вадик! Сегодня мы едем в Краснодар
```

А что будет, если программист перепутает порядок следования аргументов:

```
print("Краснодар", "Вадик")
```

Тогда параметру *name* будет передано значение "Краснодар", а параметру *city* – "Вадик". В итоге вывод будет не таким, как мы ожидали:

```
Привет, Краснодар! Сегодня мы едем в Вадик
```

Это у нас еще простой случай, а представьте, если бы второй параметр был числом, и далее шла его обработка как числа! Тогда мы бы получили ошибку, и выполнение программы было бы остановлено!

Специально для таких случаев предназначены *именованные аргументы*. Они позволяют указывать аргументы в любом порядке при условии, что мы задаем имя аргумента. Вызовем функцию так:

```
hello(city = "Краснодар", name = "Вадик")
```

Вот теперь вывод будет таким, как мы ожидали:

```
Привет, Вадик! Сегодня мы едем в Краснодар
```

Преимущества использования именованных аргументов следующие:

- **Ясность** – вы знаете, какое значение и какому параметру передаете. Даже если вы будете указывать параметры в том же порядке, в котором они и объявлены, использование именованных аргументов добавляет прозрачности в вашу программу.
- **Возможность изменения порядка следования параметров** – если вы намерено изменили порядок следования аргументов (потому что вам так захотелось) или случайно перепутали его, ничего страшного не произойдет, и функция будет работать корректно (в отличие от использования позиционных параметров).

20.2. Значения параметров по умолчанию

В Python вы можете присваивать значения параметрам по умолчанию. Данные значения будут переданы функции, если вызов функции не содержит соответствующих аргументов. Например:

```
def hello2(name = "Гость", city = "Псков"):
  print("Привет, ", name, "! Сегодня мы едем в ", city)
```

Если вызвать данную функцию без параметров, то есть так:

```
hello2()
```

тогда, во-первых, не будет никаких ошибок, а во-вторых, будет выведена строка:

```
Привет, Гость! Сегодня мы едем в Псков
```

Параметру *name* будет присвоено значение "Гость", а параметру *city* – "Псков". Вы также можете указать какой-то один из параметров, например,

```
hello2(name = "Оля")
```

Тогда Оля поедет в Псков – город по умолчанию.

Зачем нужны значения по умолчанию? Наши примеры – тривиальны. Но есть случаи, когда функция может принимать десятки параметров, но большую часть из них можно установить по умолчанию и значения по умолчанию будут приемлемы в большинстве случаев. Именно в таких случаях на помощь придут значения параметров по умолчанию – они избавят программиста от набора лишнего кода.

20.3. Переменное число параметров

Представим, что вам нужно написать функцию, принимающую любое число параметров. Для этого используйте аргумент *. Пример:

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))
```

Рассмотрим пример использования функции:

```
print(avg(1, 2))     # 1.5
print(avg(1, 2, 3, 4))    # 2.5
```

В данном случае *rest* – это кортеж, содержащий все дополнительно переданные аргументы. Наш код считает его последовательностью и работает как с последовательностью.

Чтобы принять любое число *именных параметров* (*keyword arguments*), используйте параметр, который начинается с **. Например:

```
import html

def make_element(name, value, **attrs):
keyvals = [' %s="%s"' % item for item in attrs.items()]
attr_str = ''.join(keyvals)
element = '<{name}{attrs}>{value}</{name}>'.format(
name=name,
attrs=attr_str,
value=html.escape(value))
return element
```

Примеры использования:

```
# Создаем '<item size="large" quantity="6">Вентилятор</item>'
make_element('item', 'Вентилятор', size='large', quantity=6)

# Создаем '<p>&lt;слово&gt;</p>'
make_element('p', '<слово>')
```

Здесь **attrs** – словарь, который хранит переданные именные аргументы (если они были переданы, конечно).

Если вы хотите написать функцию, которая сможет принимать любое число позиционных и именных параметров, используйте * и ** вместе. Например:

```
def anyargs(*args, **kwargs):
  print(args)  # Кортеж
  print(kwargs)   # Словарь
```

С этой функцией все позиционные аргументы будут помещены в кортеж **args**, а все именные аргументы будут помещены в словарь **kwargs**.

Параметр * может быть указан исключительно как последний позиционный параметр в определении функции. Параметр ** тоже может появиться как последний параметр. Тонкий аспект определения функции – то, что параметры могут все еще появиться после параметра *:

```
def a(x, *args, y):
      pass
def b(x, *args, y, **kwargs):
      pass
```

20.4. Анонимные и встроенные функции

Некоторые функции, например, функции сортировки, подразумевают передачу в качестве параметров пользовательских функций, определяющих порядок сортировки или что-либо еще. В таких случаях удобнее использовать короткие встроенные функции, а не создавать полноценные функции оператором *def*.

Простые функции, которые делают ни что иное, как просто вычисляют выражение, могут быть заменены выражением *lambda*. Например:

```
>>> add = lambda x, y: x + y
>>> add(2,2)
4
>>> add('привет', 'мир')
'приветмир'
>>>
```

Использование *lambda* здесь аналогично следующим кодом:

```
>>> def add(x, y):
...    return x + y
...
>>> add(2,2)
4
>>>
```

Как правило, *lambda* используется в контексте некоторой другой операции, такой как сортировка или сокращение данных:

```
>>> names = ['Иван Иванов', 'Петя Петров',
...      'Вася Смирнов', 'Коля Пупкин']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Иван Иванов', 'Петя Петров', 'Коля Пупкин', 'Вася Смирнов']
>>>
```

Хотя *lambda* позволяет вам определять простую функцию, определять такую функцию не рекомендуется. В частности, может быть определено только единственное выражение, результатом которого является возвращаемое значение. Это означает, что никакие другие функции языка, включая многократные операторы, условные выражения, итерация и обработка исключений не могут быть включены в эту функцию.

Вы можете написать много Python-кода, вообще не используя *lambda*. Однако вы будете иногда встречаться с ней в программах, где кто-то пишет много крошечных функций, которые вычисляют различные выражения или в программах, которые требуют, чтобы пользователи предоставили callback-функции.

Рассмотрим поведение следующего кода:

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

Теперь задайте себе вопрос. Каковы значения a(10) и b(10)? Если вы думаете, что 20 и 30, то вы ошибаетесь.

```
>>> a(10)
30
>>> b(10)
30
>>>
```

Проблема здесь в том, что значение *x* в выражении *lambda* является свободной переменной, которая связывается во время выполнения, а не во время определения. Поэтому значение *x* в lambda-выражении – то же, что и значение переменной *x* во время выполнения. Например:

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

Если вы хотите написать анонимную функцию, которая получает значение на момент определения и хранить ее, добавьте значение как значение по умолчанию, например:

```
>>> x = 10
>>> a = lambda y, x=x: x + y.
>>> x = 20
>>> b = lambda y, x=x: x + y
```

```
>>> a(10)
20
>>> b(10)
30
>>>
```

Теперь поговорим о переносе дополнительного состояния в функциях обратного вызова. Представим, что вы написали код, который основывается на использовании callback-функций (например, обработчики событий), но вы хотите, чтобы callback-функция хранила дополнительную информацию о состоянии для использования внутри функции.

Этот пример связан с использованием функций обратного вызова, которые используются во многих библиотеках и структурах – особенно связанных с асинхронной обработкой. Для иллюстрации и из соображений тестирования определим следующую функцию, которая вызывает функцию обратного вызова:

```
def apply_async(func, args, *, callback):
  # Вычисляем результат
  result = func(*args)
  # Вызываем callback-функцию и передаем ей результат
  callback(result)
```

На практике такой код мог бы выполнять сортировку с использованием потоков, процессов и таймеров, но здесь мы не об этом. Вместо этого мы просто фокусируемся на вызове callback-функции. Вот пример, показывающий, как можно использовать предыдущий код:

```
>>> def print_result(result):
...   print('Результат:', result)
...
>>> def add(x, y):
...   return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Результат: helloworld
>>>
```

Как видите, функция print_result() принимает только один аргумент, который является результатом. Никакая другая информация не передается в нее. Такой недостаток информации иногда может представлять проблему, например, когда вы хотите, чтобы функции обратного вызова взаимодействовала с другими переменными или частями среды.

Один из способов хранить дополнительную информацию в функции обратного вызова – использовать метод (и, соответственно, класс) вместо функции. Например, следующий класс хранит внутренний номер последовательности, который вызывается при каждом вызове метода handler():

```
class ResultHandler:
  def __init__(self):
    self.sequence = 0
  def handler(self, result):
    self.sequence += 1
    print('[{}] Результат: {}'.format(self.sequence, result))
```

Чтобы использовать этот класс, вам нужно создать экземпляр и использовать связанный метод *handler* в качестве функции обратного вызова:

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Результат: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Результат: helloworld
>>>
```

20.5. Возвращаем несколько значений

Иногда нужно, чтобы функция вернула несколько значений. Для этого просто возвращайте кортеж, например:

```
def fun():
    return 1, 2, 3

a, b, c = fun()

print(a, b, c)
```

В результате будет выведено:

```
1 2 3
```

Хотя кажется, что функция fun() возвращает несколько значений, на самом деле она возвращает одно значение, но в виде кортежа. Это выглядит немного странным, но кортеж формирует запятая, а не круглые скобки. Пример:

```
>>> a = (1, 2)    # Со скобками
>>> a
(1, 2)
>>> b = 1, 2   # Без скобок
>>> b
(1, 2)
>>>
```

При вызове функций, которые возвращают кортеж, принято присваивать результат несколько переменным, как было показано выше. Это просто – распаковка кортежа. Возвращаемое значение можно также присвоить одной переменной.

```
>>> x = fun()
>>> x
(1, 2, 3)
>>>
```

20.6. Изменения синтаксиса псевдонима типа

В более ранних версиях Python добавлены псевдонимы типов, позволяющие создавать синонимы пользовательских классов. В Python 3.9 и более ранних версиях псевдонимы записывались так:

```
FileName = str

def parse(file: FileName) -> None:
    ...
```

Здесь FileName – псевдоним базового типа строки Python. Начиная с Python 3.10, синтаксис определения псевдонимов типов будет изменён:

```
FileName: TypeAlias = str

def parse(file: FileName) -> None:
    ...
```

Благодаря этому простому изменению и программистам, и инструментам проверки проще отличить присваивание переменной от псевдонима. Новый синтаксис обратно совместим, так что вам не нужно обновлять старый код с псевдонимами.

Глава 21.

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

21.1. Инкапсуляция

А зачем функции возвращаю значения? Посмотрим на такую функцию:

```
def fun():
  res = 10
  return res
```

Почему бы нам не обратиться к переменной *res* напрямую – в коде нашей программы? Спешу вас огорчить: потому что нельзя. Переменная *res* не существует вне функции. Вообще ни одна переменная, созданная внутри функции (в том числе и параметры), извне не доступна. Данная техника называется *инкапсуляцией*. Она помогает сохранить независимость отдельных фрагментов кода. Параметры и возвращаемые значения используются, чтобы передавать важную информацию и игнорировать все прочее. За значениями переменных, созданных внутри функции, не нужно следить во всем остальном коде, что очень удобно. И чем больше программа, тем более заметно это преимущество.

Поскольку код функции скрыт от основной программы и от других функций, в разных функциях вы можете использовать одни и те же имена переменных, например, *res* для обозначения результата.

21.2. Область видимости. Ключевое слово *global*

Благодаря инкапсуляции, функции как бы закрыты от основной программы и от других функций. Пока мы знаем только единственный механизм обмена информацией между ними – это параметры и возвращаемые значения. Однако есть еще и другой способ – глобальные переменные.

Область видимости – это способ представления разных частей программы, отделенных друг от друга.

Рассмотрим небольшой пример:

```
def fun1():
    res = 10
    print(res)

def fun2()
    res = 20
    print(res)

res = 30
print(res)
fun1()
fun2()
print(res)
```

Вывод программы будет таким:

```
30
10
20
30
```

Сначала мы определили две функции, внутри каждой из них переменной *res* присваивается разное значение – 10 и 20 соответственно. Далее в основной программе мы определили переменную *res* со значением 30.

Сначала мы выводим значение переменной *res* до запуска функций. Затем запускаем обе функции и снова выводим значение переменной *res*, чтобы убедиться, что ни одна из функций его не изменила.

Как видите, программа и две наших функции выводят собственные значения переменной *res*, а все потому, что у нас есть целых три области видимости – одна глобальная (программа) и две локальные – по одной для каждой из функций.

Любая переменная, созданная в глобальной области видимости, называется *глобальной*. Переменная, объявленная в локальной области, называется *локальной*.

Если вам нужно получить доступ к глобальной переменной, тогда нужно использовать ключевое слово *global*. Изменим нашу программу так:

```
def fun1():
    global res
    print(res)

def fun2():
    res = 20
    print(res)

res = 30
print(res)
fun1()
fun2()
print(res)
```

Теперь вывод программы будет такой:

```
30
30
20
30
```

Посмотрим, что произошло. Сначала мы вывели значение *res*, определенное в основной программе. Затем мы вызвали функцию fun1(), которая благодаря ключевому слову *global* получила доступ к глобальной переменной *res* и вывела ее значение. Функция fun2() вывела собственное значение *res*. Далее мы отобразили значение переменной *res* из глобальной области.

Использование ключевого слова *global* позволяет не только читать, но и записывать, то есть изменять значение глобальной переменной. Рассмотрим следующий пример:

```
def fun1():
    global res
    res = 50
    print(res)

def fun2():
    global res
    print(res)

res = 30
print(res)
fun1()
fun2()
print(res)
```

Изначально значение глобальной переменной *res* было 30. Затем в функции fun1() мы изменили его на 50. Функция fun2(), поскольку она вызывается после функции fun1(), получает уже новое значение – 50. Поскольку функция fun1() изменила значение переменной res() в глобальной области, то последний оператор print() отобразит также 50. В итоге вывод программы будет таким:

```
30
50
50
50
```

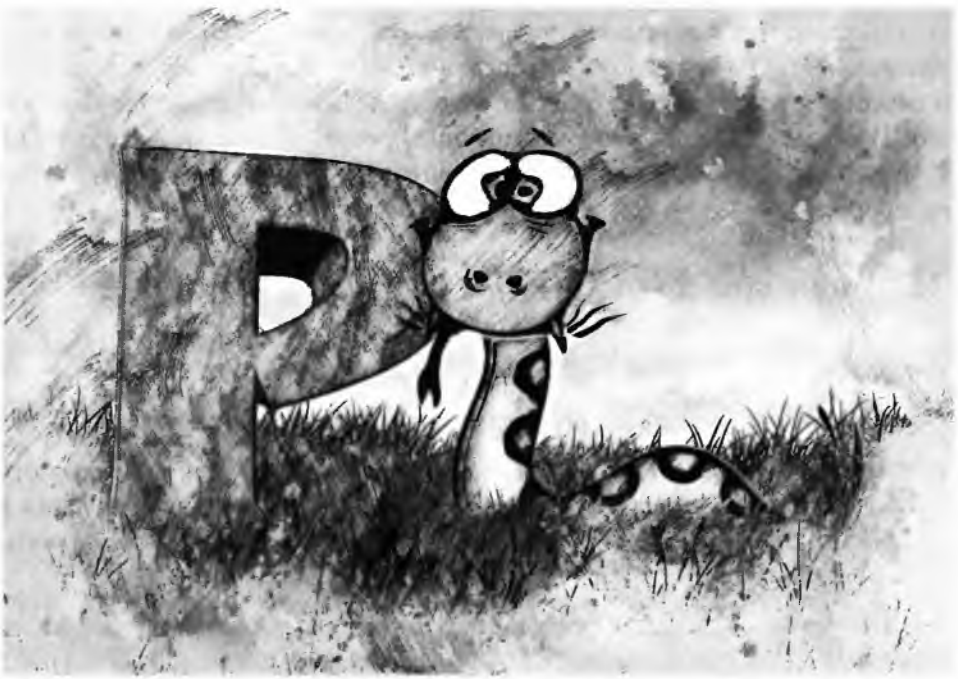
21.3. Стоит ли использовать глобальные переменные?

Да, в Python вы можете использовать глобальные переменные. А нужно ли? Ведь, по сути, тогда вы лишаетесь преимуществ инкапсуляции – вам нужно будет следить за значением переменной. Одна логическая ошибка в большой программе приведет к непоправимым последствиям и многочасовой

отладке. Нужно ли вам это? Глобальные переменные только запутывают код, поскольку за их постоянно меняющимися значениями сложно следить. Поэтому постарайтесь ограничить их использование по максимуму. Основной девиз должен быть таким: если можно обойтись без глобальной переменной, сделайте это.

Глава 22.

ИГРА «КРЕСТИКИ-НОЛИКИ» ДЛЯ ДВУХ ИГРОКОВ



В этой главе будет рассмотрен практический пример использования подпрограмм. Мы напишем игру "Крестики-нолики" для двух игроков, то есть в качестве второго игрока будет выступать не компьютер, а второй человек. Программа демонстрирует написание простой консольной игрушки на Python, а также позволяет сэкономить множество бумаги и сохранить окружающую среду.

22.1. Список функций

В нашей программе будут определены следующие функции:

- *draw_board()* – рисует игровое поле в привычном для человека формате.
- *take_input()* – принимает ввод пользователя. Проверяет корректность ввода. Здесь вы увидите пример обработки исключительной ситуации в случае ошибочного ввода. Подробно об обработке исключений мы поговорим в других главах.
- *check_win()* – функция проверки игрового поля, проверяет, выиграл ли игрок.
- *main()* – основная функция игры, которая будет запускать все ранее описанные функции. По сути, данная функция запускает и управляет игровым процессом.

22.2. Функция draw_board()

Это самая простая функция нашей программы. Внутри программы игровое поле будет представлено в виде одномерного списка с числами от 1 до 9. Для создания списка можно воспользоваться функцией rand():

```
board = list(range(1,10))

def draw_board(board):
    print ("-" * 13)
    for i in range(3):
        print ("|", board[0+i*3], "|", board[1+i*3], "|",
board[2+i*3], "|")
        print ("-" * 13)
```

Результат выполнения функции draw_board() приведен на рис. 22.1.

Рис. 22.1. Игровое поле

22.3. Функция take_input()

Задачи функции take_input() следующие:

1. Принять ввод пользователя
2. Обработать некорректный ввод, например, когда введено не число. Для преобразования строки в число мы будем использовать уже знакомую нам функцию int()
3. Обработать ситуации, когда клетка занята или когда введено число не из диапазона 1..9.

```
def take_input(player_token):
    valid = False
    while not valid:
        player_answer = input("Куда поставим " + player_token+"? ")
        try:
            player_answer = int(player_answer)
        except:
            print ("Некорректный ввод. Вы уверены, что ввели число?")
            continue
        if player_answer >= 1 and player_answer <= 9:
            if (str(board[player_answer-1]) not in "XO"):
                board[player_answer-1] = player_token
                valid = True
            else:
                print ("Эта клеточка уже занята")
        else:
            print ("Некорректный ввод. Введите число от 1 до 9.")
```

Обратите внимание, как мы обрабатываем некорректный ввод, когда введена строка, а не число. Мы заключаем int(input()) в блок try..except. Если пользователь введет строку, то выполнение программы не прервется, а будет выведено сообщение "*Некорректный ввод. Вы уверены, что ввели число?*", а затем цикл перейдет на следующую итерацию, следовательно, опять будет возможность ввести число.

Если же введено число, то нам нужно обработать, не заполнена ли ячейка крестиком или ноликом. Если она заполнена, тогда мы выводим сообщение "Эта клеточка уже занята". В противном случае – заполняем клеточку токеном (иксом или ноликом) пользователя.

22.4. Функция check_win()

Данная функция проверяет игровое поле. Проверка результатов игры "Крестики-нолики" достаточно распространенная задача по программированию. Часто бывает, одно и то же задание в программировании можно ре-

шить несколькими способами. В данном случае мы просто создали кортеж с выигрышными координатами и прошлись циклом *for* по нему. Если символы во всех трех заданных клетках равны – возвращаем выигрышный символ, иначе – возвращаем значение *False*. При этом важно помнить, что непустая строка (наш выигрышный символ) при приведении ее к логическому типу вернет *True* (это понадобится нам в дальнейшем).

```
def check_win(board):
    win_coord =
((0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6))
    for each in win_coord:
        if board[each[0]] == board[each[1]] == board[each[2]]:
            return board[each[0]]
    return False
```

22.5. Функция main()

Функция main() предельно проста. В принципе, можно было бы обойтись и без нее, а описать весь код в основной программе, но раз мы уже в этой части книги изучаем функции, то было принято решение оформить код так.

В цикле *while*, который выполняется пока один из игроков не выиграл, мы выводим игровое поле, принимаем ввод пользователя, при этом определяя токен (икс или нолик) игрока.

Мы ждем, когда переменная *counter* станет больше 4 для того, чтобы избежать заведомо ненужного вызова функции *check_win* (до пятого хода никто точно не может выиграть). Переменная *tmp* была создана опять же для того, чтобы лишний раз не вызывать функцию *check_win*, мы просто "запоминаем" ее значение и при необходимости используем в строке 47 (print (tmp, "выиграл!")). Польза от такого подхода не так заметна при работе с небольшими объемами данных, но в целом подобная экономия процессорного времени – хорошая практика.

```
def main(board):
    counter = 0
    win = False
    while not win:
        draw_board(board)
        if counter % 2 == 0:
            take_input("X")
```

```
        else:
            take_input("O")
        counter += 1
        if counter > 4:
            tmp = check_win(board)
            if tmp:
                print (tmp, "выиграл!")
                win = True
                break
        if counter == 9:
            print ("Ничья!")
            break
    draw_board(board)
```

Полный исходный код программы приведен в листинге 22.1.

Листинг 22.1. Полный исходный код программы "Крестики-нолики"

```
board = list(range(1,10))

def draw_board(board):
    print ("-" * 13)
    for i in range(3):
        print ("|", board[0+i*3], "|", board[1+i*3], "|", board[2+i*3], "|")
        print ("-" * 13)

def take_input(player_token):
    valid = False
    while not valid:
        player_answer = input("Куда поставим " + player_token+"? ")
        try:
            player_answer = int(player_answer)
        except:
            print ("Некорректный ввод. Вы уверены, что ввели число?")
            continue
        if player_answer >= 1 and player_answer <= 9:
            if (str(board[player_answer-1]) not in "XO"):
                board[player_answer-1] = player_token
                valid = True
            else:
                print ("Эта клеточка уже занята")
        else:
            print ("Некорректный ввод. Введите число от 1 до 9.")

def check_win(board):
    win_coord =
```

```
((0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6))
    for each in win_coord:
        if board[each[0]] == board[each[1]] == board[each[2]]:
            return board[each[0]]
    return False

def main(board):
    counter = 0
    win = False
    while not win:
        draw_board(board)
        if counter % 2 == 0:
            take_input("X")
        else:
            take_input("O")
        counter += 1
        if counter > 4:
            tmp = check_win(board)
            if tmp:
                print (tmp, "выиграл!")
                win = True
                break
        if counter == 9:
            print ("Ничья!")
            break
    draw_board(board)

main(board)
```

Теперь посмотрим на программу в действии. На рис. 22.2 показано, что прошло уже три хода и по диагонали выстроились X, O, X.

Рис. 22.2. Процесс игры, рисунок 1

На рис. 22.3 показана обработка некорректного ввода – когда пытаемся поставить токен в занятую ячейку, и когда вместо числа введена строка. Также показано, что O выиграл:)

Рис. 22.3. Процесс игры, рисунок 2

Конечно, это довольно простая игра. Но в других главах книги мы разработаем полноценную графическую игру – Змейку.

ЧАСТЬ VII.

РАБОТА С ФАЙЛАМИ

До этого момента мы хранили обрабатываемые данные только в переменных. Переменные – это хорошо, но они уничтожаются при завершении работы программы, а данные часто нужно хранить постоянно – даже если программа завершена или компьютер перезагружен. В этой части книги будет рассмотрена работа с файлами. Вы узнаете, как:

- *Читать и записывать данные в текстовые файлы*
- *Работать с файлами различных форматов – XML, CSV*
- *Перехватывать и обрабатывать ошибки программы во время ее выполнения*

Глава 23.

ЧТЕНИЕ И ЗАПИСЬ ТЕКСТОВОГО ФАЙЛА

23.1. Чтение информации из текстового файла

23.1.1 Демонстрация разных способов чтения из файла

Самый простой вариант, с которым может столкнуться программист при работе с файлами – это чтение информации из текстового файла. Сейчас мы научимся читать данные из обычного текстового файла. Создайте текстовый файл text.txt следующего содержимого:

```
Первая строка
А это вторая строка
Строка номер 3
Четвертая строка
```

Кодировка файла должна быть UTF-8 (например, вы можете воспользоваться текстовым редактором Notepad2 или Atom для редактирования файлов с заданием кодировки). Данный файл нужно создать в том же каталоге, что и Python-файл, который будет производить его чтение.

Теперь попробуем прочитать созданный файл. В листинге 23.1 представлена программа, демонстрирующая различные способы чтения информации из файла.

Листинг 23.1. Разные способы чтения из текстового фаила

```
print("** Открытие и закрытие файла")
txt = open("text.txt", "r", encoding='utf-8')
txt.close()

print("** Посимвольное чтение файла")
txt = open("text.txt", "r")
print(txt.read(1))
print(txt.read(2))
print(txt.read(6))
txt.close()

print("** Посимвольное чтение с указанием кодировки")
txt = open("text.txt", "r", encoding='utf-8')
print(txt.read(1))
print(txt.read(2))
print(txt.read(6))
txt.close()

print("** Чтение всего файла")
txt = open("text.txt", "r", encoding='utf-8')
content = txt.read()
print(content)
txt.close()

print("** Читаем строку из файла посимвольно")
txt = open("text.txt", "r", encoding='utf-8')
print(txt.readline(1))
print(txt.readline(5))
txt.close()

print("** Читаем строку из файла полностью")
txt = open("text.txt", "r", encoding='utf-8')
print(txt.readline())       # Строка 1
print(txt.readline())       # Строка 2
txt.close()

print("** Чтение всего файла в список")
txt = open("text.txt", "r", encoding='utf-8')
lines = txt.readlines()

print(lines)
print(len(lines))
for line in lines:
    print(line)
```

```
txt.close()

print("** Построчное чтение файла:")
txt = open("text.txt", "r", encoding='utf-8')
for line in txt:
      print(line)
txt.close()
```

Данная программа демонстрирует всевозможные доступы к информации из текстового файла. Далее мы рассмотрим эту программу подробнее.

23.1.2. Открытие и закрытие файла

Для открытия файла используется функция open():

```
txt = open("text.txt", "r", encoding='utf-8')
```

Функции open() нужно передать три параметра:

- Имя файла
- Режим доступа (см. табл. 23.1)
- Кодировку

Начнем с первого параметра. С ним как раз все просто. Если не указан полный путь к файлу, то интерпретатор будет искать его в том же каталоге, что и программа, которая пытается открыть его.

Режим доступа определяет, какие операции можно будет выполнять с файлом. В листинге 23.1 мы использовали постоянно режим "r" – только чтение. Остальные режимы описаны в таблице 23.1.

Таблица 23.1. Режимы доступа к файлам

Режим	Описание
"r"	Чтение информации из текстового файла, если файл не существует, вы получите сообщение об ошибке

"w"	Запись в текстовый файл. Если он существует, он будет перезаписан. Если не существует, то будет создан
"a"	Дозапись в текстовый файл. Если он существует, новые данные будут записаны в его конец, если не существует, то будет создан
"r+"	Чтение и запись в текстовый файл. Если файл не существует, вы получите сообщение об ошибке
"w+"	Запись и чтение из текстового файла. Если не существует, то будет создан, если существует, то будет перезаписан
"a+"	Дозапись и чтение текстового файла. Если существует, данные будут дописаны в конец, если не существует, то будет создан

Третий параметр – это кодировка. Желательно указывать кодировку, поскольку без ее указания вместо символов русского алфавита вы можете получить абракадабру:

```
** Посимвольное чтение файла
Р
µР
µСЂРІР
** Посимвольное чтение с указанием кодировки
П
ер
вая ст
```

Чтобы закрыть файл, нужно вызвать метод close() файлового объекта:

```
txt.close()
```

23.1.3. Посимвольное чтение из файла. Чтение всего файла сразу

Посимвольное чтение редко встречается на практике. Мы рассматриваем его сугубо из академического интереса. Как правило, на практике файл читают либо полностью (когда он относительно небольшого размера), либо построчно (для экономии памяти), или же как список (для более удобной работы построчно).

Для посимвольного чтения используется метод read(), которому нужно передать количество символов, которые будут прочитаны с момента последнего чтения. Пример:

```
print(txt.read(1))
print(txt.read(2))
```

Напомню содержимое нашего файла (первая строка):

```
Первая строка
```

Первый вызов read() вернет первый символ первой строки:

```
П
```

Второй вызов read() вернет 2 символа с момента последнего чтения, то есть

```
ер
```

Интерпретатор Python помнит, где он последний раз остановился. Если дочитать файл до конца, то очередной вызов read() вернет пустую строку. Чтобы вернуться в начало файла, нужно его закрыть и снова открыть. Каких-либо методов вроде *rewind* не предусмотрено.

Чтобы прочитать весь файл сразу, не нужно указывать каких-либо параметров:

```
txt = open("text.txt", "r", encoding='utf-8')
content = txt.read()
print(content)
txt.close()
```

Результат:

```
** Чтение всего файла
```

```
Первая строка
А это вторая строка
Строка номер 3
Четвертая строка
```

Чтение всего файла сразу удобно, когда вам не нужно обрабатывать файл или обработка применяется сразу ко всему содержимому файла (например, прочитали HTML-файл и хотите вычистить его от HTML-тегов). Как правило, чтение всего файла сразу применяется, когда файл относительно небольшого размера. Большие файлы лучше читать построчно.

23.1.4. Посимвольное чтение строки

Посимвольное чтение строки – еще одна редкая операция, но она выполнима методом readline(), которому нужно передать количество символов, которые будут прочитаны из строки с момента последней операции чтения:

```
print("** Читаем строку из файла посимвольно")
txt = open("text.txt", "r", encoding='utf-8')
print(txt.readline(1))
print(txt.readline(5))
txt.close()
```

Результат:

```
** Читаем строку из файла посимвольно
П
ервая
```

Тот же метод readline() может прочитать сразу всю строку – для этого не нужно передавать ему каких-либо параметров. В этом случае файл будет прочитан построчно. При первом вызове readline() будет прочитана первая строка, при втором – вторая и т.д. Заметьте, что после вывода каждой строки в выводе программы есть пустая строка:

```
** Читаем строку из файла полностью
Первая строка

А это вторая строка
```

Это связано с тем, что функция print() добавляет символ '\n' для перевода строки и каждая строка в файле заканчивается этим же символом, поэтому у нас получается двойной символ '\n'.

23.1.5. Чтение всех строк файла в список

Для чтения всех строк файла в список используется метод readlines():

```
print("** Чтение всего файла в список")
txt = open("text.txt", "r", encoding='utf-8')
lines = txt.readlines()

print(lines)              # Выводим список
print(len(lines))         # Длина списка
# Обработка списка
for line in lines:
    print(line)
```

Переменная *lines* ссылается на список, в котором каждый элемент совпадает со строкой из нашего файла:

```
['Первая строка\n', 'А это вторая строка\n', 'Строка номер 3\n',
'Четвертая строка']
```

Дальше мы можем делать с этим списком все, что угодно, например, вывести на экран всего его элементы, что и было продемонстрировано в лист. 23.1.

23.1.6. Перебор строк файла

Можно не читать файл в список, а просто перебрать все его строки в цикле *for*:

```
txt = open("text.txt", "r", encoding='utf-8')
for line in txt:
      print(line)
```

Данный метод считается оптимальным при построчном чтении файла – он самый простой и даже не требует использования каких-либо файловых методов.

23.2. Запись в текстовый файл

23.2.1. Запись строк в файл

Для записи строк в файл используется метод write(). Но сначала файл нужно открыть в нужном режиме, например, в "w" (лист. 23.2)

Листинг 23.2. Построчная запись файла

```
print("Запись файла построчно")
txt = open("test.txt", "w", encoding='utf-8')

txt.write("Строка 1\n")
txt.write("Строка 2\n")

txt.close()

print("Чтение созданного файла")
txt = open("test.txt", "r", encoding='utf-8')
for line in txt:
    print(line)

txt.close()
```

Обратите внимание, что метод write() записывает строку, как она есть и не добавляет символ '\n', как это делает print() при выводе. Поэтому не забывайте добавлять в конце каждой строки символ '\n'.

В результате выполнения программы из листинга 23.2 вы должны увидеть вывод:

```
Запись файла построчно
Чтение созданного файла
Строка 1

Строка 2
```

23.2.2. Запись списка строк в файл

Метод writelines() позволяет записать в файл целый список строк. Пример кода:

```
print("Запись списка строк")
lines = ["Строка 1\n", "Строка 2\n", "Строка 3\n"]
txt = open("test.txt", "w", encoding='utf-8')
txt.writelines(lines)
txt.close()
```

Данный способ удобно использовать, когда у нас уже есть сформированный список строк. Вместо того, чтобы проходиться по списку и записывать строки методом write(), мы можем записать список с помощью одного вызова метода writelines().

23.2.3. Перенаправление функции print() в файл

Вывод функции print() можно перенаправить в файл. Это и будет еще одним способом записи текстовых данных в файл:

Используйте аргумент *file* для функции print(), вот так:

```
with open('somefile.txt', 'w') as f:
    print('Hello World!', file=f)
```

23.3. Чтение и запись сжатых файлов

Python позволяет работать со сжатыми файлами. Модули **gzip** и **bz2** решают нашу задачу. Оба модуля предоставляют альтернативную реализацию функции open(), которая может использоваться для нашей цели. Например, чтение сжатых файлов как текстовых может быть выполнено так:

```
# сжатие gzip
import gzip
```

```
with gzip.open('somefile.gz', 'rt') as f:
     text = f.read()
# сжатие bz2
import bz2
with bz2.open('somefile.bz2', 'rt') as f:
     text = f.read()
```

Аналогично, для записи сжатых данных можно использовать такой код:

```
# сжатие gzip
import gzip
with gzip.open('somefile.gz', 'wt') as f:
      f.write(text)
# сжатие bz2
import bz2
with bz2.open('somefile.bz2', 'wt') as f:
      f.write(text)
```

Как показано, все операции ввода/вывода используют текст и осуществляют кодирование/декодирование Unicode. Если вам нужно работать с бинарными данными, используйте режимы *rb* или *wb*.

Чтение или запись сжатых данных – довольно простые операции. Однако знайте, что критически важен выбор корректного режима открытия файла. Если вы не определяете режим, то по умолчанию режим будет бинарным, что не есть хорошо для программ, которые ожидают текст. Обе функции gzip.open() и bz2.open() принимают те же параметры, что и встроенная функция open(), в том числе *encoding*, *errors*, *newline* и т.д.

При записи сжатых данных уровень сжатия можно установить аргументом *compresslevel*. Например:

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
     f.write(text)
```

По умолчанию используется значение 9, что обеспечивает наивысший уровень сжатия. Если нужно повысить производительность, но при этом можно пожертвовать степенью сжатия, тогда можно понизить значение параметра *compresslevel*.

Наконец, малоизвестная особенность функций gzip.open() и bz2.open() – то, что они могут работать поверх существующего файла, открытого в двоичном режиме. Вот как это работает:

```
import gzip

f = open('somefile.gz', 'rb')
with gzip.open(f, 'rt') as g:
text = g.read()
```

Это позволяет модулям gzip и bz2 работать с разными файлообразными объектами, такими как сокеты, каналы и файлы в памяти.

23.4. Проверка существования и получение дополнительной информации о файле

Ранее было сказано, что если файл не существует, то вы или получите сообщение об ошибке или он будет создан – в зависимости от выбранного режима доступа. Иногда лучше проверить файл на существование, прежде чем обращаться к нему.

Для проверки существования файла можно использовать модуль os.path:

```
>>> import os
>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('/tmp/spam')
False
>>>
```

Вы можете произвести дополнительные проверки, чтобы узнать тип этого файла. Эти проверки возвращают *false*, если указанный файл не существует:

```
>>> # Это обычный файл?
>>> os.path.isfile('/etc/passwd')
True
```

```
>>> # Это каталог?
>>> os.path.isdir('/etc/passwd')
False
>>> # Это символическая ссылка?
>>> os.path.islink('/usr/local/bin/python3')
True
>>> # Получить файл, на который указывает ссылка
>>> os.path.realpath('/usr/local/bin/python3')
'/usr/local/bin/python3.4'
>>>
```

Если вам нужно получить метаданные (например, размер файла или дату изменения), то для этого также есть функции в модуле os.path:

```
>>> os.path.getsize('/etc/passwd')
4669
>>> import time
>>> time.ctime(os.path.getmtime('/etc/passwd'))
'Wed Aug 09 19:15:24 2017'
>>>
```

23.5. Контекстные менеджеры

Контекстные менеджеры отлично подходят для открытия и закрытия файлов, работы с базой данных и т.д. В Python 3.10 они станут немного удобнее. Изменение позволяет в скобках указывать несколько контекстных менеджеров, что удобно, если вы хотите создать в одном операторе *with* несколько менеджеров:

```
with (
    open("somefile.txt") as some_file,
    open("otherfile.txt") as other_file,
):
    ...

from contextlib import redirect_stdout

with (open("somefile.txt", "w") as some_file,
      redirect_stdout(some_file)):
    ...
```

В данном коде видно, что мы даже можем ссылаться на переменную, созданную одним контекстным менеджером (... as some_file) в следующем за ним менеджере!

Это всего лишь два из многих новых форматов в Python 3.10. Улучшенный синтаксис довольно гибок, поэтому можете быть уверены, что новый Python обработает все, что вы ему "скормите".

В следующих главах мы рассмотрим запись структурированных данных в файл.

Глава 24.

ХРАНЕНИЕ СТРУКТУРИРОВАННЫХ ДАННЫХ В ФАЙЛАХ

Текстовые файлы хороши тем, что их можно просматривать и редактировать в любом текстовом редакторе. Ну или почти в любом – главное, чтобы выбранный текстовый редактор поддерживал кодировку, в которой записан файл. Но функциональность текстовых файлов ограничена. Вы не можете просто взять и сохранить в них любые данные, например, словарь или список. Вам придется изобретать какой-то формат файла или использовать уже существующие, например, XML. Но все это сложно. Ведь Python позволяет хранить в файлах структурированные данные, что и будет показано в этой главе.

24.1. Введение в консервацию

В Python есть понятие консервации. ***Консервация*** – *это долговременное хранение структурированных данных*. Программист может "законсервировать" в файл, то есть сохранить в неизменном виде структуру, такую как список или словарь. Консервация похожа на сериализацию данных в PHP – там любую переменную (объект, список и т.д.) можно сохранить в виде строки, которую потом можно записать куда угодно – в базу данных, в Cookies, в файл.

Для поддержки консервации нужно импортировать в программу два модуля:

```
import pickle, shelve
```

Первый модуль позволяет консервировать структуры данных и сохранять их в файлах, а модуль *shelve* обеспечивается произвольный доступ к законсервированным объектам.

Для записи бинарных (двоичных) данных нам нужно открыть файл в двоичном режиме (табл. 24.1).

Таблица 24.1. Режимы доступа к бинарным файлам

Режим	Описание
"rb"	Чтение информации из бинарного файла, если файл не существует, вы получите сообщение об ошибке
"wb"	Запись в бинарный файл. Если он существует, он будет перезаписан. Если не существует, то будет создан
"ab"	Дозапись в бинарный файл. Если он существует, новые данные будут записаны в его конец, если не существует, то будет создан
"rb+"	Чтение и запись в бинарный файл. Если файл не существует, вы получите сообщение об ошибке
"wb+"	Запись и чтение из бинарного файла. Если не существует, то будет создан, если существует, то будет перезаписан
"ab+"	Дозапись и чтение бинарного файла. Если существует, данные будут дописаны в конец, если не существует, то будет создан

Посмотрим, как можно законсервировать данные в файл (листинг 24.1).

Листинг 24.1. Консервация список

```
import pickle, shelve

first_name = ["Оля", "Вася", "Коля"]
last_name = ["Петрова", "Пупкин", "Смирнов"]

datafile = open("names.dat", "wb")

pickle.dump(first_name, datafile)
```

```
pickle.dump(last_name, datafile)

datafile.close()
```

Итак, у нас есть два списка, которые нам нужно записать в файл. Мы открываем файл в режиме "wb". Кодировку указывать не нужно, поскольку данные будут записаны в двоичном виде (рис. 24.1).

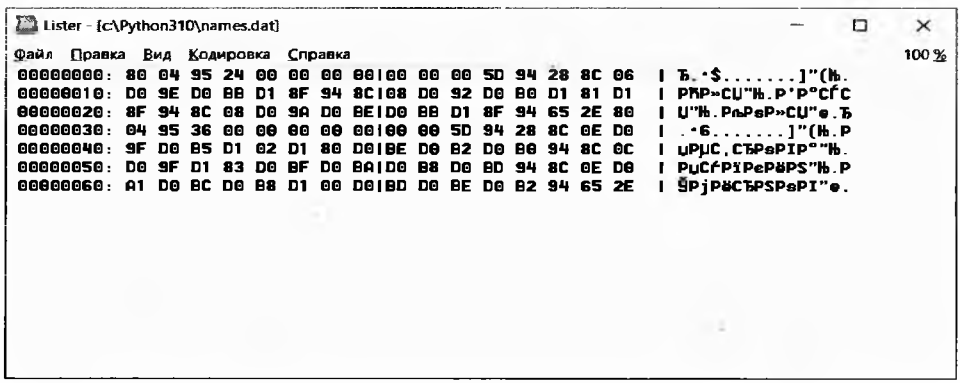


Рис. 24.1. Данные записаны в двоичном виде

После того, как мы открыли файл в бинарном режиме, с помощью метода pickle.dump() мы можем записать наши списки в открытый файл. После этого файл нужно закрыть.

Вы можете законсервировать объекты разных типов:

- Числа
- Строки
- Кортежи
- Списки
- Словари

24.2. Чтение данных из бинарного файла

Чтобы прочитать данные из бинарного файла, используется метод load(). Ему нужно передать только один параметр – файловый объект. Данные читаются в том же порядке, в котором они были записаны. Если вы сначала

записали список *first_name*, а потом – *last_name*, то именно в таком порядке их и нужно читать:

```
datafile = open("names.dat", "rb")
fnames = pickle.load(datafile)
lnames = pickle.load(datafile)
datafile.close()

print(fnames)
print(lnames)
```

Вывод программы будет таким:

```
['Оля', 'Вася', 'Коля']
['Петрова', 'Пупкин', 'Смирнов']
```

24.3. Произвольный доступ к законсервированным данным

Представим, что мы законсервировали несколько списков, а теперь хотим получить доступ к одному произвольному списку. То есть мы не хотим читать все списки подряд, пока не прочитаем тот, который нам нужен. В этом нам поможет модуль *shelve*, который мы подключили ранее к программе.

Первым делом нам нужно создать полку *s*:

```
s = shelve.open("names2.dat")
```

Данная функция работает подобно файловой функции open(), но открывает не текстовый файл, а файл с консервированными объектами. Доступ к полке мы будем осуществлять через переменную *s*, которая работает как словарь, содержимое которого хранится в файле.

Функции shelve.open() нужно передать только имя файла. Второй необязательный параметр – это режим доступа, который может быть:

- "c" (используется по умолчанию) – открытие файла на чтение или запись, если файл не существует, он будет создан.
- "n" – создание нового файла для чтения и записи, если файл существует, он будет перезаписан.
- "r" – доступ только чтение, если файл не существует, вы получите сообщение об ошибке.
- "w" – доступ только запись, если файл не существует, сообщение об ошибке вам гарантировано.

По функциональности полка ничем не отличается от словаря, и вы можете работать с этой переменной, как с обычным словарем. Вот как можно поместить данные "на полку":

```
s["first_name"] = ["Оля", "Вася", "Коля"]
s["last_name"] = ["Петрова", "Пупкин", "Смирнов"]
```

После этого нужно убедиться, что данные записаны в файл:

```
s.sync()
```

Закрыть полку можно методом close():

```
s.close()
```

Теперь откроем файл полки и прочитаем просто список фамилий:

```
s = shelve.open("names2.dat")
print(s["last_name"])
```

Вывод программы будет таким:

```
['Петрова', 'Пупкин', 'Смирнов']
```

Полки предоставляют более гибкий механизм консервации, поэтому на практике, думаю, вы будете пользоваться именно ими.

Глава 25.

РАБОТА С ПОПУЛЯРНЫМИ ФОРМАТАМИ

Python поддерживает работу с различными популярными форматами данных – XML, CSV, JSON. Вам не нужно изобретать велосипед, а использовать стандартные методы работы с такими файлами.

25.1. Работа с CSV-данными

Начнем рассмотрение форматов с формата CSV. Это довольно популярный формат и многие средства импорта/экспорта работают с ним. От части своей популярности он обязан MS Excel, которая позволяет сохранять в CSV свои таблицы. Для доступа к CSV-файлам нужно импортировать модуль **csv**.

Пусть у нас есть следующий CSV-файл (рис. 25.1), содержащий информацию о версиях iPhone. Напишем программу, выводящую первые N-строк из этого файла (листинг 25.1)

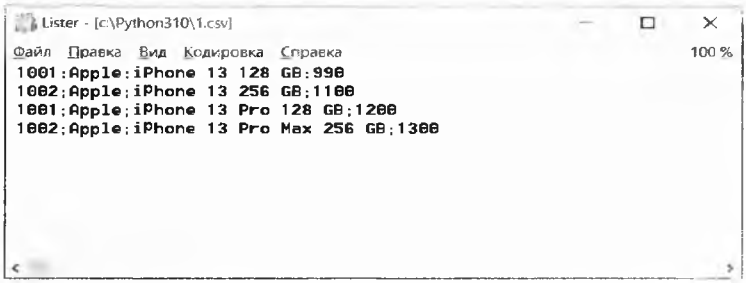


Рис. 25.1. CSV-файл

Листинг 25.1. Чтение из CSV-файла

```
import csv

max = int(input("Сколько строк вывести из файла: "))

k = 0
with open('1.csv') as f:
  f_csv = csv.reader(f)
  headers = next(f_csv)
  for row in f_csv:
    print(row)
    k = k + 1
    if k == max:
        break
```

Вывод программы будет такой:

Рис. 25.2. Вывод из CSV-файла

По умолчанию модуль **csv** создан так, чтобы понимать правила кодирования CSV, используемые Microsoft Excel. Это наиболее распространенный вариант, и он предоставит вам лучшую совместимость. Однако если вы обратитесь к документации по **csv**, вы обнаружите, что есть возможность подстроить кодирование под разные форматы (например, изменить символ разделителя). Например, если вместо разделителя нужно использовать символ табуляции, используйте следующий код:

```
# Пример чтения разделенных табуляцией значений
with open('1.tsv') as f:
  f_tsv = csv.reader(f, delimiter='\t')
  for row in f_tsv:
  # Обрабатываем ряд
...
```

Чтобы записать CSV-данные, вы также можете использовать модуль **csv**, но сначала создайте объект **writer**. Например:

```
headers = ['Symbol','Price','Date','Time','Change','Volume']
rows = [('AA', 39.48, '6/11/2017', '9:36am', -0.18, 181800),
('AIG', 71.38, '6/11/2017', '9:36am', -0.15, 195500),
('AXP', 62.58, '6/11/2017', '9:36am', -0.46, 935000),
]
with open('stocks.csv','w') as f:
  f_csv = csv.writer(f)
  f_csv.writerow(headers)
  f_csv.writerows(rows)
```

25.2. Чтение и запись JSON-данных

Python также обладает поддержкой формата JSON (JavaScript Object Notation). Модуль json предоставляет простой способ кодировать и декодировать данные в JSON. Вам нужно освоить две основные функции – json.dumps() и json.loads(), их работа похожа на другие функции сериализации в других библиотеках, например, в **pickle**. Вот как преобразовать структуру данных Python в JSON:

```
import json

data = {
  'name' : 'Den',
  'shares' : 100,
  'count' : 542.23
}
json_str = json.dumps(data)
```

А вот как вы можете преобразовать JSON-закодированную строку обратно в структуру данных Python:

```
data = json.loads(json_str)
```

Если вы работаете с файлами, а не строками, вы можете использовать функции json.dump() и json.load() для кодирования и декодирования JSON-данных. Например:

```
# Записываем JSON-данные
with open('data.json', 'w') as f:
json.dump(data, f)
# Читаем данные обратно
with open('data.json', 'r') as f:
data = json.load(f)
```

JSON-кодирование поддерживает базовые типы данных – *None*, *bool*, *int*, *float* и *str*, а также списки, кортежи и словари, состоящие из тех базовых типов. Для словарей считается, что ключами будут строки (любые нестроковые ключи в словаре конвертируются в строки при кодировании). Чтобы быть совместимыми со спецификацией JSON, вы должны кодировать только списки и словари. Кроме того, в веб-приложениях принято, что объект верхнего уровня является словарем.

Формат кодирования JSON почти идентичен синтаксису Python за исключением нескольких незначительных изменений. Например, *True* отображается в *true*, *False* – в *false*, а *None* – в *null*. Далее приведен пример того, как выглядит такое кодирование:

```
>>> json.dumps(False)
```

```
'raise'
>>> d = {'a': True,
     ... 'b': 'Hello',
     ... 'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

25.3. Парсинг XML-файлов

Для извлечения данных из простого XML-документа можно использовать модуль xml.etree.ElementTree. Чтобы проиллюстрировать, предположим, что вы хотите проанализировать и сделать сводку произвольной RSS-ленты, содержащей элементы *title*, *pubDate*, *link*. Вот сценарий, который делает это:

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Загружаем RSS-ленту и парсим ее
u = urlopen('http://сайт/rss20.xml')
doc = parse(u)
# Извлекаем и выводим интересующие теги
for item in doc.iterfind('channel/item'):
  title = item.findtext('title')
  date = item.findtext('pubDate')
  link = item.findtext('link')

  print(title)
  print(date)
  print(link)
  print()
```

Очевидно, если вы хотите произвести дополнительную обработку, вам нужно заменить операторы print() на что-то более интересное.

Работа с данными, представленными в виде XML, осуществляется во многих приложениях. Мало того, что XML широко используется в качестве формата для обмена данными в Интернете, также он является стандартным форматом для хранения данных приложения (например, при обработке текста, в музыкальных библиотеках и т.д.). Все сказанное далее предполагает, что читатель уже знаком с основами XML.

Во многих случаях, когда XML используется просто для хранения данных, структура документа компактна и понятна. Например, вот типичная RSS-лента:

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://pur1.org/dc/
elements/1.1/">
<channel>
  <title>Типичная RSS-лента</title>
  <link>http://сайт/</link>
  <language>en</language>
  <description>Описание</description>
  <item>
    <title>Заголовок 1</title>
    <link>Ссылка 1</link>
    <description>Описание 1</description>
    <pubDate>Дата 1</pubDate>
  </item>
  <item>
    <title>Заголовок 2</title>
    <link>Ссылка 2</link>
    <description>Описание 2</description>
    <pubDate>Дата 2</pubDate>
  </item>
...
</channel>
</rss>
```

Функция xml.etree.ElementTree.parse() парсит весь XML-документ в объект документа. Вы можете использовать методы вроде find(), iterfind() и findtext() для поиска определенных XML-документов. Аргументы к этим функциям – имена определенных тегов, вроде *channel/item* или *title*.

При определении тегов вы должны принять полную структуру документа во внимание. Каждая операция *find* работает относительно начального элемента. Аналогично, имя тега, которое вы передаете каждой операции, тоже указывается относительно начального элемента. В примере вызов к doc.iterfind('channel/item') находит все элементы "item", которые находятся внутри элемента "channel". "doc" представляет верхнюю часть документа (элемент "rss"). Более поздние вызовы item.findtext() будут иметь место относительно найденных элементов "item".

У каждого элемента, представленного модулем ElementTree, есть несколько существенных атрибутов и методов, которые полезны при парсинге. Атрибут *tag* содержит имя тега, атрибут *text* содержит текст, а метод get() может быть использоваться для извлечения атрибутов.

Нужно отметить, что xml.etree.ElementTree – не единственное средство для парсинга XML. Для более сложных приложений вы можете рассмотреть использование lxml[1]. Эта библиотека использует тот же API, что и ElementTree, таким образом, пример, показанный в этом рецепте, будет работать и с lxml. Просто нужно заменить первый оператор импорта на from lxml.etree import parse.lxml. Библиотека lxml лучше совместима с XML-стандартами. Также она быстрее и предоставляет дополнительные функции вроде валидации, XSLT и XPath.

25.4. Преобразование словаря в XML

Иногда нужно сохранить содержимое словаря в XML-формате. Хотя библиотека xml.etree.ElementTree обычно используется для парсинга, она также может быть использована для создания XML-документов. Например, рассмотрим эту функцию:

```
from xml.etree.ElementTree import Element

def dict_to_xml(tag, d):
  '''
  Преобразуем простой словарь из пар  ключей/значений в XML
  '''
  elem = Element(tag)
  for key, val in d.items():
    child = Element(key)
    child.text = str(val)
    elem.append(child)
  return elem
```

Вот пример использования:

```
>>> s = { 'name': 'Den', 'shares': 70, 'price':590.1 }
>>> e = dict_to_xml('test', s)
>>> e
```

1 https://pypi.python.org/pypi/lxml

```
<Element 'test' at 0x1004b64c8>
>>>
```

Результат этого преобразования – экземпляр Element. Для ввода/вывода проще конвертировать это в байтовую строку, используя функцию tostring() в xml.etree.ElementTree. Например:

```
>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<test><price>590.1</price><shares>70</shares><name>Den</name></test>'
>>>
```

Если вы хотите присоединить атрибуты к элементу, используйте метод set():

```
>>> e.set('_id','1234')
>>> tostring(e)
b'<test _id="1234"><price>590.1</price><shares>70</
shares><name>Den</name>
</test>'
>>>
```

При создании XML программисты обычно просто создают XML-строку. Например:

```
def dict_to_xml_str(tag, d):
  '''
  Аналог нашей функции, но здесь просто создается XML-строка
  '''
  parts = ['<{}>'.format(tag)]
  for key, val in d.items():
    parts.append('<{0}>{1}</{0}>'.format(key,val))
  parts.append('</{}>'.format(tag))
  return ''.join(parts)
```

Проблема заключается в том, что вы можете запутаться, если попытаетесь заняться обработкой XML-файла вручную. Например, что произойдет, если значения словаря содержат специальные символы?

```
>>> d = { 'name' : '<spam>' }

>>> # Создание строки
>>> dict_to_xml_str('item',d)
'<item><name><spam></name></item>'
>>> # Правильное создание XML
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

Обратите внимание, как в последнем примере заменяются символы < и > на < и >.

Для справки: если вы даже вам захочется вручную обрабатывать такие специальные символы, вы можете использовать функции escape() и unescape() в xml.sax.saxutils. Например:

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

Кроме создания корректного вывода есть и другая причина, почему лучше создавать экземпляры Element вместо строк – они могут быть легко объединены вместе, чтобы создать большой документ. Результирующие экземпляры Element также могут быть обработаны разными способами и при этом не нужно волноваться о парсинге XML. По существу, вы можете сделать всю обработку данных в более высокоуровневой форме, а затем вывести его как строку в самом конце.

25.5. Модификация и перезапись XML-кода

Другая часто распространенная задача – нужно прочитать XML-документ, внести в него изменения и записать обратно как XML.

Модуль xml.etree.ElementTree упрощает выполнение таких задач. По существу, вы начинаете парсинг документа обычным способом. Например, пред-

положим, что у вас есть документ, который называется pred.xml и похож на это:

```
<?xml version="1.0"?>
<stop>
  <id>14791</id>
  <nm>Иванов и Ко</nm>
  <sri>
    <rt>22</rt>
    <d>Улица</d>
    <dd>Индекс</dd>
  </sri>
  <cr>22</cr>
  <pre>
    <pt>5000</pt>
    <fd>Тест</fd>
    <v>1378</v>
    <rn>22</rn>
  </pre>
  <pre>
    <pt>10000</pt>
    <fd>Тест</fd>
    <v>1867</v>
    <rn>22</rn>
  </pre>
</stop>
```

Далее приведен пример использования ElementTree для чтения этого файла и внесения изменения в его структуру:

```
>>> from xml.etree.ElementTree import parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>
>>> # Удаляем несколько элементов
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))
>>> # Вставляем новый элемент после <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'Это тест'
```

```
>>> root.insert(2, e)
>>> # Записываем результат обратно в файл
>>> doc.write('newpred.xml', xml_declaration=True)
>>>
```

В результате этих операций будет создан новый XML-файл, который выглядит так:

```
<?xml version='1.0' encoding='us-ascii'?>
<stop>
  <id>14791</id>
  <nm>Иванов и Ко</nm>
  <spam>Это тест</spam><pre>
    <pt>5000</pt>
    <fd>Тест</fd>
    <v>1378</v>
    <rn>22</rn>
  </pre>
  <pre>
    <pt>1000</pt>
    <fd>Тест</fd>
    <v>1867</v>
    <rn>22</rn>
  </pre>
</stop>
```

Изменение структуры XML-документа достаточно простое, но вы должны помнить, что все модификации обычно делаются в родительском элементе, как будто это список. Например, если вы удаляете элемент, то он удаляется из его непосредственного родителя, используя метод remove() родителя. Если вы вставляете или добавляете новые элементы, то вы также используете методы insert() или append() родителя. Элементами можно также управлять, используя индексы и слайсы, например, element[i] или element[i:j].

Если вам нужно создать новые элементы, используйте класс Element, как показано в решении рецепта.

Глава 26.

ОБРАБОТКА ИСКЛЮЧЕНИЙ

26.1. Что такое исключительная ситуация

Когда Python сталкивается с ошибкой, он останавливает выполнение программы и выводит сообщение об ошибке. Например, вот пример сообщения об ошибке при попытке открыть несуществующий файл:

```
Traceback (most recent call last):
  File "C:/Python310/37.py", line 1, in <module>
    f = open('111.txt', "r")
FileNotFoundError: [Errno 2] No such file or directory: '111.txt'
```

А вот пример сообщения при попытке преобразовать строку в число:

```
Traceback (most recent call last):
  File "C:/Python310/37.py", line 3, in <module>
    num = int("Строка")
ValueError: invalid literal for int() with base 10: 'Строка'
```

Здесь Python попытался преобразовать строку в число, и у него это не получилось, поэтому было сгенерировано исключение. Если такое произойдет на практике, например, когда пользователь вместо числового значения введет строку, то ваша программа "вылетит", а это очень нехорошо. Она должна обрабатывать исключения и сообщать о недопустимом вводе. Ранее уже был приведен пример обработки исключений, теперь настало время поговорить об обработке исключений подробнее.

26.2. Конструкция *try/except*

Типичный пример обработки исключений – конструкция try/except. Аналогичная конструкция используется и в других языках программирования.

В операторе *try* определяется потенциально опасный фрагмент кода – выполнение которого может вызвать исключения. Затем пишется оператор *except*, а блок кода, следующий за ним, будет выполнен только, если система сгенерирована исключение.

В листинге 26.1 приводится пример обработки исключения.

Листинг 26.1. Пример обработки исключения

```
try:
    k = int(input("Введите целое число: "))
    print("Вы ввели: ", k)
except:
    print("Нужно было ввести целое число!!!")
```

Вывод программы:

```
Введите целое число: 1
Вы ввели:  1
>>>
Введите целое число: hello
Нужно было ввести целое число!!!
>>>
```

Если ввести число, то программа выведет его. А вот если вы ввели строку, которая не может быть преобразована в целое число, то будет отображена строка "Нужно было ввести целое число!!!".

26.3. Типы исключений

Разные типы ошибок генерируют разные типы исключений. Как было показано ранее, при попытке открыть несуществующий файл, было сгенерировано исключение `FileNotFoundError`, а при попытке преобразовать число в строку – `ValueError`.

Разные типы исключений представлены в таблице 26.1. Всего существует более 20 исключений, мы же рассмотрим только самые популярные.

Таблица 26.1. Самые распространенные исключения

Исключение	Описание
IOError	Генерируется, если невозможно выполнить операцию ввода/вывода
IndexError	Генерируется, если в последовательности не найден элемент с заданным индексом
KeyError	Если в словаре не найден указанный ключ
NameError	Если не найдено имя (переменной или функции)
SyntaxError	Если в коде обнаружена синтаксическая ошибка
TypeError	Если стандартная операция применяется к объекту неподходящего типа
ValueError	Если операция или функция принимает аргумент с неподходящим значением
ZeroDivisionError	Если есть деление на 0

Мы можем заставить интерпретатор реагировать не на все исключения, а только на определенные, например:

```
try:
    k = int(input("Введите целое число: "))
    print("Вы ввели: ", k)
except ValueError:
    print("Нужно было ввести целое число!!!")
```

Здесь мы обрабатываем исключение определенного типа. При необходимости можно обработать несколько исключений:

Листинг 26.2. Пример обработки нескольких исключений

```
try:
    a = int(input("Введите целое a: "))
    b = int(input("Введите целое b: "))
    print("a/b = ", a/b)
except ValueError:
    print("Нужно было ввести целое число!!!")
```

```
except ZeroDivisionError:
    print("На 0 делить нельзя!")
```

Рис. 26.1. Обработка нескольких исключений

Да, можно обрабатывать все исключения подряд, не указывая тип исключения в *except*. Но если вы будете обрабатывать только определенные исключения, то можете указать разные сообщения пользователю – так вы сделаете свою программу информативнее.

Гибкость Python в том, что он может передать в вашу программу свое ругательство, то есть сообщение об ошибки, но при этом программа не будет прервана. Это называется передачей аргумента исключения (лист. 26.3).

Листинг 26.3. Передача аргумента исключения

```
try:
    a = int(input("Введите целое a: "))
    b = int(input("Введите целое b: "))
    print("a/b = ", a/b)
except ValueError:
    print("Нужно было ввести целое число!!!")
except ZeroDivisionError as e:
    print("На 0 делить нельзя!")
    print("А вот ругательство интерпретатора:")
    print(e)
```

На рис. 26.2 показано, что кроме наших сообщений, самим Python будет выведено сообщение *division by zero*.

Рис. 26.2. Сообщение об ошибке интерпретатора вместе с пользовательским сообщением

26.4. Блок *else*

Посмотрим еще раз на программу:

```
try:
    k = int(input("Введите целое число: "))
    print("Вы ввели: ", k)
except:
    print("Нужно было ввести целое число!!!")
```

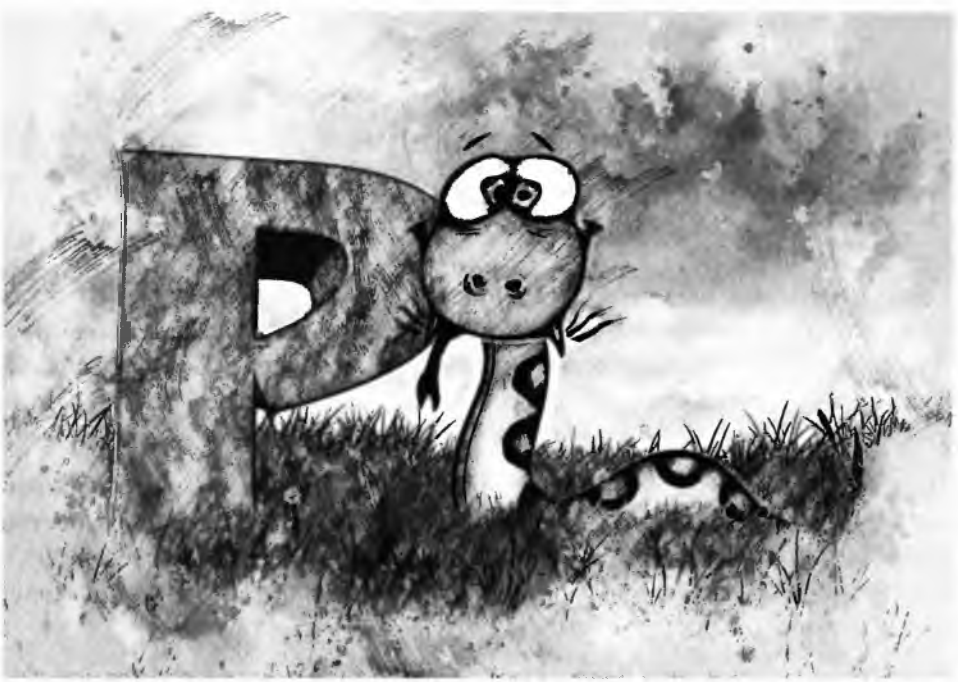
Мы пытаемся ввести число, а потом выводим его. Но правильнее было бы переписать этот код так:

```
try:
    k = int(input("Введите целое число: "))
except:
    print("Нужно было ввести целое число!!!")
else:
    print("Вы ввели: ", k)
```

Ведь в *try* принято помещать код, который может сгенерировать исключение. В *else* мы можем поместить код, который будет выполнен, если все нормально и исключение не будет сгенерировано.

Глава 27.

ПРОГРАММА «ТЕСТЫ 2.0»



27.1. Модификация 1: хранение вопросов и ответов в списках

Ранее нами была написана простейшая программа "Тесты", но мы тогда мало что знали. На данный момент наш уровень знаний существенно вырос, и вместо того, чтобы просто последовательно выводить вопросы функцией print() и сразу же читать ответ, попробуем сделать программу более элегантной.

По сути, нет ничего плохого в том, что мы выводим вопросы и потом читаем ответы линейно. Но если нам понадобится изменить количество вопросов, нам нужно будет дописывать дополнительные print(), input(), делать дополнительные расчеты, то есть вносить существенные изменения в код программы.

Попробуем хранить вопросы и ответы в списках. Также у нас будет переменная n, содержащая количество вопросов. Мы с легкостью сможем изменять количество вопросов. Вопросы будем выводить в цикле *for*:

```
i = 0
for i in range(0, n):
    print(questions[i])
    a = int(input("Ваш ответ: "))
    if a == answers[i]:
       mark = mark + 1
       print("Правильно!")
    else:
        print("Неправильно :(")
```

В цикле мы выводим вопрос из списка *questions*, далее читаем ответ. Если ответ совпадает с правильным (которые хранятся в списке *answers*), то мы увеличиваем переменную *mark* и выводим сообщение "Правильно!". В противном случае, выводим сообщение "Неправильно :(". Вывод программы изображен рис. 27.1.

Рис. 27.1. Программа в действии

Как видите, наша программа стала гораздо лучше. Полный код программы приведен в листинге 27.1.

Листинг 27.1. Программа "Тесты, v 1.5"

```
n = 5
mark = 0

questions = ["""
      Компания-разработчик Windows
      1) Mikrosoft
      2) Melkosoft
      3) Cybersoft
      4) Microsoft
      """,
      """
      Самая "яблочная" операционная система
      1) AppleOS
      2) Linux
      3) macOS
      4) FreeBSD
      """,
      """
      Символом какой операционной системы является пингвин
      1) Linux
      2) FreeBSD
      3) OS/2
      4) Windows
      """,
      """
      Сколько бит в одном байте
      1) 8
      2) 6
      3) 4
      4) 2
      """,
      """
      Сколько байт в одном килобайте
      1) 1000
      2) 1024
      3) 1048
      4) 256
      """]

answers = [4, 3, 1, 8, 2]

i = 0
for i in range(0, n):
```

```
    print(questions[i])
    a = int(input("Ваш ответ: "))
    if a == answers[i]:
       mark = mark + 1
       print("Правильно!")
    else:
        print("Неправильно :(")

print("Вы правильно ответили на ", mark, " вопросов из ", n)
```

27.2. Хранения вопросов и ответов в файле. Программа "Редактор тестов"

Все бы хорошо, но большую часть кода нашей программы занимает список вопросов. Давайте это исправим. Первым делом нужно написать программу, которая будет консервировать вопросы и ответы в файл.

Программа "Редактор тестов" будет записывать вопросы и ответы в файл test.dat с помощью модуля **pickle**. Преимущество такого подхода еще в том, что тестируемый не сможет открыть Python-файл программы и просмотреть список правильных ответов. Ведь в test.dat все данные хранятся в двоичном виде. Конечно, если тестируемый сумел выполнить распаковку списка правильных ответов, то можно быть уверенным, что тест по программированию он точно сдаст!

В листинге 27.2 приводится код программы "Редактор тестов". Вы можете использовать ее для редактирования списка вопросов и ответов. В результате работы программы будет создан файл test.dat с законсервированными списками вопросов и правильных ответов.

Листинг 27.2. Редактор тестов

```
import pickle

questions = ["""
      Компания-разработчик Windows
      1) Mikrosoft
      2) Melkosoft
      3) Cybersoft
```

```
        4) Microsoft
        """,
        """
        Самая "яблочная" операционная система
        1) AppleOS
        2) Linux
        3) macOS
        4) FreeBSD
        """,
        """
        Символом какой операционной системы является пингвин
        1) Linux
        2) FreeBSD
        3) OS/2
        4) Windows
        """,
        """
        Сколько бит в одном байте
        1) 8
        2) 6
        3) 4
        4) 2
        """,
        """
        Сколько байт в одном килобайте
        1) 1000
        2) 1024
        3) 1048
        4) 256
        """]

answers = [4, 3, 1, 8, 2]

datafile = open("test.dat", "wb")

pickle.dump(questions, datafile)
pickle.dump(answers, datafile)

datafile.close()
```

Какого-либо вывода программа не генерирует. Переходим к следующему этапу.

27.3. Программа "Тесты v2.0"

Теперь осталось нашу программу "Тесты" научить читать данные из файла test.dat. В листинге 27.3 мы используем обработку исключений – вдруг файл test.dat куда-то денется, и мы не сможем загрузить список вопросов. В этом случае будет выведено соответствующее сообщение. Если же все нормально (блок **else**), мы загрузим списки вопросов и ответов, определим количество вопросов, а остальной код мало чем отличается от кода из листинга 27.1 – мы задаем вопросы, читаем ответы и вычисляем количество правильных.

Листинг 27.3. Программа "Тесты 2.0"

```
import pickle

mark = 0

print("*" * 10, " Тесты 2.0 ", "*" * 10)

# Загружаем списки вопросов и ответов
try:
    datafile = open("test.dat", "rb")
except:
    print("Ошибка при загрузке вопросов!")
else:

    questions = pickle.load(datafile)
    answers = pickle.load(datafile)
    datafile.close()
    n = len(answers)        # К-во вопросов и ответов
    i = 0
    for i in range(0, n):
        print(questions[i])
        a = int(input("Ваш ответ: "))
        if a == answers[i]:
           mark = mark + 1
           print("Правильно!")
        else:
            print("Неправильно :(")
    print("Вы правильно ответили на ", mark, " вопросов из ", n)
```

Рисунок 27.2 подтверждает, что наша программа работает. Код программы стал гораздо компактнее – ведь нам не нужно в нем хранить списки вопросов и ответов.

Рис. 27.2. Программа "Тесты 2.0" в действии

В прошлых главах я обещал вам, что наша программа будет защищена от дурака, то есть от пользователя, который будет вводить некорректные значения. Пока такой защиты у нас нет. В листинге 27.4 будет представлен вариант программы с защитой от неправильного ввода. Защита будет довольно простой: если пользователь введет значение, не соответствующее числовому, его ответ просто будет засчитан как неправильный.

Листинг 27.4. Тесты 2.1 с "защитой от дурака"

```
import pickle

mark = 0

print("*" * 10, " Тесты 2.1 ", "*" * 10)

# Загружаем списки вопросов и ответов
try:
    datafile = open("test.dat", "rb")
except:
```

```
    print("Ошибка при загрузке вопросов!")
else:

    questions = pickle.load(datafile)
    answers = pickle.load(datafile)
    datafile.close()
    n = len(answers)          # К-во вопросов и ответов
    i = 0
    for i in range(0, n):
        print(questions[i])
        try:
            a = int(input("Ваш ответ: "))
            if a == answers[i]:
                mark = mark + 1
                print("Правильно!")
            else:
                print("Неправильно :(")
        except:
                print("Нужно было ввести число. Ответ засчитан как
неправильный!")
    print("Вы правильно ответили на ", mark, " вопросов из ", n)
```

Рис. 27.3. Наша программа после изменений

Наша программа уже значительно лучше, чем была, но все же у нее есть недостатки. Давайте рассмотрим их, чтобы вы знали, как вам можно улучшить ее.

Начнем с формата представления данных. Файл данных хоть и содержит абракадабру, это все же не шифрование и достаточно легко можно добраться к данным, которые хранятся в names.dat. На рис. 27.4 показано, как за несколько секунд я смог "расшифровать" его.

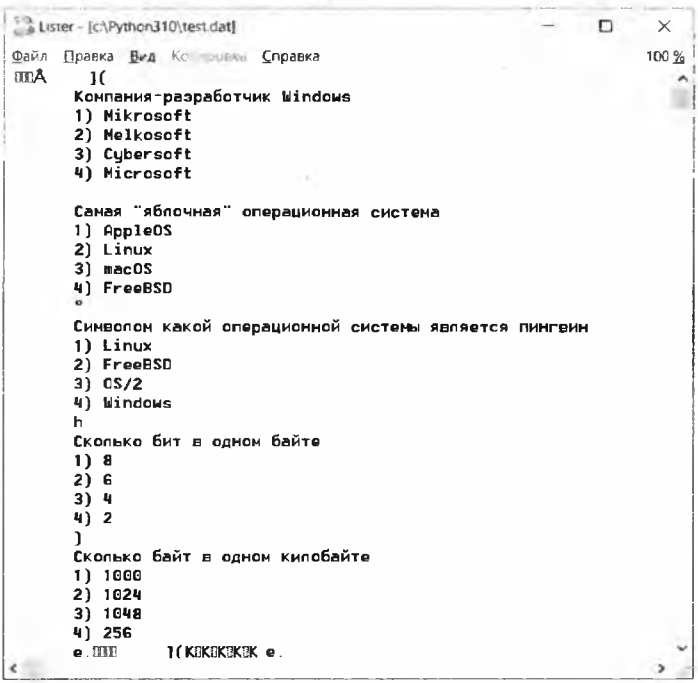


Рис. 27.4. Как относительно легко посмотреть, что находится внутри test.dat

> **Примечание.** Для "расшифровки" достаточно открыть файл в просмотрщике Total Commander и выбрать команду **Вид, Только текст**, а затем – кодировку UTF-8.

Конечно, я не получил список правильных ответов, а только список вопросов, но, думаю, еще немного времени и список ответов также был у меня.

Идем дальше. Программа задает одни и те же вопросы по порядку. Я предлагаю создать больше вопросов, скажем, 20 или даже 50 и задавать случайным образом 10 из них. Тогда каждый пользователь, запустивший программу, получит свой список вопросов. Чем больше вопросов в "базе данных", тем меньше вероятность того, что пользователи будут получать одинаковые вопросы. Даже если в файле с вопросами будет всего 20 вопросов, а задавать программа будет всего 10, то пользователи хотя бы получат вопросы в случайном порядке, что усложнит списывание.

Также неплохо было бы продумать механизм записи результатов. Например, чтобы каждый пользователь вводил свое имя, и чтобы программа заносила в файл его результат. Было бы неплохо, чтобы программа выводила предыдущий результат пользователя. При желании это все не очень сложно сделать.

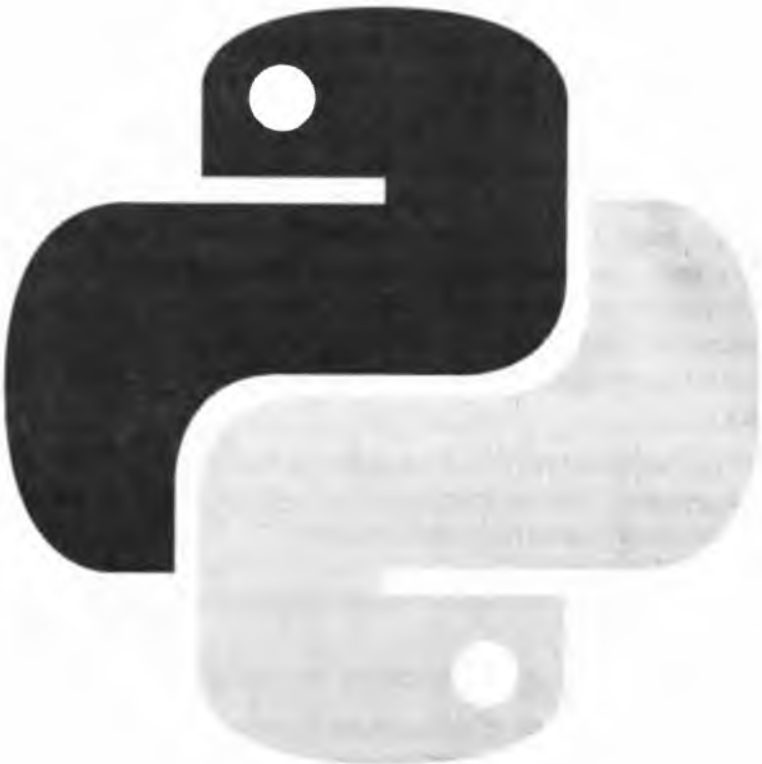
Если совсем уже хотите все усложнить, можно создать программу, которая бы позволяла произвести тестирование различных предметных областей. При этом для каждой предметной области будет свой **файл** с вопросами и ответами, а при запуске программы будет отображаться меню, позволяющее выбрать ту или иную предметную область. Вам понадобится N + 1 файлов. N – это количество предметных областей и один файл, в котором вы будете хранить список предметных областей, например, "Математика", "Информатика" и т.д.

Что касается шифрования, то это самый сложный пункт в нашем *checklist*. Один из способов – использовать библиотеку PyCrypto. Дополнительную информацию о ней можно получить по адресу:

https://rtfm.co.ua/python-bibliotka-pycrypto-shifrovanie-fajla/

А можно использовать и другие средства, например, модуль RSA.

Как по мне, использование криптографии с открытым ключом – выход для нашего приложения. Ведь если шифровать все одним ключом, то этот ключ должен быть или прописан в коде программы или его должен знать тестируемый – чтобы программа сама могла расшифровать вопросы. А все это негативно влияет на безопасность. К сожалению, подробное рассмотрение модуля RSA выходит за рамки этой книги. Если вы заинтересовались, вам в помощь Приложение 1, где рассмотрены различные методы шифрования и криптографии в Python.



ЧАСТЬ VIII.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ПРОГРАММА «ВИРТУАЛЬНАЯ КОШКА»

Объектно-ориентированное программирование (ООП) – современная методика программирования, широко применяемая в последнее время. По сути, большая часть современных программ является объектно-ориентированными, особенно это касается программ с графическим интерфейсом – все они являются объектно-ориентированными.

В этой части книги вы узнаете, как создавать классы, методы и атрибуты, как создавать объекты (экземпляры класса), как ограничивать доступ к атрибутам объекта.

Глава 28.

КЛАССЫ, МЕТОДЫ И ОБЪЕКТЫ

28.1. Основные сущности ООП

Давайте попробуем разобраться, что же представляют собой сущности объектно-ориентированного программирования. Представим, что есть какой-то абстрактный объект, например, стул. У него есть ряд характеристик, например, вес, габариты, тип обивки и т.д. На языке объектно-ориентированного программирования характеристики называются *атрибутами*. У объекта могут быть способы поведения, например, собака лает. На языке ООП такие способы поведения называются *методами*.

На основе описаний, называемых классами, создаются объекты. ***Класс*** *– это фрагмент кода, в котором объявлены атрибуты и методы*. Классы подобны чертежам: это не объекты, а схемы объектов. Используя один и тот же чертеж, можно построить много домов. Аналогично, используя один и тот же класс, можно создать много объектов.

Поскольку объекты создаются на основании классов, то объекты еще называют *экземплярами класса*. У каждого экземпляра класса свой набор атрибутов. Например, один стул может весить 5 килограмм, другой – 10, один иметь обивку из кожи, другой – из велюра.

28.2. Программа "Виртуальная кошка"

На протяжении этой части книги мы будем разрабатывать программу "Виртуальная кошка". Программа будет демонстрировать объектно-ориентированный подход к программированию, хотя аналогичную программу можно было бы написать и без ООП.

28.2.1. Объявление объекта

Листинг 28.1. Создание класса и объекта

```
class Cat(object):
    """ Виртуальная кошка """
    def talk(self):
        print("Мяу")

Bagira = Cat()
Bagira.talk()
```

Вывод программы будет "Мяу". Конечно, ради такой программы не стоило бы применять ООП, но мы все же это сделали. Посмотрим, что именно.

Первым делом мы объявили класс Cat. Для создания класса используется служебное слово *class*. Имя класса начинается с большой буквы. Так принято в Python. Впрочем, вы можете назвать класс и с маленькой буквы, но это будет считаться дурным тоном.

```
class Cat(object):
```

Наш класс будет создан с опорой на фундаментальный встроенный тип *object*, проще говоря, мы наследуем наш класс от *object*. Понимаю, что о наследовании вы пока ничего не знаете, но об этом мы поговорим позже.

Следующая строка документирует класс. Об этом мы уже говорили, когда рассматривали функции:

```
    """ Виртуальная кошка """
```

28.2.2. Объявление метода

Далее мы объявляем метод. По своей структуре он напоминает обычную функцию:

```
def talk(self):
    print("Мяу")
```

По сути, так оно и есть. Можете представлять себе методы, как функции, связанные с объектами. Метод talk() выводит ту самую строку "Мяу".

У метода talk() есть параметр *self*, который мы не используем. Каждый метод экземпляра, то есть метод, которым обладают все объекты данного класса, должен иметь особый первый параметр, называемый по договоренности *self*, Данный параметр позволяет сослаться на сам объект. Пока не задумывайтесь о том, что он означает, но позже вы обязательно узнаете, зачем он нужен.

28.2.3. Создание объекта и вызов метода

Далее мы создаем объект Bagira класса Cat() и вызываем его метод talk():

```
Bagira = Cat()
Bagira.talk()
```

Обратите внимание: сначала создается переменная Bagira, затем используется точечная нотация для вызова метода talk(). Именно при вызове метода talk() мы видим нашу строку "Мяу".

28.3. Конструкторы

Ранее мы создали метод talk(). Но есть специальный метод, называемый конструктором. Данный метод будет вызываться автоматически сразу после создания нового объекта. На практике конструкторы используются для инициализации объекта, например, для загрузки информации из базы данных – очень удобно сразу после создания объекта получить готовый

экземпляр, а не вызывать какой-то метод получения данных. Обычно конструктор устанавливает начальные значения атрибутов объекта, но мы его будем использовать несколько иначе – просто для вывода строки, информирующей о создании нового объекта.

Листинг 28.2. Использование конструктора

```
class Cat(object):
    """ Виртуальная кошка """
    def __init__(self):
        print("Родилась новая кошка!")
    def talk(self):
        print("Мяу")

cat1 = Cat()
cat2 = Cat()
cat1.talk()
cat2.talk()
```

Конструктор также называется методом инициализации, поэтому имя у него специфическое – __init__():

```
    def __init__(self):
        print("Родилась новая кошка!")
```

Вы не можете как-либо изменить имя конструктора. Когда Python видит имя __init__(), то понимает, что это конструктор и этот метод нужно вызвать автоматически.

Далее мы создаем несколько объектов класса Cat. Помните, в начале этой главы я сравнивал класс с чертежом? Так вот, здесь тому подтверждение. На основании одного класса мы создали несколько объектов:

```
cat1 = Cat()
cat2 = Cat()
```

При создании объектов будет выведено сообщение о рождении новой кошки:

```
Родилась новая кошка!
Родилась новая кошка!
```

Затем мы вызываем метод talk() каждого объекта и получаем следующий вывод:

```
Мяу
Мяу
```

28.4. Атрибуты

28.4.1. Создание атрибута

Итак, у нас есть две кошки. Но мы пока еще их никак не назвали. Давайте это исправим, и добавим атрибут *name* в наш класс.

Листинг 28.3. Использование атрибута

```
class Cat(object):
    """ Виртуальная кошка """
    def __init__(self, name):
        print("Родилась новая кошка!")
        self.name = name
    def talk(self):
        print(self.name, ": Мяу")

cat1 = Cat("Багира")
cat2 = Cat("Марсик")
cat1.talk()
cat2.talk()
```

Разберемся, что мы сделали. Мы добавили параметр *name* нашему конструктору. Теперь при создании объекта мы можем назвать наших кошек:

```
cat1 = Cat("Багира")
cat2 = Cat("Марсик")
```

Далее мы выводим (в конструкторе) строку о рождении кошки, а вот следующая за print() строка как раз создает атрибут *name* и присваивает ему значение параметра *name*.

Также мы изменили метод talk(). Теперь он выводит имя кошки, то есть значение атрибута *self.name*:

```
    def talk(self):
        print(self.name, ": Мяу")
```

При желании мы можем обратиться к атрибуту *name* напрямую:

```
print(cat1.name)
```

В нашем случае этот оператор выведет строку:

```
Багира
```

Настало время поговорить о параметре *self*. Параметр *self* автоматически становится ссылкой на объект, по отношению к которому вызван метод. Поэтому через метод *self* метод получает доступ к вызывающему объекту, к его атрибутам и методам.

Если вернуться к нашему методу-конструктору, становится ясно, что при создании класса Cat данный параметр становится ссылкой на объект, а параметр *name* принимает значение "Багира".

Строка кода *self.name = name* создает атрибут *name* у объекта и устанавливает его значение, равное значению параметра *name* ("Багира" в случае с первой кошкой). Аналогично, при создании второго объекта устанавливается значение "Марсик".

28.4.2. Доступ к атрибутам

Если бы к атрибутам невозможно было получить доступ, они были бы бесполезны. В методе talk() мы выводим значение атрибута *name*. Теперь мы знаем, какая кошка мяукает.

Посмотрим, что происходит:

```
cat1.talk()
```

Мы вызываем метод talk(), который объявлен как:

```
def talk(self):
```

То есть этому методу также передается параметр *self*, благодаря чему мы можем получить доступ к атрибуту *name*:

```
    print(self.name, ": Мяу")
```

Обычно атрибуты доступны, как для чтения, так и для изменения не только внутри класса, но и за его пределами:

```
print(cat1.name)
```

Именно поэтому данный код будет выполнен без каких-либо ошибок. Он просто выведет значение атрибута *name*. Для доступа к атрибуту вне класса используется запись:

```
имя_объекта.имя_атрибута
```

Как правило, программисты стремятся избежать доступа к атрибутам вне объявления класса. Позже будет показано, как это сделать.

28.5. Вывод объекта на экран

Попробуйте выполнить оператор:

```
print(cat1)
```

Вы увидите надпись:

```
<__main__.Cat object at 0x00000000036FC358>
```

Ничего особо непонятно. Понятно, что это объект класса Cat и все. Давайте попробуем исправить это. Добавьте метод в наш класс:

```
def __str__(self):
    res = "Объект класса Cat\n name: " + self.name
    return res
```

После этого при print(cat1) вы увидите понятное сообщение (рис. 28.1):

```
Объект класса Cat
 name: Багира
```

Рис. 28.1. Вывод объекта на экран

Полный исходный код примера будет приведен далее (там же можете посмотреть, как правильно определить метод __str__).

28.6. Статические методы. А сколько кошек у нас есть?

Попробуем посчитать наших кошек. Для этого мы создадим атрибут класса *total* и присвоим ему значение *total*. На этот раз мы его объявим иначе – без *self*, а просто внутри класса. Атрибут класса создает каждая команда, кото-

рая расположена внутри класса и присваивает значение ранее не существовавшей переменной (так что не обязательно использовать *self.*для создания атрибута):

```
class Cat(object):
    """ Виртуальная кошка """
    total = 0
```

Далее мы объявляем статический метод count(), в котором мы выводим количество кошек:

```
    @staticmethod
    def count():
        print("Всего кошек: ", Cat.total)
```

При создании кошки в конструкторе нам нужно увеличить *total* на 1:

```
    def __init__(self, name):
        print("Родилась новая кошка!")
        self.name = name
        Cat.total += 1
```

Далее в программе мы вызываем метод count():

```
Cat.count()
```

Теперь попробуем разобраться более детально. Посмотрите на объявление статического метода. Во-первых, есть декоратор @staticmethod. Во-вторых, отсутствует параметр *self.*

Особенность статичных методов в том, что их можно вызвать, даже если не объявлено ни одного объекта, поскольку доступ к статическим методам осуществляется через класс, а не объект:

```
Cat.count()
```

Также особенность атрибута *total* в том, что он относится ко всему классу, а не к конкретному объекту, поэтому, собственно, мы и не использовали *self*.

На этом данная глава близится к завершению. В листинге 28.4 приводится текущая версия нашей "Виртуальной кошки".

Листинг 28.4. "Виртуальная кошка", полный код на данный момент

```
class Cat(object):
    """ Виртуальная кошка """
    total = 0
    @staticmethod
    def count():
        print("Всего кошек: ", Cat.total)
    def __init__(self, name):
        print("Родилась новая кошка!")
        self.name = name
        Cat.total += 1
    def __str__(self):
        res = "Объект класса Cat\n name: " + self.name
        return res
    def talk(self):
        print(self.name, ": Мяу")

cat1 = Cat("Багира")
cat2 = Cat("Марсик")
cat1.talk()
cat2.talk()

Cat.count()
```

28.7. Разница между type() и instance()

Давайте отвлечемся от кошек и рассмотрим разницу между type() и instance() на следующем примере кода:

```
class Polygon:
    def sides_no(self):
        pass

class Triangle(Polygon):
```

```
    def area(self):
        pass

obj_polygon = Polygon()
obj_triangle = Triangle()

print(type(obj_triangle) == Triangle)        # true
print(type(obj_triangle) == Polygon)         # false

print(isinstance(obj_polygon, Polygon))      # true
print(isinstance(obj_triangle, Polygon))     # true
```

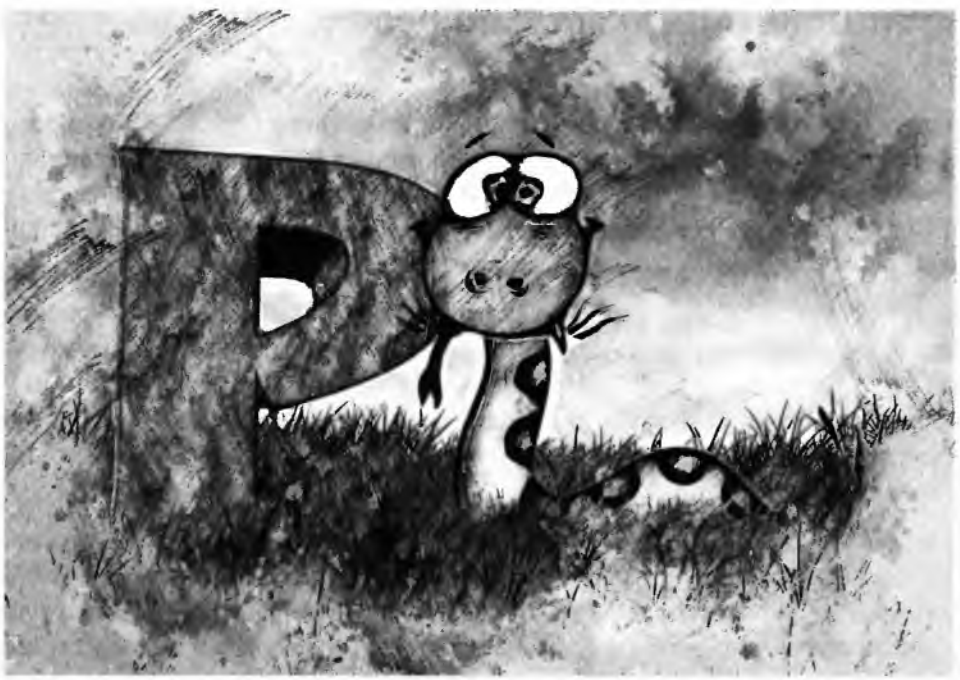
Вывод будет таким:

```
True
False
True
True
```

В приведенном выше примере мы видим, что type() не может различить, связан ли экземпляр класса каким-либо образом с базовым классом. В нашем случае, хотя **obj_triangle** является экземпляром дочернего класса Triangle, он наследуется от базового класса Polygon. Если вы хотите связать объект дочернего класса с базовым классом, вы можете сделать это с помощью instance().

Глава 29.

УПРАВЛЕНИЕ ДОСТУПОМ К АТРИБУТАМ. ЗАКРЫТЫЕ АТРИБУТЫ И МЕТОДЫ



29.1. Понятие инкапсуляции объектов

Ранее мы говорили об инкапсуляции при рассмотрении области видимости функций. Было показано, что инкапсулированная функция скрывает детали своего внутреннего устройства от основной программы и других функций. Если функция реализована правильно, то она взаимодействует с основной программой только путем передачи параметров и возвращения значений.

С объектами дела обстоят аналогично. В основной программе мы должны взаимодействовать с объектами, передавая параметры их методам и получая возвращенные значения. При этом не принято напрямую изменять значения атрибутов объектов и даже обращаться к их значениям. В правильно реализованных классах есть методы для получения и установки значений атрибутов класса.

Представим, что у нас есть класс Car и объект MyCar. У этого класса есть различные атрибуты, один из которых *current_speed*. Это текущая скорость автомобиля. Конечно, можно было бы обращаться к *current_speed* напрямую – увеличивать и уменьшать значение скорости. Теоретически, мы можем так уменьшить скорость, что она станет отрицательной, а это уже невозможно,

по крайней мере, с точки зрения физики. Поэтому гораздо правильнее реализовать метод change_speed(), который перед установкой атрибута *current_speed* будет проверять, чтобы новое значение лежало в диапазоне от 0 до *max_speed* – максимальной скорости, с которой может двигаться автомобиль MyCar. Непрямой доступ к атрибутам через методы является залогом безопасности объекта.

29.2. Закрытые атрибуты и методы

По умолчанию все атрибуты и методы класса являются *открытыми* или *публичными* (*pubic*). Публичные методы можно вызывать за пределами класса, а к публичным атрибутам можно обращаться (читать и изменять их значения) также за пределами класса. Существуют также и *закрытые* или *приватные* (*private*) методы и атрибуты. Приватный метод можно вызывать только внутри класса – в других методах. К приватным атрибутам невозможно обратиться напрямую – в коде программы вы не сможете обратиться к cat1.name, если атрибут *name* объявлен как приватный. Вы не можете ни прочитать, ни изменить его значение.

Давайте попробуем добавить в наш класс закрытый атрибут *__w* (два знака подчеркивания означают, что атрибут будет закрытым), в котором мы будем хранить вес кошки:

```
    def __init__(self, name):
        print("Родилась новая кошка!")
        self.name = name
        Cat.total += 1
        self.__w = 1
```

В конструкторе мы устанавливаем вес кошки – 1 кг. Затем в методе *__str__* мы обращаемся к этому атрибуту при выводе характеристик объекта:

```
    def __str__(self):
        res = "Объект класса Cat\n name: " + self.name + "\nВес:
" + str(self.__w)
        return res
```

Оператор print(cat1) напечатает:

```
Объект класса Cat
 name: Багира
Вес: 1
```

Посмотрим, что произойдет, если мы попытаемся обратиться к этому атрибуту в коде программы:

```
print(ca1.__w)
```

Ошибка будет такая:

```
Traceback (most recent call last):
  File "C:/Python310/cat.py", line 27, in <module>
    print(cat1.__w)
AttributeError: 'Cat' object has no attribute '__w'
```

Интерпретатор сообщает, что у объекта Cat нет атрибута *__w*. Аналогичное сообщение вы увидите, если попытаетесь обратиться просто к *w* (без __):

```
Traceback (most recent call last):
  File "C:/Python310/cat.py", line 27, in <module>
    print(cat1.w)
AttributeError: 'Cat' object has no attribute 'w'
```

Все же есть специальное имя, по которому можно обратиться к закрытому атрибуту:

```
print(cat1._Cat__w)
```

Спрашивается, какой смысл в закрытых атрибутах, если к ним все-таки можно обратиться, хоть и через специальное имя? А смысл в том, что в Python закрытость члена класса – это показатель, что данный атрибут или метод предназначен для внутреннего использования. К тому же у вас не получит-

ся по неосторожности или случайно воспользоваться таким атрибутом или методом.

Кстати, приватный метод объявляется аналогично:

```
def __private_method(self):
   print("Закрытый метод")
```

Обратиться к закрытому методу можно внутри класса, например, в публичном методе:

```
self.__private_method()
```

Если обратиться к приватному методу в основной программе, то мы получим уже знакомое сообщение об ошибке:

```
cat1.__private_method()

Traceback (most recent call last):
  File "C:/Python310/cat.py", line 27, in <module>
    cat1.__private_method()
AttributeError: 'Cat' object has no attribute '__private_method'
```

29.3. Когда нужно использовать закрытые, а когда – открытые методы

Мы только что научились "закрывать" методы и атрибуты класса. Но означает ли это, что нужно закрыть все атрибуты от внешнего мира? Конечно же нет. Нужно пользоваться здравым смыслом.

Если атрибут предназначен для служебного использования и его неправильное изменение может привести к сбою, то лучше его сделать закрытым и изменять только внутри класса – так вы сможете контролировать процесс изменения значения.

29.4. Свойства. Управление доступом к закрытому атрибуту

Для управления доступом к закрытому атрибуту используются свойства – объект с методами, которые позволяют обращаться к атрибутам и ограничивают косвенный доступ.

Сейчас мы создадим свойство *name*, через которое будет возможен непрямой доступ к закрытому атрибуту __name:

```
@property
def name(self):
    return self.__name
```

Для создания свойства нужно написать метод, который возвращает интересующее нас значение (в данном случае – *self.__name*). Перед таким методом нужно объявить декоратор @property.

Теперь через свойство *name* любого объекта класса Cat можно узнать значение закрытого атрибута *__name* этого объекта, причем неважно где – внутри класса или за его пределами. Для этого используется уже знакомая нотация.

Создав свойство, мы приоткрываем закрытый атрибут и делаем его доступным для чтения. Но с помощью свойств можно сделать его доступным не только для чтения, но и для записи:

```
@name.setter
def name(self, new_name):
    if new_name == "":
        print("Укажите имя кошки!")
    else:
        self.__name = new_name
        print("Новое имя: ", self.__name)
```

Сейчас мы создали так называемый сеттер – метод, устанавливающий значение закрытого атрибута. Данный код начинается с декоратора @name.setter. При обращении к атрибуту *setter* свойства name мы говорим, что метод, приводимый далее, устанавливает значение свойства *name*.

Формат сеттера следующий:

```
@имя_свойства.setter
```

Рассмотрим полный код нашей текущей версии "Виртуальной кошки":

Листинг 29.1. Полный код на данный момент (включая код сеттера)

```
class Cat(object):
    """ Виртуальная кошка """
    total = 0
    @staticmethod
    def count():
        print("Всего кошек: ", Cat.total)
    def __init__(self, name):
        print("Родилась новая кошка!")
        self.name = name
        Cat.total += 1
        self.__w = 1
    def __str__(self):
        res = "Объект класса Cat\n name: " + self.name + "\nВес:
" + str(self.__w)
        return res
    def talk(self):
        print(self.name, ": Мяу")
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, new_name):
        if new_name == "":
            print("Укажите имя кошки!")
        else:
            self.__name = new_name
            print("Новое имя: ", self.__name)

cat1 = Cat("Багира")
cat2 = Cat("Марсик")
cat1.talk()
cat2.talk()

cat2.name = "Барсик"
cat2.talk()
```

Рассмотрим вывод программы:

```
Родилась новая кошка!
Новое имя:  Багира
Родилась новая кошка!
Новое имя:  Марсик
Багира : Мяу
Марсик : Мяу
Новое имя:  Барсик
Барсик : Мяу
```

Сначала мы создали два объекта – Багира и Марсик. Багира и Марсик "мяукнули". Обратите внимание, что наш сеттер вызывается, даже если мы устанавливаем имя внутри класса, а не только извне. Затем мы присваиваем новое имя Марсику и теперь он Барсик. Затем вызываем метод cat2.talk(), чтобы убедиться, что переименование прошло успешно.

29.5. "Виртуальная кошка". Готовое решение

Теперь рассмотрим готовое решение. Нашу виртуальную кошку можно будет покормить, с ней можно будет поиграть, ее можно будет взвесить.

Листинг 29.2. Полное решение

```
class Cat(object):
    """ Виртуальная кошка """
    total = 0
    @staticmethod
    def count():
        print("Всего кошек: ", Cat.total)

    def __init__(self):
        print("Родилась новая кошка!")
        self.name = input("Как мы ее назовем? ")
        Cat.total += 1
        self.__w = 300
        self.hunger = 1

    def __str__(self):
        res = "Объект класса Cat\n name: " + self.name + "\nВес:
" + str(self.__w)
        return res

    @property
    def weight(self):
```

```
        return self.__w

    def talk(self):
        print(self.name, ": Мяу")

    def eat(self):
        if self.hunger == 5:
            print("Кошка не голодная")
        else:
            self.hunger += 1
            self.__w += 30
            print("Мур!")

    def play(self):
         self.talk()
         self.__w -= 5
         if self.hunger > 0:
             self.hunger -= 1
         else:
             self.hunger = 1

def main():

    bagira = Cat()
    choice = None
    while choice != "0":
        print \
        ("""
        Что будем делать?

        0 - Выйти
        1 - Поговорить с кошкой
        2 - Покормить
        3 - Поиграть
        4 - Взвесить
        """)

        choice = input(">>: ")
        print()

        # exit
        if choice == "0":
            print("Пока.")

        # послушать
        elif choice == "1":
```

```
            bagira.talk()

        # покормить
        elif choice == "2":
            bagira.eat()

        # поиграть
        elif choice == "3":
            bagira.play()
        elif choice == "4":
            print("Вес: ", bagira.weight, " гр.")

        # неправильный ввод
        else:
            print("\nНеправильный ввод!")

main()
```

При рождении пользователю дается возможность выбрать имя кошки. Также устанавливается начальный вес в 300 грамма и уровень голода 1, что означает, что кошка голодна. При каждом кормлении уровень голода повышается. Когда он достигнет 5, кошка считается неголодной. При каждой игре с питомцем уровень голода понижается на единицу, то есть после того, как с кошкой поиграли, ее можно покормить. При каждом кормлении вес кошки увеличивается на 30 граммов. При каждой игре с кошкой она теряет 5 грамм веса. Да, набрать вес проще, чем похудеть. Соблюдайте идеальный вес вашего питомца! На рис. 29.1 показан процесс игры.

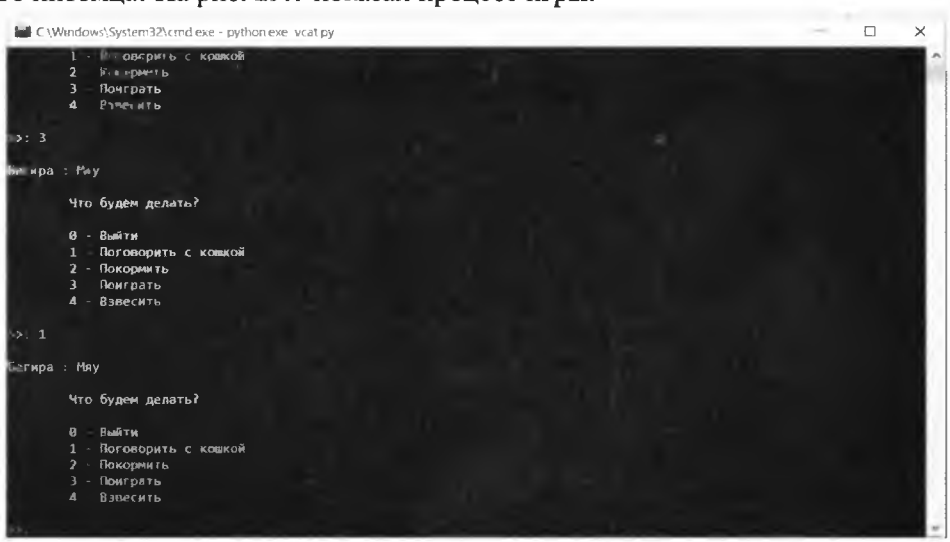
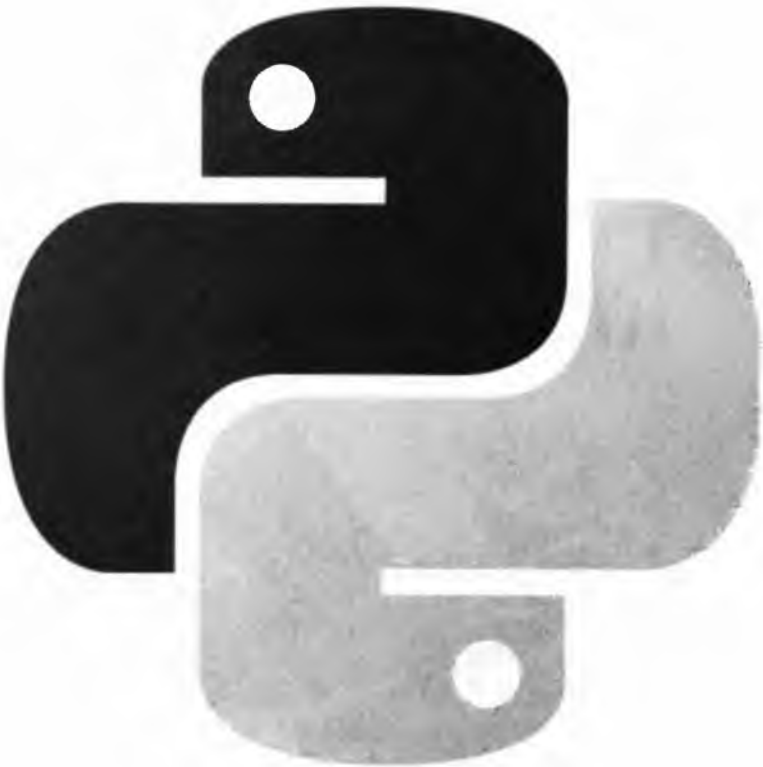


Рис. 29.1. Игра "Виртуальная кошка"

ЧАСТЬ IX.

РАЗРАБОТКА ГРАФИЧЕСКИХ ИНТЕРФЕЙСОВ



Глава 30.

ВВЕДЕНИЕ В *TKINTER*. ПРОГРАММА "ПРИВЕТ ОТ КНОПКИ"

30.1. Этапы разработки приложения с GUI

Большинство прикладных программ обладают графическим интерфейсом пользователя (GUI). К сожалению, наши программы пока не могут похвастаться такими благами. При разработке программ с GUI становятся важными не только алгоритмы обработки данных, но и разработка интерфейса, удобного для пользователя программы, взаимодействуя с которым, он будет определять поведение приложения.

В наши дни пользователь в основном взаимодействует с программой с помощью различных кнопок, меню, значков, вводя информацию в специальные поля, выбирая определенные значения в списках и т. д. Эти элементы и формируют GUI, в дальнейшем мы их будем называть виджетами (от англ. widget).

В Python мы можем создать виджеты с помощью библиотеки **tkinter**. Если ее подключить к нашей программе, то мы сможем использовать ее компоненты для создания графического интерфейса.

Этапы разработки интерфейса с GUI следующие:

1. Импорт библиотеки
2. Создание главного окна
3. Создание виджетов
4. Установка их свойств
5. Определение событий
6. Определение обработчиков событий
7. Расположение виджетов на главном окне
8. Отображение главного окна

Рассмотрим все эти этапы подробнее.

30.1. Импорт библиотеки *tkinter*

Как и любой модуль, **tkinter** в Python можно импортировать двумя способами: командами *import tkinter* или *from tkinter import* *. Далее мы будем пользоваться только вторым способом, т. к. это позволит не указывать каждый раз имя модуля при обращении к объектам, которые в нем содержатся. Следует обратить внимание, что в версии Python 3 и выше имя модуля пишется со строчной буквы (tkinter), хотя в более ранних версиях использовалась прописная (Tkinter).

Первая строчка программы должна выглядеть так:

```
from tkinter import *
```

30.2. Создание главного окна

В современных операционных системах любое пользовательское приложение заключено в окно, которое можно назвать главным, поскольку в нем располагаются все остальные виджеты. Объект окна верхнего уровня создается при обращении к классу Tk модуля **tkinter**. Переменную, связанную с

объектом-окном принято называть *root* (хотя вы можете назвать ее как угодно). Вторая строчка кода:

```
root = Tk()
```

Задать заголовок окна можно так:

```
root.title("Заголовок окна программы")
```

Установить начальные размеры окна можно так:

```
root.geometry("300x250")
```

Если не установить геометрию окна, то размер окна будет подогнан под размет виджетов, содержащихся в окне.

30.3. Создание виджета

После создания основного окна можно создать виджет. Представим, что мы хотим создать кнопку. Кнопка создается при обращении к классу Button модуля **tkinter**. Объект кнопки связывается с какой-нибудь переменной. У класса Button (как и всех остальных классов, за исключением Tk) есть обязательный параметр — объект, которому кнопка принадлежит. Пока у нас есть единственное окно (root), оно и будет аргументом, передаваемым в класс при создании объекта-кнопки:

```
but1 = Button(root)
```

Существует много разных виджетов. В этой книге мы рассмотрим только некоторые из них, а об остальных вы сможете найти информацию в Интернете. Чтобы вы знали, как называется тот или иной виджет, в таблице 30.1 представлены названия виджетов в **tkinter**.

Таблица 30.1. Названия виджетов

Класс	Описание
Button	Кнопка виджет используется для отображения кнопок в приложении
Canvas	Canvas виджет используется для рисования фигур, таких как линии, овалы, многоугольники и прямоугольники, в вашем приложении
Checkbutton	Виджет Checkbutton используются для отображения количества вариантов, как флажки. Пользователь может выбрать несколько вариантов одновременно
Entry	Запись виджет используется для отображения текстового поля в одну строку для приема значения от пользователя
Frame	Виджет Рамы используется в качестве контейнера виджета, чтобы организовать другие виджеты
Label	Этикетка виджет используется, чтобы обеспечить однострочный заголовок для других виджетов. Он также может содержать изображения
Listbox	Listbox виджет используется для получения списка опций для пользователя
Menubutton	Виджет кнопку MENU используются для отображения меню в приложении
Menu	Виджет меню используется, чтобы обеспечить различные команды для пользователя. Эти команды содержатся внутри кнопку MENU
Message	Виджет сообщений используются для отображения многострочных текстовых полей для приема значений от пользователя
Radiobutton	Виджет RadioButton используются для отображения ряда опций, как радиокнопки. Пользователь может выбрать только один вариант в то время

Scale	Шкала виджет используется для обеспечения слайдера виджета
Scrollbar	Виджет Scrollbar используются для добавления прокрутки возможности различных виджетов, таких как списки
Text	Текстовый виджет используется для отображения текста в нескольких строках
Toplevel	Toplevel виджет используется, чтобы обеспечить отдельный контейнер окна
Spinbox	Виджет со счётчиком представляет собой вариант стандартного ввода Tkinter виджета, который может быть использован для выбора из фиксированного числа значений
PanedWindow	PanedWindow является контейнером виджет, который может содержать любое количество панелей, расположенных по горизонтали или по вертикали
LabelFrame	Labelframe является простым контейнером виджетом. Его основная цель состоит в том, чтобы действовать в качестве прокладки или контейнера для сложных оконных раскладок
tkMessageBox	Этот модуль используется для отображения окна сообщений в приложениях

30.4. Установка свойств виджета

У кнопки есть много свойств: надпись, цвет фона, размер и т.д. О них мы поговорим в следующих главах, а пока рассмотрим, как можно установить текст кнопки:

```
but1["text"] = "Привет"
```

Еще пример установки параметров виджетов (более расширенный), но на этот раз при создании самого виджета:

```
btn = Button(text="Hello",              # текст кнопки
             background="#555",         # фоновый цвет кнопки
             foreground="#ccc",         # цвет текста
             padx="20",                 # отступ от границ до содержимого
по горизонтали
             pady="8",                  # отступ от границ до содержимого
по вертикали
             font="16"                  # высота шрифта
             )
```

Часто параметры виджета определяются при его создании:

```
btn = Button(root, text="Hello")
```

Но никто не мешает вам изменить их в процессе работы программы. Второй параметр *options* представляет набор на самом деле набор параметров, которые мы можем установить по их имени. Для кнопки список параметров будет выглядеть так:

- *activebackground* – цвет кнопки, когда она находится в нажатом состоянии
- *activeforeground* – цвет текста кнопки, когда она в нажатом состоянии
- *bd* – толщина границы (по умолчанию 2)
- *bg/background* – фоновый цвет кнопки
- *fg/foreground* – цвет текста кнопки
- *font* – шрифт текста, например, font="Arial 14" – шрифт Arial высотой 14px, или font=("Verdana", 13, "bold") – шрифт Verdana высотой 13px с выделением жирным
- *height* – высота кнопки
- *highlightcolor* – цвет кнопки, когда она в фокусе
- *image* – изображение на кнопке
- *justify* – устанавливает выравнивание текста. Значение LEFT выравнивает текст по левому краю, CENTER – по центру, RIGHT – по правому краю

- *padx* – отступ от границ кнопки до ее текста справа и слева
- *pady* – отступ от границ кнопки до ее текста сверху и снизу
- *relief* – определяет тип границы, может принимать значения SUNKEN, RAISED, GROOVE, RIDGE
- *state* – устанавливает состояние кнопки, может принимать значения DISABLED, ACTIVE, NORMAL (по умолчанию)
- *text* – устанавливает текст кнопки
- *textvariable* – устанавливает привязку к элементу StringVar
- *underline* – указывает на номер символа в тексте кнопки, который подчеркивается. По умолчанию значение -1, то есть никакой символ не подчеркивается
- *width* – ширина кнопки
- *wraplength* – при положительном значении строки текста будут переносится для вмещения в пространство кнопки

30.5. Определение событий и их обработчиков

Что же будет делать кнопка, и в какой момент она это будет делать? Предположим, что задача кнопки вывести какое-нибудь сообщение в поток вывода, используя функцию *print*. Делать она это будет при нажатии на нее левой кнопкой мыши.

Действия (алгоритм), которые происходят при том или ином событии, могут быть достаточно сложным. Поэтому часто их оформляют в виде функции, а затем вызывают, когда они понадобятся. Пусть у нас печать на экран будет оформлена в виде функции *hello*:

```
def hello(event):
     print ("Hello World!")
```

Эту функцию желательно (почти обязательно) размещать в начале кода. Параметр *event* – это какое-либо событие.

Событие нажатия левой кнопкой мыши выглядит так: <Button-1>. Требуется связать это событие с обработчиком (функцией hello). Для связи предназначен метод *bind*. Синтаксис связывания события с обработчиком выглядит так:

```
but1.bind("<Button-1>",hello)
```

30.6. Размещение виджета в окне

Над созданием хорошего и удобного графического интерфейса нужно хорошенько поработать. Мы пока будем использовать самый простой способ – использовать метод pack():

```
but1.pack()
```

Если вы не вызовите этот метод, то кнопка так и не появится в окне программы.

30.7. Отображение главного окна. Программа "Привет от кнопки"

Окно тоже, кстати, не появится, пока вы не вызовите метод mainloop():

```
root.mainloop()
```

Данная строчка кода должна быть всегда в конце скрипта! Полный код нашего приложения с кнопкой приведен в листинге 30.1, а само окно показано на рис. 30.1. Если нажать кнопку "Привет", то в консоли Python вы увидите строчку "Hello, world!" (см. рис. 30.2).

Листинг 30.1. Простое приложение с GUI

```
from tkinter import *

def hello(event):
     print ("Hello World!")

root = Tk()
but1 = Button(root)
but1["text"] = "Привет"
but1.bind("<Button-1>",hello)

but1.pack()
root.mainloop()
```

Рис. 30.1. Наше приложение

Рис. 30.2. Консоль Python

Я вас поздравляю! Вы только что создали, хоть и простое, но приложение с графическим интерфейсом!

Глава 31.

ВИДЖЕТЫ

Каждый виджет имеет определенные свойства, значения которых можно задавать при их создании, а также программировать их изменение при действии пользователя и в результате выполнения программы.

31.1. Кнопки

Объект-кнопка создается вызовом класса Button модуля **tkinter**. При этом обязательным аргументом является лишь родительский виджет (обычно это окно верхнего уровня). Другие свойства могут указываться при создании кнопки или задаваться (изменяться) позже. Синтаксис:

```
переменная = Button (родит_виджет, [свойство=значение, … ….])
```

У кнопки много свойств, в примере ниже указаны лишь некоторые из них.

Пример:

```
from tkinter import *

root = Tk()

but = Button(root,
          text="Это кнопка", #надпись на кнопке
          width=30,height=5, #ширина и высота
          bg="white",fg="blue") #цвет фона и надписи

but.pack()
root.mainloop()
```

Параметры *bg* и *fg* – это сокращения от *background* (фон) и *foreground* (передний план). Ширина и высота измеряются в знакоместах (количество символов).

31.2. Надписи (метки)

Надписи *— это достаточно простые виджеты, содержащие строку (или несколько строк) текста и служащие в основном для информирования пользователя.*

```
lab = Label(root, text="Это надпись! \n Вторая строка.", font="Arial 16")
```

31.3. Поля ввода

Однострочное поле ввода создается с помощью класса *Entry*. Такое поле позволяет ввести только одну строчку кода:

```
ent = Entry(root,width=20,bd=3)
```

Параметр *bd* – это сокращение от *borderwidth* (ширина границы).

Класс *Text* предназначен для создания многострочного текстового поля:

```
tex = Text(root,width=40,
           font="Arial 12",
           wrap=WORD)
```

Последнее свойство (wrap), в зависимости от своего значения, позволяет переносить текст, вводимый пользователем либо по символам, либо по словам, либо вообще не переносить, пока пользователь не нажмет **Enter**.

Как обычно, ширина полей задается в количестве символов.

31.4. Переключатели и флажки

Переключатель никогда не используется по одному. Их используют группами, при этом в одной группе может быть "включена" лишь одна кнопка. Пример:

```
var=IntVar()
var.set(1)
rad0 = Radiobutton(root,text="Windows",
           variable=var,value=0)
rad1 = Radiobutton(root,text="Linux",
           variable=var,value=1)
rad2 = Radiobutton(root,text="macOS",
           variable=var,value=2)
```

Одна группа определяет значение одной переменной, т. е. если в примере будет выбрана радиокнопка *rad2*, то значение переменной *var* будет 2. Изначально также требуется установить значение переменной (выражение var.set(1) задает значение переменной *var* равное 1).

Для создания независимых переключателей (флажков) используется класс Checkbutton. В отличие от зависимых переключателей (радиокнопок), значение каждого флажка привязывается к своей переменной, значение которой определяется опциями *onvalue* (включено) и *offvalue* (выключено) в описании флажка.

Пример:

```
c1 = IntVar()
c2 = IntVar()
che1 = Checkbutton(root,text="Первый флажок",
          variable=c1,onvalue=1,offvalue=0)
che2 = Checkbutton(root,text="Второй флажок",
          variable=c2,onvalue=2,offvalue=0)
```

31.5. Списки

Вызов класса *Listbox* создает объект, в котором пользователь может выбрать один или несколько пунктов в зависимости от значения опции *selectmode*. В примере ниже значение SINGLE позволяет выбирать лишь один пункт из списка.

Пример:

```
r = ['Audi','BMW','Lexus','Toyota']
lis = Listbox(root,selectmode=SINGLE,height=4)
for i in r:
     lis.insert(END,i)
```

Изначально *список* (Listbox) пуст. С помощью цикла *for* в него добавляются пункты из списка (тип данных) *r*. Добавление происходит с помощью специального метода класса *Listbox* — *insert*. Данный метод принимает два параметра: куда добавить и что добавить.

31.6. Рамка. Программа "Цветные рамки"

Рамки – прекрасный инструмент для организации других виджетов в группы внутри окна. Рассмотрим небольшой пример:

Листинг 31.1. Пример использования рамок

```
from tkinter import *

root = Tk()
```

```
fra1 = Frame(root,width=500,height=100,bg="darkblue")
fra2 = Frame(root,width=300,height=200,bg="green",bd=20)
fra3 = Frame(root,width=500,height=150,bg="yellow")

fra1.pack()
fra2.pack()
fra3.pack()

root.mainloop()
```

Данная программа создает три фрейма разного размера. Свойство *bd* (сокращение от *boderwidth*) определяет расстояния от края рамки до заключенных в нее виджетов (если они есть).

Рис. 31.1. Использование рамок

На фреймах также можно размещать виджеты как на основном окне (root). Здесь текстовое поле находится на рамке fra2.

```
ent1 = Entry(fra2,width=20)
ent1.pack()
```

31.7. Дочерние окна

С помощью класс *Toplevel* создаются дочерние окна, на которых также могут располагаться виджеты. Следует отметить, что при закрытии главного окна (или родительского), окно *Toplevel* также закрывается. С другой стороны, закрытие дочернего окна не приводит к закрытию главного.

Пример:

```
win = Toplevel(root,relief=SUNKEN,bd=10,bg="lightblue")
win.title("Дочернее окно")
win.minsize(width=400,height=200)
```

Метод *title* определяет заголовок окна. Метод *minsize* конфигурирует минимальный размер окна (есть метод *maxsize*, определяющий максимальный размер окна). Если значение аргументов *minsize* будет таким же, как у *maxsize*, то пользователь не сможет менять размеры окна.

31.8. Шкала

Назначение ***шкалы*** — *это предоставление пользователю выбора какого-то значения из определенного диапазона*. Внешне шкала представляет собой горизонтальную или вертикальную полосу с разметкой, по которой пользователь может передвигать движок, осуществляя тем самым выбор значения.

```
sca1 = Scale(fra3,orient=HORIZONTAL,length=300,
          from_=0,to=100,tickinterval=10,resolution=5)
sca2 = Scale(root,orient=VERTICAL,length=400,
          from_=1,to=2,tickinterval=0.1,resolution=0.1)
```

Свойства:

- *orient* – определяет направление шкалы;
- *length* – длина шкалы в пикселях;

- *from_* и *to* – с какого значения шкала начинается и каким заканчивается (т. е. диапазон значений);
- *tickinterval* – интервал, через который отображаются метки для шкалы;
- *resolution* – минимальная длина отрезка, на которую пользователь может передвинуть движок.

31.9. Полоска прокрутки

Виджет Scrollbar позволяет прокручивать содержимое другого виджета (например, текстового поля или списка). Прокрутка может быть как по горизонтали, так и по вертикали.

```
from tkinter import *

root = Tk()

tx = Text(root,width=40,height=3,font='14')
scr = Scrollbar(root,command=tx.yview)
tx.configure(yscrollcommand=scr.set)

tx.grid(row=0,column=0)
scr.grid(row=0,column=1)
root.mainloop()
```

В примере сначала создается *текстовое поле* (tx), затем *полоса прокрутки* (scr), которая привязывается с помощью опции *command* к полю *tx* по *вертикальной оси* (yview). Далее поле *tx* изменяется (конфигурируется) с помощью метода *configure*: устанавливается значение опции *yscrollcommand*.

Здесь используется незнакомый нам пока еще метод *grid*, представляющий собой другой способ расположения виджета на окне. Но о нем мы поговорим позже.

31.10. Меню. Программы "Меню" и "Цвет окна"

Меню — это объект, который присутствует во многих пользовательских приложениях. Находится оно под строкой заголовка и представляет собой выпадающие списки под словами; каждый такой список может содержать

другой вложенный в него список. Каждый пункт списка представляет собой команду, запускающую какое-либо действие или открывающую диалоговое окно.

В листинге 31.2 рассматривается программа, создающая окно с полоской меню.

Листинг 31.2. Программа "Меню"

```
from tkinter import *
root = Tk()

# создается объект Меню на главном окне
m = Menu(root)
# окно конфигурируется с указанием меню для него
root.config(menu=m)

# создается пункт меню с размещением на основном меню (m)
fm = Menu(m)
m.add_cascade(label="Файл",menu=fm)
fm.add_command(label="Открыть...")
fm.add_command(label="Создать")
fm.add_command(label="Сохранить...")
fm.add_command(label="Выход")

# второй пункт меню
hm = Menu(m)
m.add_cascade(label="?",menu=hm)
hm.add_command(label="Справка")
hm.add_command(label="О программе")

root.mainloop()
```

Рис. 31.2. Программа с "Меню"

Метод *add_cascade* добавляет новый пункт в меню, который указывается как значение опции *menu*.

Метод *add_command* добавляет новую команду в пункт меню. Одна из опций данного метода (в примере выше ее пока нет) — *command* – связывает данную команду с функцией-обработчиком.

Можно создать вложенное меню. Для этого создается еще одно меню и с помощью *add_cascade* привязать к родительскому пункту:

```
nfm = Menu(fm)
fm.add_cascade(label="Import",menu=nfm)
nfm.add_command(label="Image")
nfm.add_command(label="Text")
```

Меню то мы создали, а теперь нужно разобраться, как привязать к нему обработчики. Каждая команда меню обычно должна быть связана со своей функцией, выполняющей те или иные действия (выражения). Связь происходит с помощью опции *command* метода *add_command*. Функция обработчик до этого должна быть определена.

Для примера выше далее приводятся исправленные строки добавления команд "О программе", "Создать" и "Выход", а также функции, вызываемые, когда пользователь щелкает левой кнопкой мыши по соответствующим пунктам подменю.

```
def new_win():
     win = Toplevel(root)

def close_win():
     root.destroy()

def about():
     win = Toplevel(root)
     lab = Label(win,text="Моя программа 1.0 ")
     lab.pack()

....
fm.add_command(label="Создать",command=new_win)
....
fm.add_command(label="Выход",command=close_win)
```

```
...
hm.add_command(label="О программе",command=about)
```

В листинге 31.3 приводится программа "Цвет окна". Программа создает окно со строкой меню, команды которого позволяют изменить цвет фона и размер основного окна.

Листинг 31.3. Программа "Цвет окна"

```
from tkinter import *
root = Tk()

def colorR():
     fra.config(bg="Red")
def colorG():
     fra.config(bg="Green")
def colorB():
     fra.config(bg="Blue")

def square():
     fra.config(width=640)
     fra.config(height=480)
def rectangle():
     fra.config(width=800)
     fra.config(height=600)

# Параметры окна по умолчанию - 300x100 и цвет фона - черный
fra = Frame(root,width=300,height=100,bg="Black")
fra.pack()

m = Menu(root)
root.config(menu=m)

cm = Menu(m)
m.add_cascade(label="Цвет",menu=cm)
cm.add_command(label="Красный",command=colorR)
cm.add_command(label="Зеленый",command=colorG)
cm.add_command(label="Синий",command=colorB)

sm = Menu(m)
m.add_cascade(label="Размер",menu=sm)
sm.add_command(label="640x480",command=square)
sm.add_command(label="800x600",command=rectangle)

root.mainloop()
```

Рис. 31.3. Программа в действии

Глава 32.

СОБЫТИЯ И МЕТОД *BIND*

32.1. Подробно о методе *bind*

Как мы уже знаем, метод bind() позволяет установить обработчик события, то есть связать событие и функцию (метод), которая будет выполнена, если произойдет установленное событие. Теперь мы рассмотрим его подробнее.

Приложения с графическим интерфейсом пользователя должны не просто красиво отображаться на экране, но и выполнять какие-либо действия, реализуя тем самым потребности пользователя. Ранее было показано, как создать GUI, а сейчас мы рассмотрим, как добавить ему функциональность, т.е. возможность совершать с его помощью те или иные действия.

В отличие от консольных приложений, которые обычно выполняются при минимальных внешних воздействиях, графическое приложение обычно ждет каких-либо внешних воздействий, например, щелчка пользователя по кнопке, а затем уже выполняет запрограммированное действие. Внешнее воздействие на графический компонент называется событием. Событий достаточно много, о них мы поговорим в следующем разделе, а сейчас пока будем одно событие – щелчок левой кнопкой мыши.

Одним из способов связывания виджета, события и функции (того, что должно происходить после события) является использование метода *bind*. Синтаксис связывания следующий:

```
виджет.bind(событие, функция)
```

32.2. Программа "Просмотрщик файлов"

Сейчас мы напишем простенькую программу, которая будет выводить содержимое текстового файла в многострочное поле для ввода текста. Параллельно программа будет демонстрировать привязку обработчика события, а также доступ к значениям текстовых полей ввода:

- Чтение значения из однострочного поля ввода (имя файла)
- Установка значения многострочного поля ввода (содержимое файла)

Наша программа (для упрощения кода) будет состоять всего из трех виджетов:

- **Поле ввода** – в него нужно ввести имя файла
- **Кнопка**
- **Многострочное поле ввода** – в него будет загружаться содержимое файла при нажатии кнопки

Работать программа будет так:

- После запуска она ждет пока пользователь введет имя файла и нажмет кнопку **Открыть**. Имя файла будет получено из переменной *ent*
- Если файл существует, его содержимое будет загружено в виджет *tex*
- Если файл не существует, в *tex* будет загружено соответствующее сообщение

Полный код программы приведен в листинге 32.1.

Листинг 32.1. Просмотрщик файлов

```
def output(event):
     # Получаем содержимое текстового поля
     s = ent.get()
     try:
        txt = open(s, "r", encoding='utf-8')
        content = txt.read()
        tex.delete(1.0,END)
        tex.insert(END, content)
     except:
        tex.delete(1.0,END)
        tex.insert(END, "Файл не существует")


from tkinter import *
root = Tk()

# Создаем виджеты
ent = Entry(root,width=20)
but = Button(root,text="Открыть")
tex = Text(root,width=80,height=30,font="Courier 12",wrap=WORD)
tex.insert(END, "Введите имя текстового файла и нажмите кнопку Открыть")

# Располагаем виджеты в окне программы
ent.grid(row=0,column=0)
but.grid(row=2,column=0)
tex.grid(row=1,column=0)

# Устанавливаем обработчик события
but.bind("<Button-1>",output)

# Запускаем программу
root.mainloop()
```

Обратите внимание на некоторые моменты. Например, на то, как устанавливается новое значение *tex*: сначала мы очищаем текущее содержимое, затем добавляем методом insert() новое:

```
        tex.delete(1.0,END)
        tex.insert(END, content)
```

1.0 означает первую строку и первый символ (нумерация символов начинается с 0), END – последний символ.

Также посмотрите на строку, устанавливающую параметры *text*, в ней мы помимо всего прочего задаем шрифт:

```
tex = Text(root,width=80,height=30,font="Courier 12",wrap=WORD)
```

Расположением виджетов в окне программы "занимается" сетка – *grid*. Мы задаем "координаты" каждого виджета в виде пары ряд и колонка:

```
ent.grid(row=0,column=0)
but.grid(row=2,column=0)
tex.grid(row=1,column=0)
```

32.3. Типы событий

Пока мы умеем использовать только одно событие – щелчок левой кнопкой мыши. Понятно, существуют и другие события. Чаще всего программам приходится реагировать на события клавиатуры и мыши.

При вызове метода *bind* событие передается в качестве первого аргумента. Название события заключается в кавычки, а также в знаки < и >. Событие описывается с помощью зарезервированных последовательностей ключевых слов.

32.3.1. События мыши. Программа "Реагируем на мышь"

К событиям мыши относятся следующие события:

- <Button-1> – щелчок левой кнопкой мыши
- <Button-2> – щелчок средней кнопкой мыши
- <Button-3> – щелчок правой кнопкой мыши
- <Double-Button-1> – двойной клик левой кнопкой мыши
- <Motion> – движение мыши

Это не все, но основные события. В листинге 32.2 приведена программа "Реагируем на мышь". Программа устанавливает заголовок главного окна в зависимости от события мыши.

Листинг 32.2. Реагируем на мышь

```
from tkinter import *
def b1(event):
     root.title("Левая кнопка мыши")
def b3(event):
     root.title("Правая кнопка мыши")
def move(event):
     root.title("Движение мышью")

root = Tk()
root.minsize(width = 500, height=400)

root.bind('<Button-1>',b1)
root.bind('<Button-3>',b3)
root.bind('<Motion>',move)

root.mainloop()
```

32.3.2. События клавиатуры. Программа "Реагируем на клавиатуру"

К событиям клавиатуры относят следующие:

- Нажатия буквенных клавиш, которые записываются без угловых скобок, например, 'A'
- Нажатия комбинаций клавиш – пишутся через тире, например, <Control-Shift>
- Для неалфавитных клавиш есть специальные зарезервированные слова:
  - <Return> – Enter
  - <space> – Пробел
  - <Shift> – Shift
  - <Alt> – Alt

- <Control> – Ctrl
- и т.д.

Напишем программу "Реагируем на клавиатуру". В окне будет поле ввода и метка. При нажатии **Enter** метод *caption* будет устанавливать текст метки – по введенному в поле тексту. При нажатии Ctrl + Z будет произведен выход (вызван метод destroy() для основного окна, что означает выход из программы).

Листинг 32.3. Реагируем на клавиатуру

```
from tkinter import *

def exit_(event):
     root.destroy()
def caption(event):
     t = ent.get()
     lbl.configure(text = t)

root = Tk()

# Создаем виджеты
ent = Entry(root, width = 40)
lbl = Label(root, width = 80)

# Располагаем виджеты
ent.pack()
lbl.pack()

ent.bind('<Return>',caption)
root.bind('<Control-z>',exit_)

root.mainloop()
```

32.4. Особенности работы с виджетом Text

В этой главе было показано, как удалять и добавлять содержимое виджета Text. Сейчас мы рассмотрим процесс работы с ним подробнее. Данный элемент предоставляет большие возможности для работы с текстовой информацией. Помимо разнообразных операций с текстом и его форматированием, в

экземпляр объекта Text можно вставлять другие виджеты (следует отметить, что такая же возможность существует и для Canvas). В данном уроке рассматриваются лишь некоторые возможности виджета Text на примере создания окна с текстовым полем, содержащим форматированный текст, кнопку и возможность добавления экземпляров холста.

Создадим новое текстовое поле и установим некоторые его параметры:

```
tx = Text(font=('times',12),width=50,height=15,wrap=WORD)
tx.pack(expand=YES,fill=BOTH)
```

Попробуем добавить какой-нибудь текст с помощью метода *insert*, которому нужно передать два параметра: место, куда вставить, и объект, который следует вставить. Объектом может быть строка, переменная, ссылающаяся на строку или какой-либо другой объект. Место вставки может указываться несколькими способами. Один из них — это индексы. Они записываются в виде 'x.y', где *x* – это строка, а *y* – столбец. При этом нумерация строк начинается с единицы, а столбцов с нуля. Например, первый символ в первой строке имеет индекс '1.0', а десятый символ в пятой строке — '5.9'.

```
tx.insert(1.0, 'Строка 1\n\
Строка 2\n\n
Строка 3\n')
```

Комбинация символов '\n' создает новую строку (т.е. при интерпретации последующий текст начнется с новой строки). Одиночный символ '\' никак не влияет на отображение текста при выполнении кода, его следует вставлять при переносе текста при написании программы.

Если содержимого текстового поля нет вообще, то единственный доступный индекс — '1.0'. В заполненном текстовом поле вставлять можно в любое место (где есть содержимое).

Если выполнить программу, содержащий только данный код (+ импорт модуля Tkinter, + создание главного окна, + mainloop() в конце), то мы увидим текстовое поле с тремя строчками текста. Текст не отформатирован.

Для форматирования различных областей текста сначала нужно эти области выделить – метод *tag_add*. Затем данные области нужно сконфигурировать – установить настройки шрифта и т.д.

```
# установка тегов для областей текста
tx.tag_add('title','1.0','1.end')
tx.tag_add('special','6.0','8.end')
tx.tag_add('special','3.0','3.11')

# конфигурирование тегов
tx.tag_config('title',foreground='red',
          font=('Aria1',14,'underline'),justify=CENTER)
tx.tag_config('special',background='grey85',font=('times',10,'bold'))
```

Добавление тега осуществляется с помощью метода *tag_add*. Первый атрибут — имя тега (произвольное), далее с помощью индексов указывается, к какой области текстового поля он прикрепляется (начальный символ и конечный). Вариант записи как '1.end' говорит о том, что нужно взять текст до конца указанной строки. Разные области текста могут быть помечены одинаковым тегом.

Метод *tag_config* применяет те или иные свойства к тегу, указанному в качестве первого аргумента.

В виджет Text можно вставлять не только текст, но и другие объекты:

```
def erase():
     tx.delete('1.0',END)
...
# добавление кнопки
bt = Button(tx,text='Очистить',command=erase)
tx.window_create(END,window=bt)
```

Здесь мы добавили кнопку с надписью "Очистить", очищающую содержимое виджета. Виджеты добавляются в текстовое поле с помощью метода *window_create*, где в качестве первой опции указывается место добавления, а второй (window) — в качестве значения присваивается переменная, связанная с объектом.

А вот как можно вставить холст:

```
def smiley(event):
     cv = Canvas(height=30,width=30)
     cv.create_oval(1,1,29,29,fill="yellow")
     cv.create_oval(9,10,12,12)
```

```
        cv.create_oval(19,10,22,12)
        cv.create_polygon(9,20,15,24,22,20)
        tx.window_create(CURRENT,window=cv)
...
tx.bind('<Button-1>',smiley)
```

Здесь, при щелчке левой кнопкой мыши в любом месте текстового поля, будет вызываться функция *smiley*. В теле данной функции создается объект холста, который в конце с помощью метода *window_create* добавляется на объект *tx*. Место вставки указано как CURRENT, т. е. "текущее" – это там, где был произведен щелчок мышью.

Подробно о графике мы поговорим в следующей части книги.

Глава 33.

ОБРАБОТКА ПЕРЕКЛЮЧАТЕЛЕЙ

33.1. Обработка зависимых переключателей (радиокнопок)

Ранее было показано, как получить значение и обработать событие некоторых виджетов. Сейчас мы поговорим о переключателях. Начнем с зависимых переключателей. Библиотека Tkinter содержит специальные классы, объекты которых выполняют роль переменных для хранения значений о состоянии различных виджетов. Изменение значения такой переменной ведет к изменению и свойства виджета, и наоборот: изменение свойства виджета изменяет значение ассоциированной переменной.

Существует несколько таких классов Tkinter, предназначенных для обработки данных разных типов.

- StringVar() – для строк;
- IntVar() – целых чисел;
- DoubleVar() – дробных чисел;
- BooleanVar() – для обработки булевых значений (*true* и *false*).

Ранее мы использовали переменную-объект типа IntVar() при создании группы радиокнопок:

```
var=IntVar()
var.set(1)
rad0 = Radiobutton(root,text="Windows",variable=var,value=0)
rad1 = Radiobutton(root,text="Linux",variable=var,value=1)
rad2 = Radiobutton(root,text="macOS",variable=var,value=2)
```

Здесь создается объект класса IntVar и связывается с переменной *var*. С помощью метода *set* устанавливается начальное значение, равное 1. Три радиокнопки относятся к одной группе: об этом свидетельствует одинаковое значение опции (свойства) *variable*. Variable предназначена для связывания переменной Tkinter с радиокнопкой. Опция *value* определяет значение, которое будет передано переменной, если данная кнопка будет в состоянии "включено". Если в процессе выполнения программы значение переменной *var* будет изменено, то это отразится на группе кнопок. Например, это делается во второй строчке кода: включена кнопка *rad1*.

Метод *set* позволяет устанавливать значения переменных, а метод *get*, наоборот, позволяет получать (узнавать) значения для последующего их использования. Пример:

```
def display(event):
     v = var.get()
     if v == 0:
          print ("Вы выбрали Windows")
     elif v == 1:
          print ("Вы выбрали Linux")
     elif v == 2:
          print ("Вы выбрали macOS")

but = Button(root,text="Получить значение")
but.bind('<Button-1>',display)
```

При вызове функции *display* в переменную *v* "записывается" значение, связанное в текущий момент с переменной *var*. Чтобы получить значение переменной *var*, используется метод *get* (вторая строчка кода).

Рассмотрим полный код примера с использованием радиокнопок (лист. 33.1)

Листинг 33.1. Выбор ОС с помощью радиокнопок

```
from tkinter import *

def sel():
   selection = "Вы выбрали " + str(var.get())
   label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Windows", variable=var, value=1,
                 command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Linux", variable=var, value=2,
                 command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="macOS", variable=var, value=3,
                 command=sel)
R3.pack( anchor = W)

label = Label(root)
label.pack()
root.mainloop()
```

Программа работает так. Сначала мы создаем радиокнопки, а затем назначаем команду *sel* для каждой радиокнопки. При выборе той или иной кнопки передается значение, указанное в *value*. Мы его выводим в надписи внизу окна.

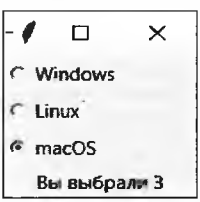


Рис. 33.1. Как работают радиокнопки

33.2. Обработка независимых переключателей. Программа выбора операционной системы

С независимыми переключателями все немного не так. Поскольку состояния флажков независимы друг друга, то для каждого должна быть введена собственная ассоциированная переменная-объект. Пример приведен в листинге 33.2.

Листинг 33.2. Программа выбора ОС

```
from tkinter import *

root = Tk()

var0=StringVar() # значение каждого флажка
var1=StringVar() # хранится в собственной переменной
var2=StringVar()
# если флажок установлен, то в ассоциированную переменную
# (var0,var1 или var2) заносится значение onvalue,
# если флажок снят, то - offvalue.
ch0 = Checkbutton(root,text="Windows",variable=var0,
          onvalue="windows",offvalue="-")
ch1 = Checkbutton(root,text="Linux",variable=var1,
          onvalue="linux",offvalue="-")
ch2 = Checkbutton(root,text="macOS",variable=var2,
          onvalue="macos",offvalue="-")

lis = Listbox(root,height=3)
def result(event):
     v0 = var0.get()
     v1 = var1.get()
     v2 = var2.get()
     l = [v0,v1,v2]     # значения переменных заносятся в список
     lis.delete(0,2)    # предыдущее содержимое удаляется из Listbox
     for v in l:        # содержимое списка l последовательно ...
          lis.insert(END,v) # ...вставляется в Listbox

but = Button(root,text="Получить значения")
but.bind('<Button-1>',result)

ch0.deselect()       # "по умолчанию" флажки сняты
ch1.deselect()
ch2.deselect()

ch0.pack()
```

```
ch1.pack()
ch2.pack()
but.pack()
lis.pack()

root.mainloop()
```

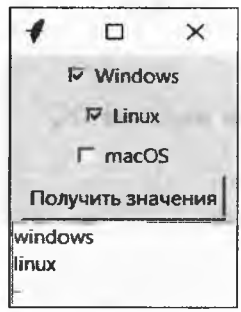


Рис. 33.2. Независимые переключатели в действии

Глава 34.

ДИАЛОГОВЫЕ ОКНА

34.1. Диалоги открытия и сохранения файлов. Программы "Просмотрщик файлов" и "Редактор файлов"

Ранее нами была разработана программа "Просмотрщик файлов", в которой имя файла приходилось вводить вручную. В Python есть возможность использовать стандартные диалоги открытия и сохранения файлов. Вообще-то различных диалоговых окон в Tkinter довольно много, мы рассмотрим лишь самые популярные из них.

Рассмотрим, как запрограммировать с помощью Tkinter вызов диалоговых окон открытия и сохранения файлов и работу с ними. При этом требуется дополнительно импортировать "подмодуль" Tkinter – tkinter.filedialog, в котором описаны классы для окон данного типа.

```
from tkinter import *
from tkinter.filedialog import *
```

```
root = Tk()
op = askopenfilename()
sa = asksaveasfilename()

root.mainloop()
```

Здесь создаются два объекта (*op* и *sa*): один вызывает диалоговое окно "Открыть", а другой "Сохранить как...". При выполнении программы, они друг за другом выводятся на экран после появления главного окна. Если не создать *root*, то оно все равно появится на экране, однако при попытке его закрытия в конце возникнет ошибка.

Давайте модифицируем нашу программу просмотра файлов, добавив в нее диалог выбора файла. Поскольку окно сохранения файла нам пока не нужно, то закомментируем эту строчку кода или удалим. В результате должно получиться примерно так:

```
from tkinter import *
from tkinter.filedialog import *

root = Tk()
txt = Text(root,width=40,height=15,font="12")
txt.pack()

op = askopenfilename()

root.mainloop()
```

При запуске скрипта появляется окно с текстовым полем и сразу диалоговое окно "Открыть". Однако если мы попытаемся открыть какой-нибудь текстовый файл, то в лучшем случае ничего не произойдет. И это понятно, ведь нужно еще написать код, который бы прочитал содержимое файла, который выбрал пользователь, и загрузил бы его в текстовое поле.

Метод *input* модуля **fileinput** может принимать в качестве аргумента адрес файла, читать его содержимое, формируя список строк. Далее с помощью цикла *for* можно извлекать строки последовательно и помещать их, например, в текстовое поле.

```
.....
import fileinput
.....
```

```
for i in fileinput.input(op):
     txt.insert(END,i)
.....
```

Обратите внимание на то, как происходит обращение к функции *input* модуля **fileinput** и его импорт. Дело в том, что в Python уже встроена своя функция *input* (ее назначение абсолютно иное) и во избежание "конфликта" требуется четко указать, какую именно функцию мы имеем в виду. Поэтому вариант импорта *'from fileinput import input'* здесь не подходит.

Окно "Открыть" запускается сразу при выполнении программы. В реальных программах так не должно быть. Необходимо связать запуск окна с каким-нибудь событием. Пусть это будет щелчок на пункте меню.

Полный код нашего "Просмотрщика файлов" приведен в листинге 34.1.

Листинг 34.1. Просмотрщик файлов v 2.0

```
from tkinter import *
from tkinter.filedialog import *
import fileinput

def _open():
     global txt
     op = askopenfilename()
     print(op)
     f = open(op, "r", encoding='utf-8')
     content = f.read()
     txt.delete(1.0,END)
     txt.insert(END, content)


root = Tk()

m = Menu(root)
root.config(menu=m)

fm = Menu(m)
m.add_cascade(label="Файл",menu=fm)
fm.add_command(label="Открыть...",command=_open)

txt = Text(root,width=40,height=15,font="Courier 10")
txt.pack()

root.mainloop()
```

По сравнению с первой версией программа стала гораздо серьезнее. Во-первых, у нее есть меню. Во-вторых, загрузка файла осуществляется с вызовом диалога открытия файла. При выборе команды "Открыть" выполняется команда *_open*, посмотрим, что она делает:

```
def _open():
     op = askopenfilename()
     print(op)
     f = open(op, "r", encoding='utf-8')
     content = f.read()
     txt.delete(1.0,END)
     txt.insert(END, content)
```

В переменную *op* записывается имя выбранного файла. Для контроля мы его выводим на консоль, но в реальных программах – это лишнее. Далее, все как было раньше – мы открываем файл, читаем его весь функцией read() и добавляем в текстовое поле.

Давайте превратим нашу программу **Просмотрщик файлов** в **Редактор файлов**. Сделать это просто – нужно добавить возможность сохранения значения, введенного в текстовое поле.

За сохранение содержимого текстового поля в файл будет отвечать следующая функция:

```
def _save():
     sa = asksaveasfilename()
     content = txt.get(1.0,END)
     f = open(sa,"w", encoding='utf-8')
     f.write(content)
     f.close()
```

Она вызывает диалог сохранения файла, получает имя файла, который был выбран пользователем, а после этого сохраняет текст из текстового поля в этом файле в кодировке UTF-8. Полный код программы "Редактор файлов" приведен в листинге 34.2.

Листинг 34.2. Редактор файлов

```
from tkinter import *
from tkinter.filedialog import *
```

```
import fileinput

def _open():
     op = askopenfilename()
     print(op)
     f = open(op, "r", encoding='utf-8')
     content = f.read()
     txt.delete(1.0,END)
     txt.insert(END, content)

def _save():
     sa = asksaveasfilename()
     content = txt.get(1.0,END)
     f = open(sa,"w", encoding='utf-8')
     f.write(content)
     f.close()


root = Tk()

m = Menu(root)
root.config(menu=m)

fm = Menu(m)
m.add_cascade(label="Файл",menu=fm)
fm.add_command(label="Открыть...",command=_open)
fm.add_command(label="Сохранить как...",command=_save)

txt = Text(root,width=40,height=15,font="Courier 10")
txt.pack()

root.mainloop()
```

Программа полностью рабочая и позволяет выполнять простейшее редактирование файлов.

34.2. MessageBox – вывод различных сообщений. Дальнейшая доработка "Текстового редактора"

Еще одна группа диалоговых окон описана в модуле tkinter.messagebox. Это достаточно простые диалоговые окна для вывода сообщений, предупреждений, получения от пользователя ответа "да" или "нет" и т. п.

Дополним нашу программу пунктом "Выход" в подменю "Файл", и пунктом "О программе" в подменю "Справка".

Листинг 34.3. Текстовый редактор v 1.1

```
from tkinter import *
from tkinter.filedialog import *
from tkinter.messagebox import *
import fileinput

def _open():
     op = askopenfilename()
     print(op)
     f = open(op, "r", encoding='utf-8')
     content = f.read()
     txt.delete(1.0,END)
     txt.insert(END, content)

def _save():
     sa = asksaveasfilename()
     content = txt.get(1.0,END)
     f = open(sa,"w", encoding='utf-8')
     f.write(content)
     f.close()

def close_win():
     if askyesno("Выход", "Вы уверены?"):
          root.destroy()

def about():
     showinfo("Редактор", "Простейший текстовый редактор")


root = Tk()

m = Menu(root)
root.config(menu=m)

fm = Menu(m)
m.add_cascade(label="Файл",menu=fm)
fm.add_command(label="Открыть...",command=_open)
fm.add_command(label="Сохранить как...",command=_save)
fm.add_command(label="Выход",command=close_win)

hm = Menu(m)
m.add_cascade(label="Справка",menu=hm)
```

```
hm.add_command(label="О программе",command=about)

txt = Text(root,width=40,height=15,font="Courier 10")
txt.pack()

root.mainloop()
```

В функции *about* происходит вызов окна *showinfo*, позволяющее выводить сообщение для пользователя с кнопкой **OK**. Первый аргумент — это то, что выведется в заголовке окна, а второй — то, что будет содержаться в теле сообщения. В функции *close_win* вызывается окно *askyesno*, которое позволяет получить от пользователя два ответа (*true* и *false*). В данном случае при положительном ответе сработает ветка *if* и главное окно будет закрыто. В случае нажатия пользователем кнопки "Нет" окно просто закроется (хотя можно было запрограммировать в ветке *else* какое-либо действие).

Рис. 34.2. Информация о программе

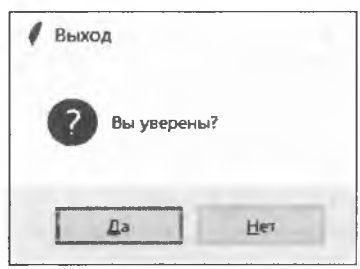


Рис. 34.3. Реакция на команду "Выход"

34.3. Проблема с кодировками. Установка модуля *chardet*

Мы используем кодировку UTF-8 как при чтении файла, так и при сохранении. И это правильно, поскольку это единственная универсальная кодировка, поддерживающая символы различных национальных алфавитов. Вспомните, как было до появления UTF-8 – для русского языка было множество кодировок – CP-1251 (ANSI), CP-866, KOI8-R и др. Первые две использовались в Windows – она в графической оболочке, другая в DOS. Третья использовалась в Linux. Были также собственные варианты и для macOS.

При сохранении содержимого редактора проблем не возникнет – все, чтобы вы не ввели – будет сохранено в UTF-8. А вот с загрузкой текстового файла все сложнее. Если вы попытаетесь открыть текстовый файл в кодировке, отличной от UTF-8, в консоли Python вы увидите следующее исключение, а содержимое файла не будет загружено:

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "C:\Python310\lib\tkinter\__init__.py", line 1538, in __call__
    return self.func(*args)
  File "C:\Python310\34-3.py", line 10, in _open
    content = f.read()
  File "C:\Python34\lib\codecs.py", line 319, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xcf in position 0:
invalid continuation byte
```

Что делать? Можно, конечно обработать исключение и просто не открывать файлы не в UTF-8 с выводом сообщения пользователю. Но это не выход из ситуации. Гораздо правильнее будет использовать модуль **chardet**, который можно применять как для определения кодировки, так и для декодировки файла.

Скачайте модуль:

http://pypi.python.org/packages/source/c/chardet/chardet-2.1.1.tar.gz

Распакуйте его в каталог C:\PythonNN\chardet-2.1.1 и перейдите в этот каталог (здесь NN – номер версии):

```
cd C:\PythonNN\chardet-2.1.1
```

Введите команду установки модуля:

```
python setup.py install
```

Далее код определения кодировки файла может быть таким:

```
import charder
…
# Открываем файл в бинарном режиме
    f = open(op, "rb")
# Читаем текст
     text = f.read()
# Определяем кодировку
     enc = chardet.detect(text).get("encoding")
     enc = enc.lower()
# Очищаем переменную для экономии файла
     text = ""
# Для проверки правильности работы модуля выводим кодировку на консоль
     print(enc)
# Закрываем файл
    f.close()

# Открываем файл и указываем кодировку, определенную ранее
     f = open(op, "r", encoding=enc)
     content = f.read()
     txt.delete(1.0,END)
     txt.insert(END, content)
```

Можно сразу перекодировать файл при открытии. Код будет примерно таким:

```
        with open(filePath, "rb") as F:
            text = F.read()
            enc = chardet.detect(text).get("encoding")
            if enc and enc.lower() != "utf-8":
```

```
                try:
                    text = text.decode(enc)
                    text = text.encode("utf-8")
                    with open(filePath, "wb") as f:
                        f.write(text)
                        print u"%s сконвертирован." % filePath
                except:
                    print u"Ошибка в имени файла: название содержит
русские символы."
            else :
                print u"Файл %s находится в кодировке %s и не требует
конвертирования." % (filePath, enc)
```

Какой вариант выбрать – решать только вам. Можно и перекодировать файл сразу при открытии. Все равно мы сохраняем файл в кодировке utf-8, поэтому он будет перекодирован – рано или поздно, а уже до загрузки в текстовый редактор или при сохранении – разницы нет.

Есть альтернативный подход, позволяющий обойтись без дополнительных модулей. Подход заключается в следующем: мы определяем список поддерживаемых кодировок и пытаемся открыть файл в каждой из них – пока не будет найдена кодировка, в которой сохранен файл.

```
# попытка открыть файл в неизвестной кодировке.

# все поддерживаемые кодировки.
encoding = [
'utf-8',
'cp866',
'utf-16',
'GBK',
'windows-1251',
'ASCII',
'US-ASCII',
'Big5'
]

correct_encoding = ''

for enc in encoding:
    try:
        open('test.txt', encoding=enc).read()
    except (UnicodeDecodeError, LookupError):
        pass
```

```
    else:
        correct_encoding = enc
        print('Done!')
        break

# Для контроля выводим кодировку
print(correct_encoding)
```

Данный код позволяет открыть файл в любой кодировке, указанной в списке *encoding*. Может такой перебор кодировок и не выглядит вершиной рациональности, но решение работает и не требует установки сторонних модулей. Кроме того, мы же не знаем, как определяет кодировку модуль **chardet** – может, таким же способом :)

ЧАСТЬ X.

ГРАФИКА

Последняя часть книги посвящена графике. Мы разберемся, как создать графические примитивы, как сделать собственный Paint и даже как написать игру на Python.

Глава 35.

ГРАФИЧЕСКИЕ ПРИМИТИВЫ

35.1. Геометрические примитивы. Программа "Нарисуй"

Начнем работу с графикой с геометрических примитивов. Чтобы создать что-то сложное, нужно сначала разобраться с простым.

Базовым при работе с двухмерной графикой является понятие *холста*.

Canvas (*холст*) — *это достаточно сложный объект библиотеки tkinter. Он позволяет располагать на самом себе другие объекты*. Это могут быть как геометрические фигуры, узоры, вставленные изображения, так и другие виджеты (например, метки, кнопки, текстовые поля). И это еще не все. Отображенные на холсте объекты можно изменять и перемещать (при желании) в процессе выполнения программы.

Учитывая все это, холст находит широкое применение при создании GUI-приложений с использованием **tkinter** (создание рисунков, оформление других виджет, реализация функций графических редакторов, программируемая анимация и др.).

В этом разделе будет рассмотрено создание на холсте графических примитивов (линии, прямоугольника, многоугольника, дуги (сектора), эллипса) и текста.

Для того чтобы создать объект-холст необходимо вызвать соответствующий класс модуля **tkinter** и установить некоторые значения свойств (опций). Например:

```
canv = Canvas(root,width=500,height=500,bg="lightgray",
          cursor="pencil")
```

Далее с помощью любого менеджера геометрии разместить на главном окне. Здесь мы создаем холст размером 500x500 пикселей и светло-серым фоном.

Перед тем как создавать геометрические фигуры на холсте следует разобраться с координатами и единицами измерения расстояния. Нулевая точка (0,0) для объекта *Canvas* располагается в верхнем левом углу. Единицы измерения пиксели (точки экрана). У любой точки первое число — это расстояние от нулевого значения по оси X, второе — по оси Y.

Чтобы нарисовать линию на холсте следует к объекту (в нашем случае, **canv**) применить метод *create_line*.

```
canv.create_line(200,50,300,50,width=3,fill="blue")
canv.create_line(0,0,100,100,width=2,arrow=LAST)
```

Четыре числа — это пары координат начала и конца линии, т.е. в примере первая линия начинается из точки (200,50), а заканчивается в точке (300,50). Вторая линия начинается в точке (0,0), заканчивается — в (100,100). Свойство *fill* позволяет задать цвет линии отличный от черного, а *arrow* – установить стрелку (в конце, начале или по обоим концам линии).

Метод *create_rectangle* создает прямоугольник. Аналогично линии в скобках первыми аргументами прописываются четыре числа. Первые две координаты обозначают верхний левый угол прямоугольника, вторые — правый нижний. В примере ниже используется немного иной подход. Он может быть полезен, если начальные координаты объекта могут изменяться, а его размер строго регламентирован.

```
x = 50
y = 500
canv.create_rectangle(x,y,x+80,y+50,fill="white",outline="blue")
```

Опция *outline* определяет цвет границы прямоугольника.

Чтобы создать произвольный многоугольник, требуется задать пары координат для каждой его точки.

```
canv.create_polygon([250,110],[200,150],[300,150],fill="red")
```

Квадратные скобки при задании координат используются для удобочитаемости (их можно не использовать). Свойство *smooth* задает сглаживание.

```
canv.create_polygon([250,100],[200,150],[300,150],fill="yellow")
canv.create_polygon([300,80],[400,80],[450,75],[450,200],
          [300,180],[330,160],outline="white",smooth=1)
```

При создании эллипса задаются координаты гипотетического прямоугольника, описывающего данный эллипс.

```
canv.create_oval([20,200],[150,300],fill="gray50")
```

Более сложные для понимания фигуры получаются при использовании метода *create_arc*. В зависимости от значения опции *style* можно получить сектор (по умолчанию), сегмент (CHORD) или дугу (ARC). Координаты по-прежнему задают прямоугольник, в который вписана окружность, из которой "вырезают" сектор, сегмент или дугу. От опций *start* и *extent* зависит угол фигуры.

```
canv.create_arc([160,230],[230,330],start=0,extent=140,fill="lightgreen")
canv.create_arc([250,230],[320,330],start=0,extent=140,
          style=CHORD,fill="green")
canv.create_arc([340,230],[410,330],start=0,extent=140,
          style=ARC,outline="darkgreen",width=2)
```

Следующий метод объекта **canvas**, который будет рассмотрен в этой главе — это метод, создающий текстовую надпись.

```
canv.create_text(20,330,text="Учимся рисовать",
          font="Verdana 12",anchor="w",justify=CENTER,fill="red")
```

Трудность здесь может возникнуть с пониманием опции *anchor* (якорь). По умолчанию в заданной координате располагается центр текстовой надписи. Чтобы изменить это и, например, разместить по указанной координате левую границу текста, используется якорь со значением *w* (*от англ. west* – запад). Другие значения: *n*, *ne*, *e*, *se*, *s*, *sw*, *w*, *nw*. Если букв, задающих сторону привязки две, то вторая определяет вертикальную привязку (вверх или вниз "уйдет" текст от координаты). Свойство *justify* определяет лишь выравнивание текста относительно себя самого.

В конце следует отметить, что часто требуется "нарисовать" на холсте какие-либо повторяющиеся элементы. Для того чтобы не загружать код, используют циклы. Например, так:

```
x=10
while x < 450:
     canv.create_rectangle(x,400,x+50,450)
     x = x + 60
```

Теперь рассмотрим программу "Нарисуй", которая рисует линию, стрелку, прямоугольник, многоугольник, овал, дугу и выводит текст на холст. Результат работы программы изображен на рис. 35.1.

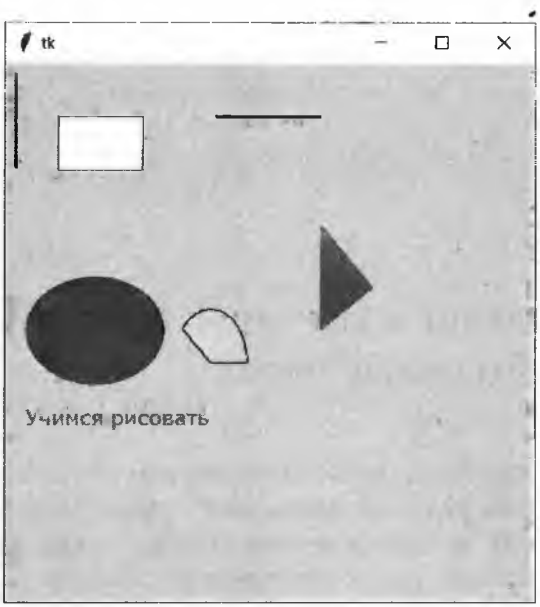


Рис. 35.1. Программа "Нарисуй"

Листинг 35.1. Программа "Нарисуй"

```
from tkinter import *
root = Tk()


canv = Canvas(root,width=500,height=500,bg="lightgray",
          cursor="pencil")

# Линия
canv.create_line(200,50,300,50,width=3,fill="blue")
# Стрелка
canv.create_line(10,10,10,100,width=2,arrow=LAST)

# Прямоугольник
x = 50
y = 50
canv.create_rectangle(x,y,x+80,y+50,fill="white",outline="blue")

# Многоугольник
canv.create_polygon([350,210],[300,250],[300,150],fill="red")

# Овал
canv.create_oval([20,200],[150,300],fill="gray50")

# Дуга
canv.create_arc([160,230],[230,330],start=0,extent=140,fill="lightgreen")

# Текст
canv.create_text(20,330,text="Учимся рисовать",
          font="Verdana 12",anchor="w",justify=CENTER,fill="red")
canv.pack()

root.mainloop()
```

35.2. Обращение к уже существующим графическим примитивам

В прошлом разделе были рассмотрены методы объекта **canvas**, формирующие на нем геометрические примитивы и текст. Однако это лишь часть методов холста. В другую условную группу можно выделить методы, изменяющие свойства уже существующих объектов холста (например, геометрических фигур). И тут возникает вопрос: как обращаться к уже созданным фигурам? Ведь если при создании было прописано что-то вроде

canvas.create_oval(30,10,130,80) и таких овалов, квадратов и др. на холсте очень много, то как к ним обращаться?

Для решения этой проблемы в **tkinter** для объектов холста можно использовать идентификаторы и теги, которые затем передаются другим методам. У любого объекта может быть, как идентификатор, так и тег. Использование идентификаторов и тегов немного различается. Рассмотрим несколько методов изменения уже существующих объектов с использованием при этом идентификаторов. Для начала создадим холст и три объекта на нем. При создании объекты "возвращают" свои идентификаторы, которые можно связать с *переменными* (*oval*, *rect* и *trial* в примере ниже) и потом использовать их для обращения к конкретному объекту.

```
c = Canvas(width=460,height=460,bg='grey80')
c.pack()
oval = c.create_oval(30,10,130,80)
rect = c.create_rectangle(180,10,280,80)
trian = c.create_polygon(330,80,380,10,430,80, fill='grey80',
outline="black")
```

Если вы выполните данный скрипт, то увидите на холсте три фигуры: овал, прямоугольник и треугольник.

Далее можно использовать методы-"модификаторы", указывая в качестве первого аргумента идентификатор объекта. Метод *move* перемещает объект по оси X и Y на расстояние, указанное в качестве второго и третьего аргументов. Следует понимать, что это не координаты, а смещение, т. е. в примере ниже прямоугольник опустится вниз на 150 пикселей. Метод *itemconfig* изменяет указанные свойства объектов, *coords* изменяет координаты (им можно менять и размер объекта)

```
c.move(rect,0,150)
c.itemconfig(trian,outline="green",width=3)
c.coords(oval,200,100,350,350)
```

Если запустить программу, содержащий две приведенные части кода (друг за другом), то мы сразу увидим уже изменившуюся картину на холсте: прямоугольник опустится, треугольник приобретет красный контур, а эллипс сместится и сильно увеличится в размерах. Обычно в программах изменения должны наступать при каком-нибудь внешнем воздействии. Пусть по

щелчку левой кнопкой мыши прямоугольник передвигается на два пикселя вниз (он будет это делать при каждом щелчке мышью):

```
def mooove(event):
     c.move(rect,0,2)
...
c.bind('<Button-1>',mooove)
```

Теперь рассмотрим, как работают теги. В отличие от идентификаторов, которые являются уникальными для каждого объекта, один и тот же тег может присваиваться разным объектам. Дальнейшее обращение к такому тегу позволит изменить все объекты, в которых он был указан. В примере ниже эллипс и линия содержат один и тот же тег, а функция *color* изменяет цвет всех объектов с тегом *group1*. Обратите внимание, что в отличие от имени идентификатора (переменная), имя тега заключается в кавычки (строковое значение).

```
oval = c.create_oval(30,10,130,80,tag="group1")
c.create_line(10,100,450,100,tag="group1")
…
def color(event):
     c.itemconfig('group1',fill="red",width=3)
…
c.bind('<Button-3>',color)
```

Еще один метод, который стоит рассмотреть, это *delete*, который удаляет объект по указанному идентификатору или тегу. В **tkinter** существуют зарезервированные теги: например, *all* обозначает все объекты холста. Так в примере ниже функция *clean* просто очищает холст.

```
def clean(event):
     c.delete('all')
...
c.bind('<Button-2>',clean)
```

Метод *tag_bind* позволяет привязать событие (например, щелчок кнопкой мыши) к определенному объекту. Таким образом, можно реализовать обращение к различным областям холста с помощью одного и того же события.

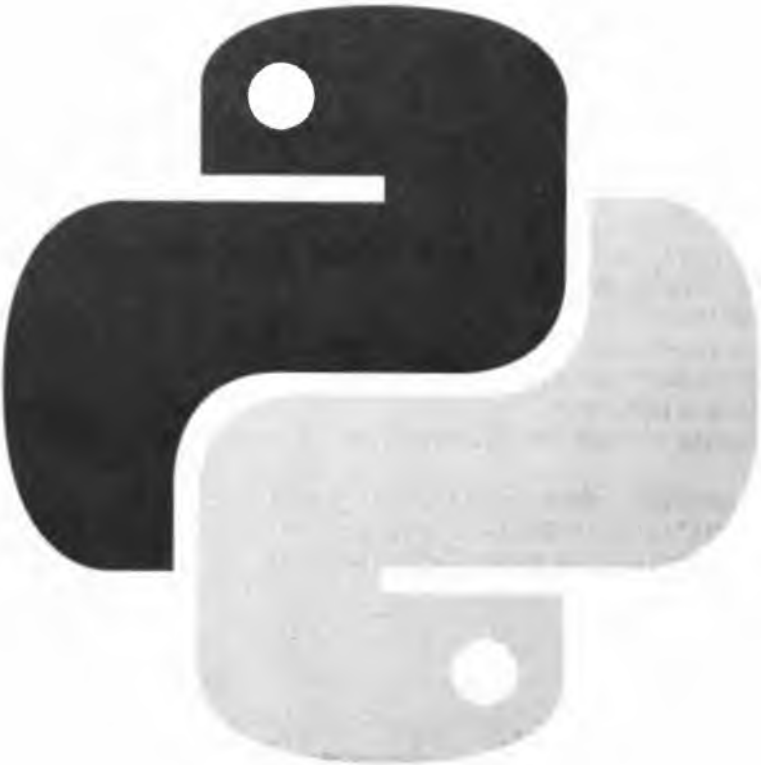
Пример ниже это наглядно иллюстрирует: изменения на холсте зависят от того, где произведен щелчок мышью.

Листинг 35.2. Обращение к графическим объектам по идентификаторам

```
from tkinter import *

c = Canvas(width=460,height=100,bg='grey80')
c.pack()

oval = c.create_oval(30,10,130,80,fill="orange")
c.create_rectangle(180,10,280,80,tag="rect",fill="lightgreen")
trian = c.create_polygon(330,80,380,10,430,80,fill='white',outline="black")

def oval_func(event):
     c.delete(oval)
     c.create_text(30,10,text="Здесь был круг",anchor="w")
def rect_func(event):
     c.delete("rect")
     c.create_text(180,10,text="Здесь был\
ппрямоугольник",anchor="nw")
def triangle(event):
     c.create_polygon(350,70,380,20,410,70,fill='yellow',outline="black")

c.tag_bind(oval,'<Button-1>',oval_func)
c.tag_bind("rect",'<Button-1>',rect_func)
c.tag_bind(trian,'<Button-1>',triangle)

mainloop()
```

Рис. 35.2. Программа в действии

В следующей главе мы создадим более сложную программу, позволяющую рисовать на холсте кистями разных размеров.



Глава 36.

PAINT СВОИМИ РУКАМИ

В этой главе мы напишем простую рисовалку на Python. Программа будет, понятное дело, обладать графическим интерфейсом и позволять выбирать цвет и размер кисти. Наш Paint будет примитивным, но зато мы потренируемся в создании макетов компоновки в **tkinter**, передаче аргументов в функцию-обработчик, а также на практике увидим, как применяются анонимные функции.

36.1. Класс *Paint* и разработка каркаса для приложения

Первым делом мы определим класс *Paint* – уже если и делать программу средней сложности, то применять ООП (листинг 36.1).

Листинг 36.1. Класс Paint

```
from tkinter import *

class Paint(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
```

```
def main():
    root = Tk()
    root.geometry("800x600+300+300")
    app = Paint(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Данный код создает каркас для нашей будущей программы – простенькое окно, которое мы будем дорабатывать.

Далее нам нужно написать для класса *Paint* место setUI, в котором будем задавать расположение всех кнопок, меток и самого холста – поля для рисования. Интерфейс программы будет прост: у нас будет два ряда кнопок. Верхний ряд задает цвет кисти, второй – размер кисти. Под кнопками будет расположен холст.

Листинг 36.2. Метод setUI

```
    def setUI(self):
        # Устанавливаем название окна
        self.parent.title("Simple Paint")
        # Размещаем активные элементы на родительском окне
        self.pack(fill=BOTH, expand=1)

        # Пусть 7-ой столбец растягивается, кнопки не будут разъезжаться при
изменении размера окна
        self.columnconfigure(6, weight=1)
        # То же самое, но для третьего ряда
        self.rowconfigure(2, weight=1)

        # Наш холст, фон - белый
        self.canv = Canvas(self, bg="white")

        # Прикрепляем канвас методом grid. Он будет находится в
        # 3м ряду, первой колонке,
        # и будет занимать 7 колонок, задаем отступы по X и Y в 5 пикселей, и
        # заставляем растягиваться при растягивании всего окна
        self.canv.grid(row=2, column=0, columnspan=7,
                       padx=5, pady=5, sticky=E+W+S+N)

        # Создаем метку для кнопок изменения цвета кисти
        color_lab = Label(self, text="Цвет: ")
        # Устанавливаем созданную метку в первый ряд и первую колонку,
```

```
        # задаем горизонтальный отступ в 6 пикселей
        color_lab.grid(row=0, column=0, padx=6)

        # Создание кнопки:  Установка текста кнопки, задание ширины кнопки
(10 символов)
        red_btn = Button(self, text="Красный", width=10)
        # Устанавливаем кнопку первый ряд, вторая колонка
        red_btn.grid(row=0, column=1)

        # Создание остальных кнопок - аналогично

        green_btn.= Button(self, text="Зеленый", width=10)
        green_btn.grid(row=0, column=2)

        blue_btn = Button(self, text="Синий", width=10)
        blue_btn.grid(row=0, column=3)

        black_btn = Button(self, text="Черный", width=10)
        black_btn.grid(row=0, column=4)

        white_btn = Button(self, text="Белый", width=10)
        white_btn.grid(row=0, column=5)

        # Создаем метку для кнопок изменения размера кисти
        size_lab = Label(self, text="Размер кисти: ")
        size_lab.grid(row=1, column=0, padx=5)
        one_btn = Button(self, text="2x", width=10)
        one_btn.grid(row=1, column=1)

        two_btn = Button(self, text="5x", width=10)
        two_btn.grid(row=1, column=2)

        five_btn = Button(self, text="7x", width=10)
        five_btn.grid(row=1, column=3)

        seven_btn = Button(self, text="10x", width=10)
        seven_btn.grid(row=1, column=4)

        ten_btn = Button(self, text="20x", width=10)
        ten_btn.grid(row=1, column=5)

        twenty_btn = Button(self, text="50x", width=10)
        twenty_btn.grid(row=1, column=6, sticky=W)
```

В метод *__init__* нужно добавить вызов этого метода self.setUI(). Если вы все сделали правильно, то при запуске этой программы вы увидите окно, изображенное на рис. 36.1. Каркас для нашего приложения создан.

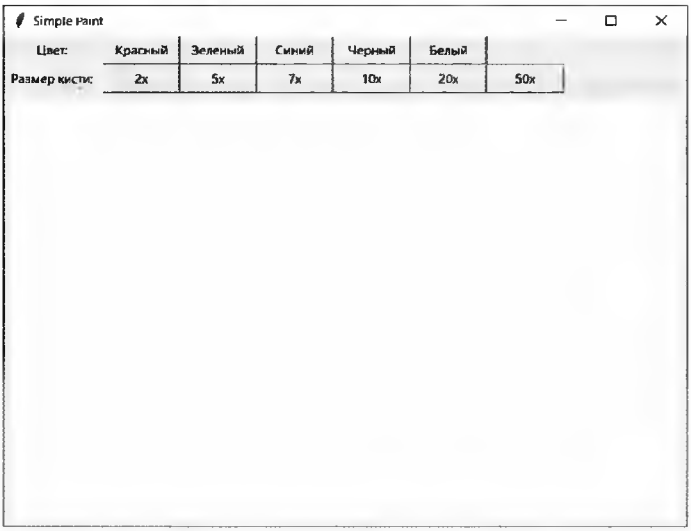


Рис. 36.1. Каркас окна рисовалки

36.2. Разработка метода draw()

Теперь нужно разработать метод рисования на холсте. Пока мы его не разработали, добавьте в *__init__* параметры кисти по умолчанию:

```
self.brush_size = 10
self.color = "red"
```

Метод draw() будет выглядеть так:

```
    def draw(self, event):
        self.canv.create_oval(event.x - self.brush_size,
                              event.y - self.brush_size,
                              event.x + self.brush_size,
                              event.y + self.brush_size,
                              fill=self.color, outline=self.color)
```

Рисование осуществляется путем создания кругов на холсте: пользователь зажимает левую кнопку мыши и при движении мышью, по пути следования курсора будут отрисовываться круги. Метод *draw* принимает аргумент *event*,

на основе которого мы будем формировать овалы. Метод *create_oval* класса *Canvas* получает четыре координаты, на основе которых создается квадрат, в который вписывается круг. В качестве этих координат мы передаем позицию курсора, поэтому первая координата по оси икс будет позиция курсора минус размер кисти, вторая координата по оси икс – позиция курсора плюс размер кисти, то же самое для оси игрек. Это может показаться сложным, но скоро вы запустите наше приложение и увидите все своими глазами.

Осталось только привязать к холсту обработку только что созданного метода. Добавьте следую строку после прикрепления холста (self.canvas.grid...):

```
self.canv.bind("<B1-Motion>", self.draw)
```

Запустите программу прямо сейчас. Вы уже можете рисовать (рис. 36.2)! По сути, основная работа сделана. Осталось научить программу реагировать на изменение цвета и размера кисти.

Рис. 36.2. Простой Paint

36.3. Изменяем цвет и размер кисти

Разработаем метод изменения цвета кисти. Он будет очень прост:

```
def set_color(self, new_color):
    self.color = new_color
```

После этого в каждой кнопке верхнего ряда следует добавить код обработки нажатия этой кнопки по следующему шаблону:

```
red_btn = Button(self, text="Красный", width=10, command=lambda:
self.set_color("red"))
red_btn.grid(row=0, column=1)
```

Код, который мы добавили – *command = lambda: self.set_color("red")*, привязывает функцию с нужным нам аргументом к кнопке. Мы используем lambda-функцию потому что, без *lambda* функция вызовется сразу при создании кнопки, а не только при ее нажатии. Добавив этот код с нужными аргументами (а это "green", "blue", "black", "white") ко всем кнопкам получим возможность изменять цвет кисти. Запустите программу. Теперь вы можете изменять цвет кисти!

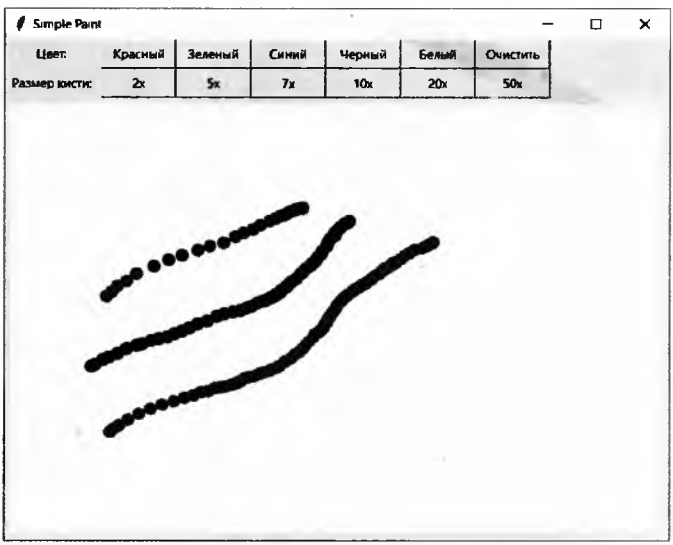


Рис. 36.3. Paint может изменять цвет кисти

Рассмотрим метод изменения размера кисти:

```
def set_brush_size(self, new_size):
    self.brush_size = new_size
```

Модернизируем код каждой кнопки нижнего ряда по следующему шаблону:

```
one_btn = Button(self, text="2x", width=10, command=lambda:
self.set_brush_size(2))
```

Запустим Paint, посмотрим, что у нас вышло. Результат изображен на рис. 36.4. Программа умеет изменять размер и цвет кисти.

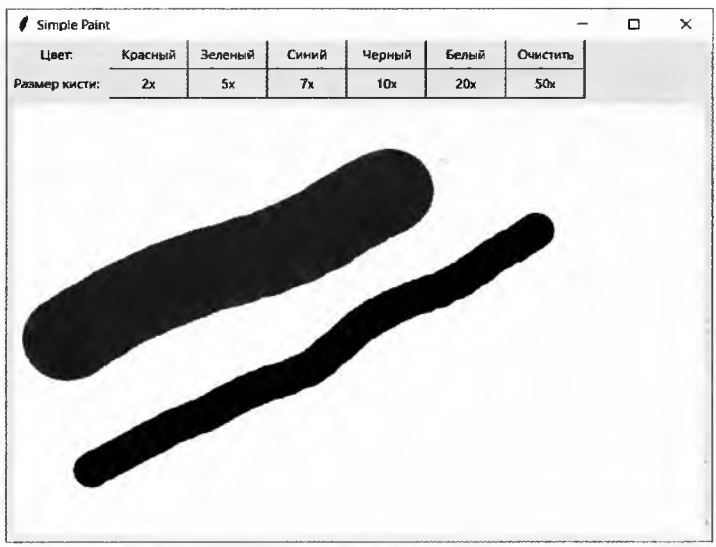


Рис. 36.4. Программа умеет изменять размер и цвет кисти

Осталось добавить одну-единственную кнопку – для нее как раз есть место в нашем окне – **Очистить**. Она очищает холст.

```
clear_btn = Button(self, text="Очистить", width=10, command=lambda:
self.canv.delete("all"))
clear_btn.grid(row=0, column=6, sticky=W)
```

Окончательный вариант нашей программы изображен на рис. 36.5. Полный исходный код программы приведен в листинге 36.3.

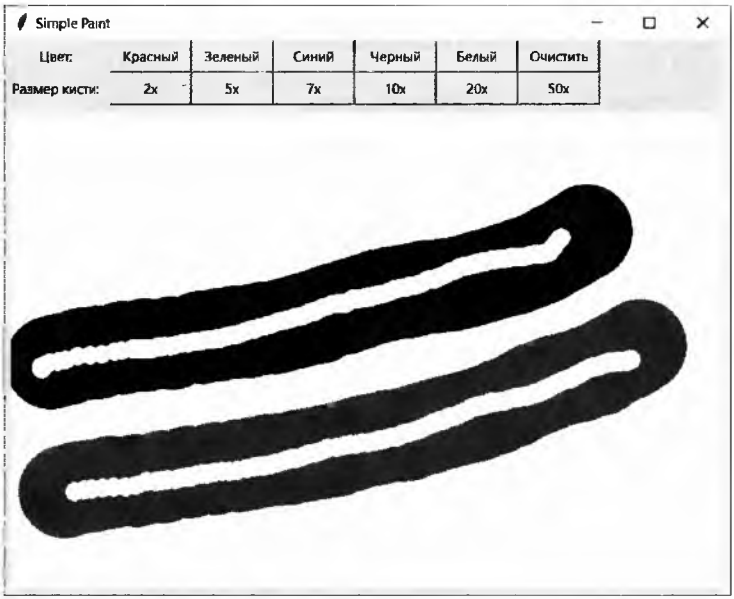


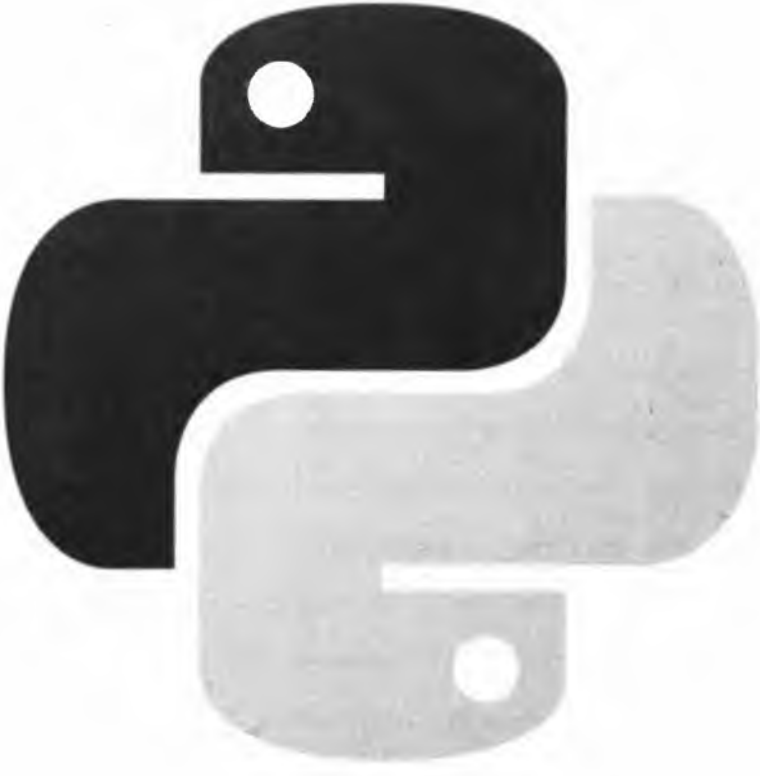
Рис. 36.5. Окончательный вариант Simple Paint

Листинг 36.3. Полный исходный код Simple Paint

```
from tkinter import *

class Paint(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
        # Параметры кисти по умолчанию
        self.brush_size = 10
        self.brush_color = "red"
        self.color = "red"
        # Устанавливаем компоненты UI
        self.setUI()
    # Метод рисования на холсте
    def draw(self, event):
        self.canv.create_oval(event.x - self.brush_size,
                              event.y - self.brush_size,
                              event.x + self.brush_size,
                              event.y + self.brush_size,
                              fill=self.color, outline=self.color)
    # Изменение цвета кисти
    def set_color(self, new_color):
        self.color = new_color
    # Изменение размера кисти
```

```
    def set_brush_size(self, new_size):
        self.brush_size = new_size
    def setUI(self):
        # Устанавливаем название окна
        self.parent.title("Simple Paint")
        # Размещаем активные элементы на родительском окне
        self.pack(fill=BOTH, expand=1)

        # Пусть 7-ой столбец растягивается, кнопки не будут разъезжаться при
изменении размера окна
        self.columnconfigure(6, weight=1)
        # То же самое, но для третьего ряда
        self.rowconfigure(2, weight=1)

        # Наш холст, фон - белый
        self.canv = Canvas(self, bg="white")

        # Прикрепляем канвас методом grid. Он будет находится в 3м ряду,
первой колонке,
        # и будет занимать 7 колонок, задаем отступы по X и Y в 5 пикселей, и
        # заставляем растягиваться при растягивании всего окна
        self.canv.grid(row=2, column=0, columnspan=7,
                       padx=5, pady=5, sticky=E+W+S+N)
        # Задаем реакцию холста на нажатие левой кнопки мыши
        self.canv.bind("<B1-Motion>", self.draw)

        # Создаем метку для кнопок изменения цвета кисти
        color_lab = Label(self, text="Цвет: ")
        # Устанавливаем созданную метку в первый ряд и первую колонку,
        # задаем горизонтальный отступ в 6 пикселей
        color_lab.grid(row=0, column=0, padx=6)

        # Создание кнопки:  Установка текста кнопки, задание ширины кнопки
(10 символов)
        red_btn = Button(self, text="Красный", width=10, command=lambda:
self.set_color("red"))
        # Устанавливаем кнопку первый ряд, вторая колонка
        red_btn.grid(row=0, column=1)

        # Создание остальных кнопок - аналогично

        green_btn = Button(self, text="Зеленый", width=10, command=lambda:
self.set_color("green"))
        green_btn.grid(row=0, column=2)

        blue_btn = Button(self, text="Синий", width=10, command=lambda:
self.set_color("blue"))
        blue_btn.grid(row=0, column=3)
```

```
        black_btn = Button(self, text="Черный", width=10, command=lambda:
self.set_color("black"))
        black_btn.grid(row=0, column=4)

        white_btn = Button(self, text="Белый", width=10, command=lambda:
self.set_color("white"))
        white_btn.grid(row=0, column=5)

        # Создаем метку для кнопок изменения размера кисти
        size_lab = Label(self, text="Размер кисти: ")
        size_lab.grid(row=1, column=0, padx=5)
        one_btn = Button(self, text="2x", width=10, command=lambda:
self.set_brush_size(2))
        one_btn.grid(row=1, column=1)

        two_btn = Button(self, text="5x", width=10, command=lambda:
self.set_brush_size(5))
        two_btn.grid(row=1, column=2)

        five_btn = Button(self, text="7x", width=10, command=lambda:
self.set_brush_size(7))
        five_btn.grid(row=1, column=3)

        seven_btn = Button(self, text="10x", width=10, command=lambda:
self.set_brush_size(10))
        seven_btn.grid(row=1, column=4)

        ten_btn = Button(self, text="20x", width=10, command=lambda:
self.set_brush_size(20))
        ten_btn.grid(row=1, column=5)

        twenty_btn = Button(self, text="50x", width=10, command=lambda:
self.set_brush_size(50))
        twenty_btn.grid(row=1, column=6, sticky=W)

        clear_btn = Button(self, text="Очистить", width=10, command=lambda:
self.canv.delete("all"))
        clear_btn.grid(row=0, column=6, sticky=W)


def main():
    root = Tk()
    root.geometry("800x600+300+300")
    app = Paint(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```



Глава 37.

ПИШЕМ ИГРУ: «ЗМЕЙКА» НА PYTHON

В предыдущей главе было показано, как создать простейшую рисовалку на Python. Толку от этого приложения немного – просто демонстрация принципов объектно-ориентированного программирования, а также практическая работа с холстом. Сейчас мы рассмотрим, как создать игру на Python. Это будет не простейшая консольная игрушка, а полноценная 2D-игра – "Змейка".

37.1. О разработке игры

Разработка игры, даже такой простой как наша – процесс довольно непростой и состоит из множества этапов. Конкретно в нашем случае этапы будут следующие:

- Создание окна приложения
- Объявление вспомогательных переменных
- Создание игрового поля
- Создание классов сегмента и змеи
- Создание разных дополнительных функций

Да, классов у нас будет два. Один – это класс одного блока Змейки – класс сегмента. Второй – класс самой змеи. Запасайтесь терпением – процесс разработки игрушки дело небыстрое!

37.2. Создание окна приложения

Первым делом нужно разработать каркас для нашего приложения – главное окно. Код главного окна приведен в листинге 37.1:

Листинг 37.1. Класс окна Змейки

```
from tkinter import *

# Создаем окно
root = Tk()
# Устанавливаем название окна
root.title("Змейка")

# Запускаем окно
root.mainloop()
```

Данная программа ничего, по сути, не делает – создает окно и устанавливает его заголовок. Окно просто пустое, поэтому никаких иллюстраций не приводится.

37.3. Объявление вспомогательных переменных

Добавим в нашу программу вспомогательные переменные. Их нужно добавить сразу после оператора импорта **tkinker**:

```
# ширина экрана
WIDTH = 800
# высота экрана
HEIGHT = 600
# Размер сегмента змейки
SEG_SIZE = 20
# Переменная, отвечающая за состояние игры
IN_GAME = True
```

Это глобальные переменные, которые мы будем использовать в процессе игры.

37.4. Создание игрового поля

Теперь создадим игровое поле. Его мы реализуем с помощью *Canvas*. Создадим холст нужного нам размера и зальем его зеленым цветом:

```
# создаем экземпляр класса Canvas (его мы еще будем использовать)
# и заливаем все зеленым
c = Canvas(root, width=WIDTH, height=HEIGHT, bg="#005500")
c.grid()
# Наводим фокус на Canvas, чтобы мы могли ловить нажатия клавиш
c.focus_set()
```

Запустите программу. Теперь мы увидим прототип окна нашей игры (рис. 37.1).

Рис. 37.1. Прототип окна игры

37.5. Создание основных классов

Создание окна с холстом – это все очень просто. Самое сложное – это классы игры. Первый класс – это класс сегмента змейки. Он очень прост. Визуально сегмент змейки будет представлен обычным прямоугольником, созданным при помощи метода *create_rectangle* класса *Canvas* модуля **tkinter**:

```
class Segment(object):
    def __init__(self, x, y):
        self.instance = c.create_rectangle(x, y,
                            x+SEG_SIZE, y+SEG_SIZE,
                            fill="white")
```

Класс змейки гораздо сложнее. Ведь он является набором сегментов и нужно учитывать движение змейки, изменение направления, добавление сегмента. Класс змейки длинный, поэтому представлен в отдельном листинге 37.2. Код тщательно комментирован, поэтому обязательно читайте комментарии.

Листинг 37.2. Класс Змейки

```
class Snake(object):
    def __init__(self, segments):
        self.segments = segments

        # список доступных направлений движения змейки
        self.mapping = {"Down": (0, 1), "Up": (0, -1),
                                "Left": (-1, 0), "Right": (1, 0) }
        # изначально змейка двигается вправо
        self.vector = self.mapping["Right"]

    def move(self):
        """ Двигает змейку в заданном направлении """

        # перебираем все сегменты кроме первого
        for index in range(len(self.segments)-1):
            segment = self.segments[index].instance
            x1, y1, x2, y2 = c.coords(self.segments[index+1].instance)
            # задаем каждому сегменту позицию сегмента стоящего после него
            c.coords(segment, x1, y1, x2, y2)

        # получаем координаты сегмента перед "головой"
        x1, y1, x2, y2 = c.coords(self.segments[-2].instance)

        # помещаем "голову" в направлении указанном в векторе движения
```

```
            c.coords(self.segments[-1].instance,
                     x1 + self.vector[0]*SEG_SIZE,
                     y1 + self.vector[1]*SEG_SIZE,
                     x2 + self.vector[0]*SEG_SIZE,
                     y2 + self.vector[1]*SEG_SIZE)

    def change_direction(self, event):
        """ Изменяет направление движения змейки """

        # event передаст нам символ нажатой клавиши
        # и если эта клавиша в доступных направлениях
        # изменяем направление
        if event.keysym in self.mapping:
            self.vector = self.mapping[event.keysym]

    def add_segment(self):
        """ Добавляет сегмент змейке """

        # определяем последний сегмент
        last_seg = c.coords(self.segments[0].instance)

        # определяем координаты куда поставить следуюший сегмент
        x = last_seg[2] - SEG_SIZE
        y = last_seg[3] - SEG_SIZE

        # добавляем змейке еще один сегмент в заданных координатах
        self.segments.insert(0, Segment(x, y))
```

Посмотрим, что есть что. Конструктор (метод *__init__*) задает список доступных направлений движения змейки и задает направление по умолчанию – вправо.

Метод move() двигает змейку в заданном направлении. Метод change_direction() изменяет направление движения змейки. За добавление нового сегмента отвечает метод add_segment().

Попробуем создать змейку. Вставьте следующие строки после c.grid():

```
# создаем набор сегментов
segments = [Segment(SEG_SIZE, SEG_SIZE),
            Segment(SEG_SIZE*2, SEG_SIZE),
            Segment(SEG_SIZE*3, SEG_SIZE)]

# сама змейка
s = Snake(segments)
```

Запустите программу. Наша Змейка пока никуда не двигается. Но она уже есть (рис. 37.2).

Рис. 37.2. Змейка

37.6. Создание вспомогательных функций

Нам понадобятся две вспомогательных функции. Первая – create_block(), которая создает пищу для змейки. В нашем случае пусть это будет яблоко – кружок красного цвета. Чтобы эта функция работала, нужно импортировать модуль **random** – не забудьте об этом.

```
def create_block():
    """ Создает блок в случайной позиции на карте """
    global BLOCK
    posx = SEG_SIZE * (random.randint(1, (WIDTH-SEG_SIZE) /
SEG_SIZE))
    posy = SEG_SIZE * (random.randint(1, (HEIGHT-SEG_SIZE) /
SEG_SIZE))

    # блок это кружочек красного цвета
    BLOCK = c.create_oval(posx, posy,
                          posx + SEG_SIZE,
                          posy + SEG_SIZE,
                          fill="red")
```

Функция main() будет управлять игровым процессом – именно она управляет тем, куда пойдет змейка, обрабатывает столкновения с границами экрана, отвечает за поедание яблок и увеличение змейки (при поедании яблока добавляется еще один сегмент змейки).

```
def main():
    global IN_GAME

    if IN_GAME:
        # Двигаем змейку
        s.move()

        # Определяем координаты головы
        head_coords = c.coords(s.segments[-1].instance)
        x1, y1, x2, y2 = head_coords

        # Столкновение с границами экрана
        if x1 < 0 or x2 > WIDTH or y1 < 0 or y2 > HEIGHT:
            IN_GAME = False

        # Поедание яблок
        elif head_coords == c.coords(BLOCK):
            s.add_segment()
            c.delete(BLOCK)
            c.create_block()

        # Змейка съела саму себя
        else:
            # Проходим по всем сегментам змеи
            for index in range(len(s.segments)-1):
                if c.coords(s.segments[index].instance) == head_coords:
                    IN_GAME = False

    # Если не в игре выводим сообщение о проигрыше
    else:
          c.create_text(WIDTH/2, HEIGHT/2,
                        text="ТЫ ПРОИГРАЛ!",
                        font="Arial 20",
                        fill="#ff0000")
```

Чтобы все работало, как надо осталось добавить следующие операторы:

```
# Реагируем на нажатия клавиш
c.bind("<KeyPress>", s.change_direction)

create_block()
main()
```

Первый оператор – это реакция на нажатия клавиш. При нажатии клавиш мы изменяем направление змейки. Затем мы вызываем функцию create_block() для создания первого яблока. Далее яблоки (после их поедания) будет создавать функция main(). Наконец, мы запускаем функцию main() для запуска игры.

Все, нашу игру можно запускать. Далее показаны две иллюстрации процесса игры. На рис. 37.3 змейка несется к еде, а на рис. 37.4 было допущено столкновение с границами игрового поля, и игрок получил сообщение о том, что он проиграл.

Рис. 37.3. Процесс игры

Рис. 37.4. Игра закончена

37.7. Полный исходный код

У нас получилось довольно сложное приложение, которое не у всех начинающих программистов получится собрать воедино. Поэтому приводится полный исходный код, чтобы вы могли свериться с вашим вариантом. Комментарии частичные, поскольку они уже были приведены ранее.

Листинг 37.3. Полный исходный код Змейки

```
from tkinter import *
import random

# Некоторые глобальные переменные
WIDTH = 800
HEIGHT = 600
SEG_SIZE = 20
IN_GAME = True


# Вспомогательные функции
def create_block():
    """ Создает еду змейки - яблоко """
    global BLOCK
    posx = SEG_SIZE * random.randint(1, (WIDTH-SEG_SIZE) / SEG_SIZE)
    posy = SEG_SIZE * random.randint(1, (HEIGHT-SEG_SIZE) / SEG_SIZE)
    BLOCK = c.create_oval(posx, posy,
                          posx+SEG_SIZE, posy+SEG_SIZE,
                          fill="red")


def main():
    """ Управляет игровым процессом """
    global IN_GAME
    if IN_GAME:
        s.move()
        # Определяем координаты головы
        head_coords = c.coords(s.segments[-1].instance)
        x1, y1, x2, y2 = head_coords
        # Столкновения с краями игрового поля
        if x2 > WIDTH or x1 < 0 or y1 < 0 or y2 > HEIGHT:
            IN_GAME = False
        # Поедание яблока
        elif head_coords == c.coords(BLOCK):
            s.add_segment()
            c.delete(BLOCK)
            create_block()
        # Поедание змейки
```

```
        else:
            for index in range(len(s.segments)-1):
                if head_coords == c.coords(s.segments[index].instance):
                    IN_GAME = False
        root.after(100, main)
    # Не IN_GAME -> выводм сообщение
    else:
        c.create_text(WIDTH/2, HEIGHT/2,
                      text="ТЫ ПРОИГРАЛ!",
                      font="Arial 20",
                      fill="red")


class Segment(object):
    """ Сегмент змейки """
    def __init__(self, x, y):
        self.instance = c.create_rectangle(x, y,
                                           x+SEG_SIZE, y+SEG_SIZE,
                                           fill="white")


class Snake(object):
    """ Класс змейки """
    def __init__(self, segments):
        self.segments = segments
        # Возможные направления движения
        self.mapping = {"Down": (0, 1), "Right": (1, 0),
                        "Up": (0, -1), "Left": (-1, 0)}
        # Начальное направление
        self.vector = self.mapping["Down"]

    def move(self):
        """ Двигает змейку в заданном направлении """
        for index in range(len(self.segments)-1):
            segment = self.segments[index].instance
            x1, y1, x2, y2 = c.coords(self.segments[index+1].instance)
            c.coords(segment, x1, y1, x2, y2)

        x1, y1, x2, y2 = c.coords(self.segments[-2].instance)
        c.coords(self.segments[-1].instance,
                 x1+self.vector[0]*SEG_SIZE, y1+self.vector[1]*SEG_SIZE,
                 x2+self.vector[0]*SEG_SIZE, y2+self.vector[1]*SEG_SIZE)

    def add_segment(self):
        """ Добавляет сегмент змейке """
        last_seg = c.coords(self.segments[0].instance)
        x = last_seg[2] - SEG_SIZE
        y = last_seg[3] - SEG_SIZE
        self.segments.insert(0, Segment(x, y))
```

```
    def change_direction(self, event):
        """ Изменяет направление движения змейки  """
        if event.keysym in self.mapping:
            self.vector = self.mapping[event.keysym]

# Настройка окна
root = Tk()
root.title("Змейка")


c = Canvas(root, width=WIDTH, height=HEIGHT, bg="#005500")
c.grid()
# захватываем фокус
c.focus_set()
# создаем сегменты и змейку
segments = [Segment(SEG_SIZE, SEG_SIZE),
            Segment(SEG_SIZE*2, SEG_SIZE),
            Segment(SEG_SIZE*3, SEG_SIZE)]
s = Snake(segments)
# Реагируем на нажатия клавиш
c.bind("<KeyPress>", s.change_direction)

# Ставим первое яблоко на карту
create_block()
# Запускаем игру
main()
root.mainloop()
```

37.8. Как можно улучшить игру

Совершенству нет предела. Наша игра хороша, но ее можно улучшить. Пусть это будет вашим домашним заданием. Я предлагаю ввести в игру счет, чтобы между игроками (например, между членами семьи) стало возможным соперничество.

Нужно учитывать количество съеденных яблок. Добавьте глобальную переменную *apples* и в функции main() при поедании яблока увеличивайте значение этой переменной:

```
        # Поедание яблока
        elif head_coords == c.coords(BLOCK):
            s.add_segment()
            apples += 1
            c.delete(BLOCK)
            create_block()
```

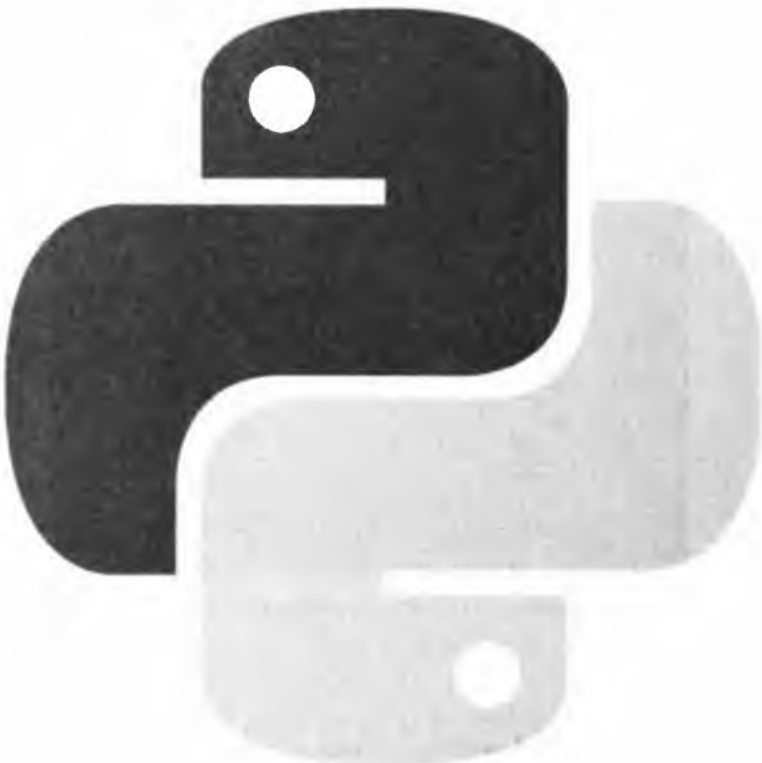
В случае проигрыша нужно сообщить игроку его счет:

```
t = "ТЫ ПРОИГРАЛ!\nСЧЕТ: " + str(apples)
c.create_text(WIDTH/2, HEIGHT/2,
              text=t,
              font="Arial 20",
              fill="red")
```

Должно получиться, как показано на рис. 37.5.

Рис. 37.5. В игру добавлен счет

Подсказка. Чтобы все работало, не забудьте добавить глобальную переменную *apples*, равную 0, и объявите ее как глобальную в функции main().



Глава 38.

РАБОТА С БАЗОЙ ДАННЫХ

В этой главе будет рассмотрен пример взаимодействия Python с базой данных MySQL. Мы не будем рассматривать синтаксис SQL – этому посвящены другие книги, как и не будем рассматривать принципы взаимодействия с базой данных как таковой.

38.1. Установка поддержки MySQL

Для работы с базой данных нужен модуль **mysql-connector-python**, который мы можем установить командой:

```
pip3 install mysql-connector-python
```

38.2. Подключение к MySQL-серверу

Для подключения к серверу используется следующий код:

```
# подключаем нужные библиотеки
import mysql.connector
```

```
# подключаемся
dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="password"
)

print(dataBase)

# отключаемся
dataBase.close()
```

Вывод фрагмента будет такой:

```
<mysql.connector.connection_cext.CMySQLConnection object at 0x7f73f0191d00>
```

Если объект не удалось создать, проверьте параметры подключения к БД – имя сервера, имя пользователя, пароль.

38.3. Создание базы данных и таблицы

Сразу нужно отметить, что не у каждого пользователя MySQL есть привилегии для создания базы данных. Чтобы создать базу данных средствами Python, вам нужно подключиться к MySQL-серверу, используя учетные данные пользователя с соответствующими правами.

Непосредственно для создания базы данных используется запрос (DATABASE_NAME замените на название базы данных):

```
CREATE DATABASE DATABASE_NAME
```

Пример кода на Python:

```
# importing required libraries
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
```

```
  user ="user",
  passwd ="password"
)

# подготавливаем объект
cursorObject = dataBase.cursor()

# создаем БД
cursorObject.execute("CREATE DATABASE test2")
```

Для создания таблицы используется следующий оператор:

```
CREATE TABLE
(
    column_name_1 column_Data_type,
    column_name_2 column_Data_type,
    :
    :
    column_name_n column_Data_type
);
```

Код на Python для выполнения этого оператора будет выглядеть так:

```
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="password",
  database = "test2"
)

# preparing a cursor object
cursorObject = dataBase.cursor()

# creating table
studentRecord = """CREATE TABLE STUDENT (
                   NAME  VARCHAR(20) NOT NULL,
                   BRANCH VARCHAR(50),
                   ROLL INT NOT NULL,
                   SECTION VARCHAR(5),
                   AGE INT
                   )"""
```

```
# table created
cursorObject.execute(studentRecord)

# отключаемся
dataBase.close()
```

Обратите внимание, что при подключении к серверу мы уже указываем имя базы данных – test2. Обязательно это нужно сделать, иначе создать таблицу без выбора БД не получится!

> **Примечание.** Имя *test* использовать нежелательно, так как такая база данных может быть создана по умолчанию. В то же время ее может и не быть, если администратор удалил ее. Прежде, чем выполнять код, убедитесь, что база данных существует!

38.4. Вставка записи в базу данных

Для вставки записи используется оператор INSERT, общий синтаксис которого выглядит так:

```
INSERT INTO table_name (column_names) VALUES (data)
```

Рассмотрим код на Python для вставки записи в таблицу:

```
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="password",
  database = "test2"
)

cursorObject = dataBase.cursor()

sql = "INSERT INTO STUDENT (NAME, BRANCH, ROLL, SECTION, AGE)\
VALUES (%s, %s, %s, %s, %s)"
val = ("Den", "CSE", "85", "B", "19")

cursorObject.execute(sql, val)
dataBase.commit()
```

```
    dataBase.close()
```

Для вставки несколько строк в таблицу можем использовать код:

```
sql = "INSERT INTO STUDENT (NAME, BRANCH, ROLL, SECTION, AGE)\
VALUES (%s, %s, %s, %s, %s)"
val = [("Den", "CSE", "98", "A", "19"),
       ("Mark", "CSE", "99", "A", "18"),
       ("Andrew", "MAE", "43", "B", "20"),
       ("Lera", "CSE", "95", "A", "19")]

cursorObject.executemany(sql, val)
dataBase.commit()
```

38.5. Получение данных

Теперь, когда мы добавили данные, разберемся как их получить. Для получения данных используется оператор SELECT. Обычно синтаксис этого оператора выглядит так:

```
SELECT * FROM table_name
```

Пример кода:

```
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="password",
  database = "test2"
)

cursorObject = dataBase.cursor()

query = "SELECT NAME, ROLL FROM STUDENT"
cursorObject.execute(query)

myresult = cursorObject.fetchall()

for x in myresult:
    print(x)

# disconnecting from server
dataBase.close()
```

Здесь важное отличие от всех предыдущих операторов. Ранее мы отдавали инструкции серверу, которые не возвращали каких-либо данных. Сейчас же сервер возвращает нам данные, которые нам нужно прочитать. Результат записывается в переменную *myresult*, далее мы проходимся по результату в цикле *for*. Вывод будет таким:

```
('Den', 98)
('Mark', 99)
...
```

Посредством WHERE мы можем уточнить результаты, отфильтровав лишнее. Например, вот так можно получить информацию о студентах старше 19 лет:

```
cursorObject = dataBase.cursor()

query = "SELECT * FROM STUDENT where AGE >=19"
cursorObject.execute(query)

myresult = cursorObject.fetchall()

for x in myresult:
    print(x)
```

38.6. Обновление записи

Для обновления записи в БД используется SQL-оператор UPDATE, синтаксис которого следующий:

```
UPDATE tablename
SET ="new value"
WHERE ="old value";
```

Ранее мы ошиблись с возрастом студента Mark, на самом деле ему 22 года. Исправим это так:

```
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
```

```
  passwd ="password",
  database = "test2"
)

cursorObject = dataBase.cursor()

query = "UPDATE STUDENT SET AGE = 22 WHERE Name ='Mark'"
cursorObject.execute(query)
dataBase.commit()

dataBase.close()
```

38.7. Удаление записи

Для удаления записей из БД используется оператор DELETE:

```
DELETE FROM TABLE_NAME
WHERE ATTRIBUTE_NAME = ATTRIBUTE_VALUE
```

Код на Python для удаления студента Anatoly:

```
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="password",
  database = "test2"
)

cursorObject = dataBase.cursor()

query = "DELETE FROM STUDENT WHERE NAME = 'Anatoly'"
cursorObject.execute(query)
dataBase.commit()

dataBase.close()
```

Мы рассмотрели полный цикл работы с базой данных – создание базы данных, создание и заполнение таблицы, выборка данных, обновление и удаление записей. Работа с остальными SQL-операторами осуществляется аналогичным образом.

Приложение 1.

СРЕДСТВА ШИФРОВАНИЯ В PYTHON

В Python не так уж много инструментов стандартной библиотеки, которые работают с шифрованием. Однако в нашем распоряжении есть библиотеки хеширования. В данном приложении мы сконцентрируемся на следующих библиотеках: PyCrypto и cryptography. Мы научимся шифровать и расшифровывать строки при помощи двух этих библиотек.

П. 1. Хеширование

Если вам нужно пароли, то для этого прекрасно подойдет модуль стандартной библиотеки Python **hashlib**. Он включает в себя безопасные алгоритмы хеширования FIPS, такие как SHA1, SHA224, SHA256, SHA384, а также SHA512 и MD5. Python также поддерживает функции хеширования adler32 и crc32, но они содержатся в модуле **zlib**.

Одно из самых популярных применений хеширования – это хранение хеша пароля, вместо самого пароля. Конечно, хеш должен быть хорошим, в противном случае он может быть расшифрован. Другой популярный случай, в котором применяется хеширование – это хеширование файла, с последующей отправкой файла и его хеша по отдельности. Получатель файла может запустить хеш в файле, чтобы убедиться в том, что файл соответствует отправленному хешу. Если это так, значит никто не менял файл, когда он был отправлен. Давайте попробуем создать хеш md5. Но оказывается, чтобы использовать хеш md5, нужно передать его строке байта, вместо обычной. Так что мы попробовали сделать это, после чего вызвали метод дайджеста, чтобы получить наш хеш. Если вы предпочитаете хешированный дайджест, мы можем сделать и это:

```
import hashlib
hashlib.md5.update(b'Hello!')
result = hashlib.md5.digest()

print(result)
```

Сначала мы импортировали модуль **hashlib** и создали экземпляр объекта md5 HASH. Далее, мы вписали небольшой текст в объект хеша и получили трассировку.

На самом деле существует метод быстрого создания хеша, мы рассмотрим его, когда создадим наш хеш sha512:

```
import hashlib
sha = hashlib.sha1(b'Hello').hexdigest()
print(sha)
```

Как вы видите, мы создали наш экземпляр хеша и вызвали его метод дайджеста одновременно. Далее, мы выводим наш хеш, чтобы на него посмотреть.

П. 2. PyCrypto

П. 2.1. Установка

Пакет PyCrypto, наверное, самый известный сторонний пакет криптографии для Python. К сожалению, его доработка остановилась в 2012 году. Однако он продолжается выпускаться под разные версии Python, вы можете получить PyCrypto даже для последних версий, если, конечно, вы не брезгуете использовать двоичный код стороннего производства. К примеру, я нашел несколько бинарных колес Python 3.5 для PyCrypto на Github (https://github.com/sfbahr/PyCrypto-Wheels). К счастью, есть продолжение проекта под названием PyCrytodome, которая является неплохой заменой PyCrypto. Для его установки на Linux вы можете использовать следующую команду:

```
pip install pycryptodome
```

В Windows команда будет другой:

```
pip install pycryptodomex
```

П. 2.2. Шифрование строки

Рассмотрим шифрование строки:

```
from Crypto.Cipher import DES

key = b'abcdefgh'          # Ключ
```

```
def pad(text):
    while len(text) % 8 != 0:
        text += b' '
    return text

des = DES.new(key, DES.MODE_ECB)
text = b'Python!'
padded_text = pad(text)

encrypted_text = des.encrypt(padded_text)
print(encrypted_text)
```

Этот код слегка запутанный, так что давайте уделим немного времени на его анализ. Обратите внимание на то, что размер ключа под шифровку DES — 8 байт, поэтому мы установили нашу переменную ключа в строку размера букв строки. Шифруемая нами строка должна быть кратна 8 в ширину, так что мы создаем функцию под названием *pad*, которая может заполнить любую строку пробелами, пока она не станет кратна 8. Далее мы создаем экземпляр DES и текст, который нам нужно зашифровать. Мы также создаем заполненную версию текста.

Давайте попытаемся зашифровать начальный, незаполненный вариант строки, что приведет нас к ошибке *ValueError*. Таким образом, Python ясно дает нам понять, что нам нужно использовать заполненную строку, так что мы передаем вторую версию. Как вы видите, мы получаем зашифрованную строку! Конечно, пример нельзя назвать полным, если мы не выясним, как расшифровать нашу строку:

```
data = des.decrypt(encrypted_text)
print(data)
```

К счастью, это очень легко сделать, так как все что нам нужно, это вызвать метод decrypt в нашем объекте des для получения расшифрованной байтовой строки.

П. 2.3. Шифрование файлов с помощью RSA

Наша следующая задача — научиться шифровать файлы и расшифровывать файлы при помощи RSA. Но для начала, нам нужно создать ключи RSA.

Если вам нужно зашифровать ваши данные при помощи RSA, тогда вам также нужно получить доступ к паре ключа RSA public / private, или сгенери-

ровать собственную. В данном примере мы генерируем собственную пару ключей. Так как это весьма легко, мы сделаем это в интерпретаторе Python:

```
from Crypto.PublicKey import RSA

code = 'nooneknows'
key = RSA.generate(2048)

encrypted_key = key.exportKey(
    passphrase=code,
    pkcs=8,
    protection="scryptAndAES128-CBC"
)

with open('my_private_rsa_key.bin', 'wb') as f:
    f.write(encrypted_key)

with open('my_rsa_public.pem', 'wb') as f:
    f.write(key.publickey().exportKey())
```

Сначала мы импортируем RSA из Crypto.PublicKey. Затем, мы создаем примитивный код доступа. Далее, мы генерируем ключ RSA на 2048 битов. Теперь мы подходим к интересной части. Для генерации приватного ключа, нам нужно вызвать метод *exportKey* нашего ключа RSA, и передать ему наш код доступа, который будет использован стандартом PKCS, чья схема шифровки будет использована для защиты нашего приватного ключа. После этого мы записываем файл на диск. Далее, мы создаем наш приватный ключ через метод *publickey* нашего ключа RSA. Мы использовали короткий путь в этой части кода, связав вызов метода *exportKey* с методом *publickey* для записи файла на диск.

Теперь у нас в распоряжении есть и приватный и публичный ключи, так что мы можем зашифровать кое-какие данные и вписать их в файл. Вот достаточно простой пример:

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP


with open('encrypted_data.bin', 'wb') as out_file:
    recipient_key = RSA.import_key(
        open('my_rsa_public.pem').read()
```

```
    )

    session_key = get_random_bytes(16)

    cipher_rsa = PKCS1_OAEP.new(recipient_key)
    out_file.write(cipher_rsa.encrypt(session_key))

    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    data = b'blah blah blah Python blah blah'
    ciphertext, tag = cipher_aes.encrypt_and_digest(data)

    out_file.write(cipher_aes.nonce)
    out_file.write(tag)
    out_file.write(ciphertext)
```

Первые три строки покрывают наши импорты из PyCryptodome. Далее мы открываем файл для записи. Далее, мы импортируем наш публичный ключ в переменной и создаем 16-битный ключ сессии. Для этого примера мы будем использовать гибридный метод шифрования, так что мы используем PKCS#1 OAEP (*Optimal asymmetric encryption padding*). Это позволяет нам записывать данные произвольной длины в файл. Далее, мы создаем наш шифр AES, создаем кое-какие данные и шифруем их. Это дает нам зашифрованный текст и MAC. Наконец, мы выписываем nonce, MAC (или тег), а также зашифрованный текст. К слову, nonce – это произвольное число, которое используется только в криптографических связях. Обычно это случайные или псевдослучайные числа. Для AES, оно должно быть минимум 16 байтов в ширину. Вы вольны попытаться открыть зашифрованный файл в своем текстовом редакторе. Вы увидите только какое-то безобразие.

Теперь попробуем расшифровать наши данные:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import AES, PKCS1_OAEP

code = 'nooneknows'

with open('encrypted_data.bin', 'rb') as fobj:
    private_key = RSA.import_key(
        open('my_rsa_key.pem').read(),
        passphrase=code
    )

    enc_session_key, nonce, tag, ciphertext = [
        fobj.read(x) for x in (private_key.size_in_bytes(), 16, 16, -1)
```

```
    ]

    cipher_rsa = PKCS1_OAEP.new(private_key)
    session_key = cipher_rsa.decrypt(enc_session_key)

    cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
    data = cipher_aes.decrypt_and_verify(ciphertext, tag)

print(data)
```

Если вы разобрались с предыдущим примером, то этот код должен быть весьма простым для разбора. В данном случае, мы открываем наш зашифрованный файл для чтения в бинарном режиме. Далее, мы импортируем наш приватный ключ. Обратите внимание на то, что когда вы импортируете приватный ключ, вы должны передать ему код доступа. В противном случае возникнет ошибка. Далее мы считываем наш файл.

Сначала мы считываем приватный ключ, затем 16 байтов для nonce, за которыми следуют 16 байтов, которые являются тегом, и наконец, остальную часть файла, который и является нашими данными.

Далее нам нужно расшифровать наш ключ сессии, пересоздать наш ключ AES и расшифровать данные. Вы можете использовать PyCryptodome в намного более широком ряде случаев. Однако нам нужно идти дальше и посмотреть, что еще мы можем сделать для наших криптографических нужд в Python.

П.3. Пакет cryptography

Пакет **cryptography** нацелен на то, чтобы быть "криптографом" для людей. Суть в том, что вам нужно разработать простые криптографические рецепты, которые и безопасны, и простые в использовании. Если нужно, вы можете перейти к низкоуровневым криптографическим примитивам, для которых требуется лишь знать, что вы делаете, в противном случае вы создадите что-то явно бесполезное в контексте защиты. Если вы работаете, например, в Python 3.5 Windows, вы можете установить этот пакет при помощи *pip* следующим образом:

```
pip install cryptography
```

Вы увидите, что cryptography установится совместно с несколькими зависимостями. Предположим, что с установкой все прошло чисто, и мы можем зашифровать какой-нибудь текст. Давайте используем для этого модуль **Fernet**.

Модуль Fernet реализует простую в использовании схему аутентификации, которая использует симметричный алгоритм шифрования, который гарантирует, что каждое зашифрованное в нем сообщение не может быть использовано или прочитано без определенного вами ключа. Модуль Fernet также поддерживает ключ ротации через MultiFernet. Давайте взглянем на простой пример:

```
from cryptography.fernet import Fernet

cipher_key = Fernet.generate_key()
print(cipher_key) # APM1JDVgT8WDGOWBgQv6EIhvxl4vDYvUnVdg-Vjdt0o=

from cryptography.fernet import Fernet

cipher = Fernet(cipher_key)
text = b'My message'
encrypted_text = cipher.encrypt(text)

print(encrypted_text)
```

Расшифровка:

```
decrypted_text = cipher.decrypt(encrypted_text)
print(decrypted_text) # 'My message'
```

Для начала, нам нужно импортировать Fernet. Затем мы генерируем ключ. Мы выводим ключ, чтобы увидеть, как он выглядит. Как вы видите, это случайная строка байтов. Если хотите, вы можете попробовать запустить метод *generate_key* несколько раз. Результат каждый раз новый. Далее мы создаем экземпляр нашего шифра Fernet при помощи нашего ключа. Теперь у нас есть шифр, который мы можем использовать для шифрования и расшифровки нашего сообщения. Следующий шаг, это создание сообщения с последующей его шифровкой при помощи метода *encrypt*. Я пошел вперед и вывел наш зашифрованный текст так, чтобы вы увидели, что вы больше не можете его читать. Для расшифровки нашего сообщения, мы просто вызовем метод *decrypt* в нашем шифре и передадим зашифрованный текст. В результате мы получим расшифрованную текстовую байтовую строку.

Книги по компьютерным технологиям, медицине, радиоэлектронике

Уважаемые авторы!

Приглашаем к сотрудничеству по изданию книг по **IT-технологиям, электронике, медицине, педагогике.**

Издательство существует в книжном пространстве более 20 лет и имеет большой практический опыт.

Наши преимущества:

- Большие тиражи (в сравнении с аналогичными изданиями других издательств);
- Наши книги регулярно переиздаются, а автор автоматически получает гонорар с *каждого* издания;
- Индивидуальный подход в работе с каждым автором;
- Лучшее соотношение цена-качество, влияющее на объёмы и сроки продаж, и, как следствие, на регулярные переиздания;
- Ваши книги будут представлены в крупнейших книжных магазинах РФ и ближнего зарубежья, библиотеках вузов, ссузов, а также на площадках ведущих маркетплейсов.

Ждем Ваши предложения:

тел. **(812) 412-70-26** / эл. почта: **nitmail@nit.com.ru**

Будем рады сотрудничеству!

Для заказа книг:

- **интернет-магазин: www.nit.com.ru / БЕЗ ПРЕДОПЛАТЫ по ОПТОВЫМ ценам**
  - более 3000 пунктов выдачи на территории РФ, доставка 3-5 дней
  - более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка 1-2 дня
  - тел. (812) 412-70-26
  - эл. почта: nitmail@nit.com.ru

- **магазин издательства:** г. Санкт-Петербург, пр. Обуховской обороны, д.107
  - метро Елизаровская, 200 м за ДК им. Крупской
  - ежедневно с 10.00 до 18.30
  - справки и заказ: тел. (812) 412-70-26

- **крупнейшие книжные сети и магазины страны**

  Сеть магазинов «Новый книжный» тел. (495) 937-85-81, (499) 177-22-11

- **маркетплейсы ОЗОН, Wildberries, Яндекс.Маркет, Myshop и др.**

Кольцов Дмитрий Михайлович

PYTHON

СОЗДАЕМ ПРОГРАММЫ И ИГРЫ

3-Е ИЗДАНИЕ

Группа подготовки издаиии:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО "Издательство Наука и Техника"

ОГРН 1217800116247, ИНН 7811763020, КПП 781101001

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107, лит. Б, пом. 1-Н

Подписано в печать 06.05.2022. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 27 п. л.

Тираж 2000. Заказ 3965.

Отпечатано ООО «Принт-М»
142300, Московская область,
г. Чехов, ул. Полиграфистов, дом 1

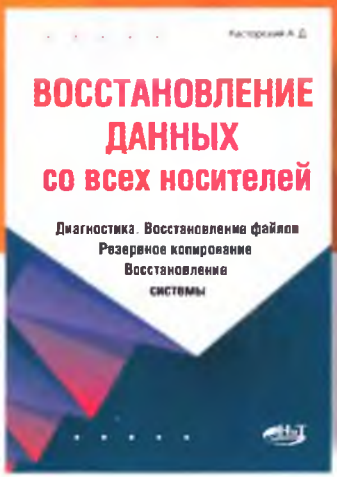
Кольцов Д. М.

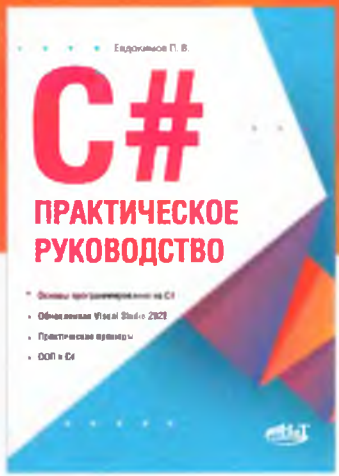
Python 3-е издание

Создаем программы и игры

Данная книга позволяет уже с первых шагов создавать свои программы на языке Python. Акцент сделан на написании компьютерных игр и небольших приложений. Для удобства начинающих пользователей в книге есть краткий вводный курс в основы языка, который поможет лучше ориентироваться на практике. По ходу изложения даются все необходимые пояснения, приводятся примеры, а все листинги (коды программ) сопровождаются подробными комментариями.

«Издательство Наука и Техника» рекомендует:





ISBN 978-5-907592-01-8

«Издательство Наука и Техника» г. Санкт-Петербург
Для заказа книг: т. (812) 412-70-26
E-mail: nitmail@nit.com.ru
Сайт: nit.com.ru

НиТ
nit.com.ru