
ПРОФЕССИЯ
Ваш лучший наставник
Широкий спектр тем: от основ Python до продвинутых концепций
Создание игр, приложений, графических интерфейсов и рабочих скриптов
Практические задания и пошаговые примеры с разбором кода
ПРОФЕССИЯ
Python-разработчик
А. Адонин
НиТ

Адонин А. М.

Профессия: Python-разработчик

"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973

Адонин А. М.

Профессия: Python-разработчик — СПб.: Издательство Наука и Техника, 2025. — 512 с., ил.

Серия "Профессия"

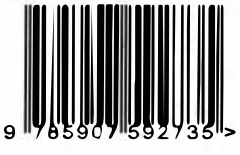
Книга позволит вам получить знания, достаточные для профессии **Python-разработчик** (начиная с уровня Junior Developer). Это даст вам возможность устройства на работу программистом, или возможность работать на себя – искать заказы на фриланс-биржах, создавать свои собственные приложения и игры, а затем выпускать их на таких площадках как Google Play, Steam и подобных.

Шаг за шагом вы познакомитесь с основными принципами Python, научитесь писать первые программы, работать с данными, создавать графические интерфейсы, разрабатывать свои приложения, писать рабочие скрипты по созданию страниц регистрации, банковских приложений и других направлений разработки. Вы также изучите принципы тестирования и отладки кода, что сделает ваши программы более надёжными. В заключительной части книги вы освоите продвинутые темы: многопоточность, асинхронное программирование, работу с модулями и пакетами, а также основы взаимодействия с реляционными, SQL и NoSQL базами данных.

В книге большое внимание уделяется практическим навыкам – каждая глава содержит примеры с разбором кода и различные задачи, которые помогут закрепить материал на практике.

Благодаря такому комплексному подходу, книга будет полезна как начинающим программистам, делающим первые шаги в Python, так и более опытным разработчикам, желающим углубить свои знания и освоить новые области применения языка.

ISBN 978-5-907592-73-5



Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

Содержание

Введение

Приветствую всех, в данной книге мы на различных примерах (как более легких, так и посложнее), познакомимся с языком программирования Python. Это позволит относительно легко понять не только его основы, но и более сложные инструменты и темы. Python можно сравнить с конструктором LEGO, где из разных элементов и блоков мы собираем полноценные программы.

"Пайтон" является очень гибким языком программирования (ЯП), что позволяет создавать самые разные вещи: от простых скриптов для автоматизации рутинных задач до сложных веб-приложений и игр. Он является одним из самых популярных ЯП, так как имеет простой синтаксис, похожий на обычный английский язык. Благодаря этому код на нем легко читаем и понятен даже новичкам.

Python был создан в конце 1980-х годов голландским программистом Гвидо ван Россумом. Он назвал язык в честь любимого комедийного шоу "Летающий цирк Монти Пайтона". Гвидо хотел создать ЯП, который был бы не только мощным, но и приятным в использовании. И, по мнению многих, ему это удалось.

Настоятельно рекомендуется переписывать весь предложенный код и прорабатывать все представленные примеры и задания. Так как голая теория бесполезна без практики. Все, что вы здесь прочитаете, забудется, если не применять эти знания на постоянной основе. Кроме того, самостоятельное написание кода, а не копирование его из других источников, позволяет решить проблему "чистого листа": когда при создании новой программы не знаешь, с чего начать.

Глава 1.

Установка Python и редактора кода

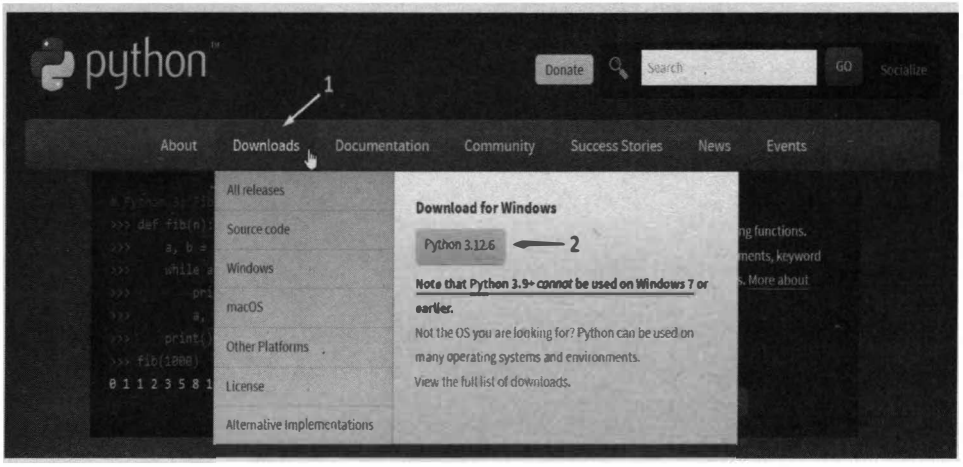
1.1. Установка интерпретатора Python

Прежде чем приступить к написанию программ, необходимо установить интерпретатор Python.

Интерпретатор — это специальная программа, которая берет наш код и выполняет его. Интерпретатор можно сравнить с переводчиком, который переводит инструкции на язык, понятный компьютеру.

То есть мы создаем текстовый файл с командами на языке Python. Интерпретатор последовательно читает каждую строчку кода, анализирует ее и выполняет соответствующие действия. Чтобы установить интерпретатор Python, необходимо посетить официальный сайт *python.org* и скачать последнюю версию дистрибутива.

По умолчанию представлена версия для Windows, но в разделе загрузок можно найти версии для других операционных систем, таких как Mac и Linux. Стоит обратить внимание, что актуальная версия интерпретатора совместима с версией Windows 8 и выше. Если ваша версия Windows 7 или XP, необходимо установить более ранние версии интерпретатора и работать с ними. Их также можно найти в разделе загрузок.



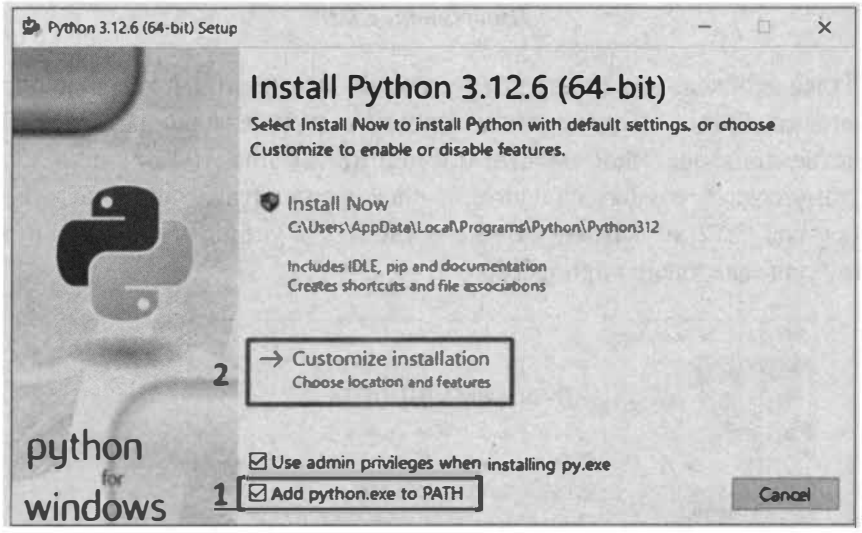
Изображение 1.1.

1.2. Установка Python

После скачивания запустите установщик двойным левым кликом мыши и поставьте галочку на пункте *Add python.exe to PATH*.

Эта настройка позволяет в командной строке запускать интерпретатор Python без необходимости прописывать полный путь к файлу.

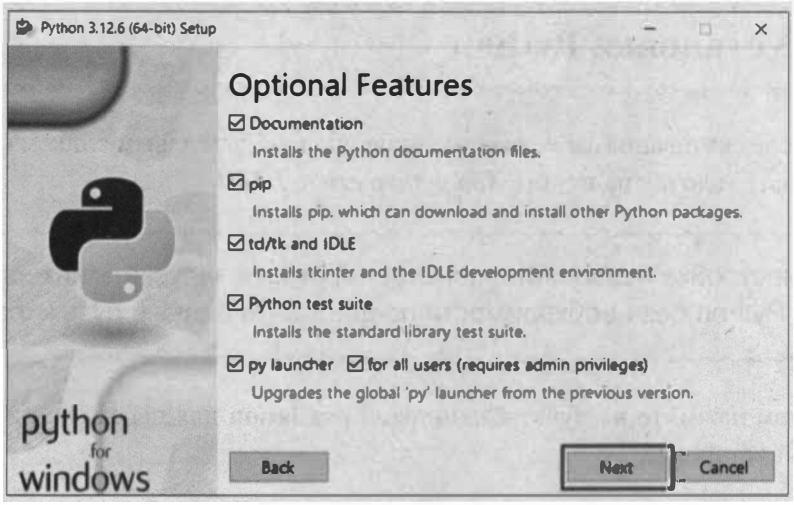
Затем нажмите на пункт **Customize installation** для перехода к настраиваемой установке.



Изображение 1.2.

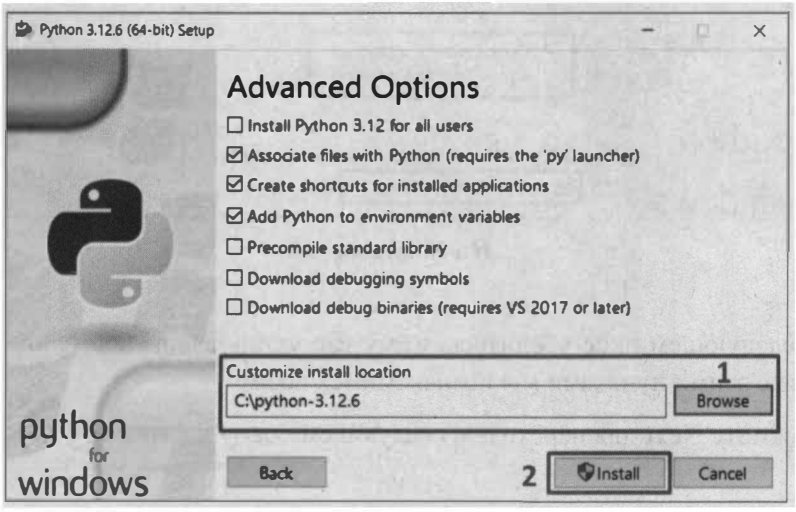
В следующем окне убедитесь, что у вас установлены все галочки, подробнее с этими пунктами мы познакомимся позже.

Нажмите **Next** для перехода к следующему окну настроек.



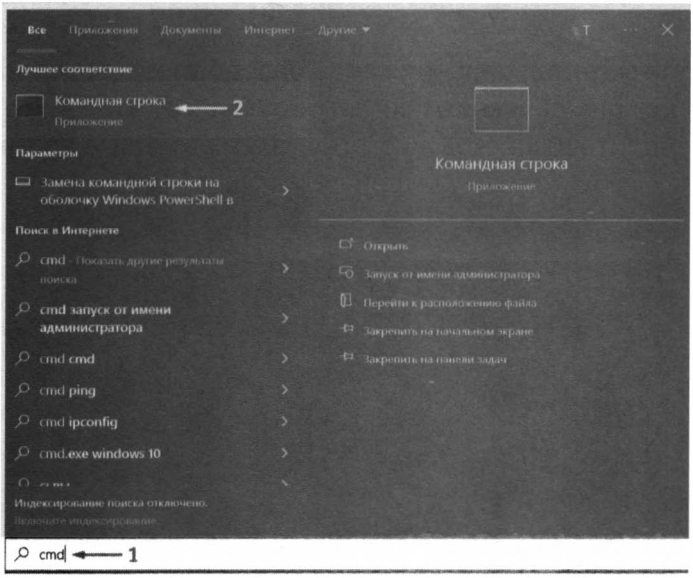
Изображение 1.3.

Далее необходимо указать оптимальную директорию, в которую следует установить Python. По умолчанию прописан путь к папке, которая у большинства пользователей скрыта, и к ней не так просто получить доступ. Поэтому создайте папку на одном из дисков с понятным названием, например python_3.12, и укажите ее в качестве места установки. Затем нажмите **Install** для завершения процесса.



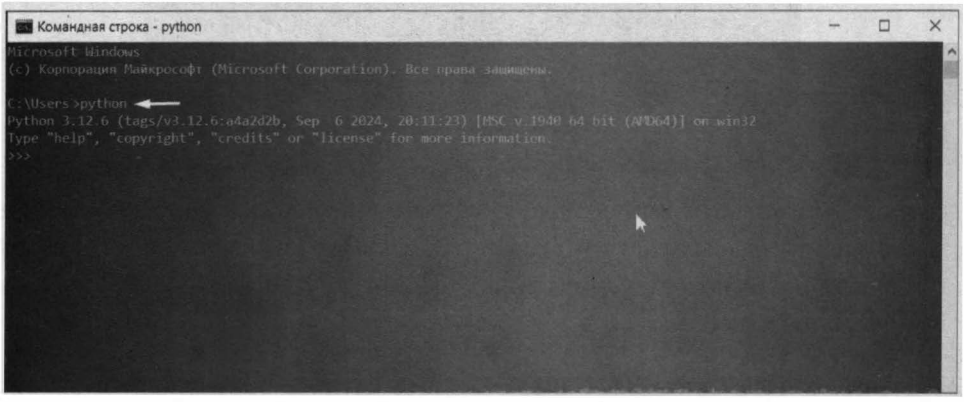
Изображение 1.4.

После установки мы можем убедиться, что интерпретатор нормально установился и работает. Для этого в поиске Windows введите команду *cmd* и запустите командную строку.



Изображение 1.5.

В открывшемся окне введите слово python и нажмите **Enter**. Система покажет нам параметры установленного "Питона".

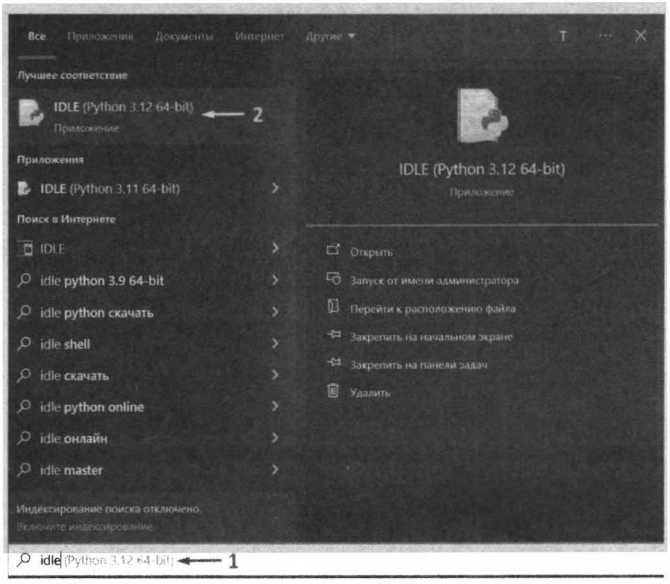


Изображение 1.6.

Для удобства работы с кодом в Python встроен редактор **IDLE** (Integrated Development and Learning Environment).

1.3. Редактор IDLE

IDLE – это интегрированная среда разработки, предназначенная в основном для обучения программированию. Чтобы открыть редактор, введите в поисковую строку ***idle*** и выберите соответствующий пункт.



Изображение 1.7.

Помимо встроенного редактора существует множество других редакторов кода. Каждый из них обладает своими уникальными особенностями и подходит для разных задач и стилей работы. Вот некоторые из самых популярных, часто использующихся для работы с Python и другими языками программирования:

- **Visual Studio Code (VS Code)** — один из самых популярных и быстро развивающихся редакторов. Отличается высокой производительностью, огромным количеством расширений и интеграцией с Git. Идеально подходит для Python благодаря наличию множества расширений, которые добавляют интеллектуальное автодополнение, отладку и другие полезные функции.

- **Atom** — еще один бесплатный и открытый редактор с большим количеством настроек и расширений. Интерфейс Atom напоминает интерфейс веб-приложений, что делает его удобным для многих пользователей.
- **Notepad++** — простой и легкий текстовый редактор для Windows, хорошо подходит для небольших проектов.
- **Sublime Text** — легкий и быстрый редактор с множеством настроек. Поддерживает большинство языков программирования и имеет активное сообщество разработчиков, создающее расширения.
- **Vim** — мощный текстовый редактор с командной строкой. Требует времени на освоение, но обладает огромными возможностями для настройки и автоматизации задач.
- **Emacs** — еще один мощный и настраиваемый текстовый редактор с богатой историей. Имеет множество режимов и функций, которые позволяют использовать его как полноценную среду разработки.
- **PyCharm** — специализированный IDE для Python. Предлагает широкий спектр функций, включая отладку, рефакторинг, интеграцию с различными инструментами и фреймворками. Имеет бесплатную версию Community и платную версию Professional с дополнительными возможностями.

Выбор редактора зависит от ваших индивидуальных предпочтений и задач. Если вы новичок, то IDLE или Visual Studio Code станут хорошим выбором. Для более опытных разработчиков подойдут PyCharm, Sublime Text или Vim. Во время обучения вы можете установить несколько разных редакторов и сравнить их на предмет удобства и функциональности.

1.4. Установка и настройка PyCharm

Ниже описана установка и настройка PyCharm, аналогично ему устанавливаются и другие редакторы. Перейдите на официальный сайт разработчика *jetbrains.com/pycharm* и скачайте актуальную версию дистрибутива. При нажатии кнопки **Download** будет открыта новая страница со ссылкой на платную версию. Прокрутите страницу вниз, чтобы найти бесплатную версию.

Изображение 1.8.

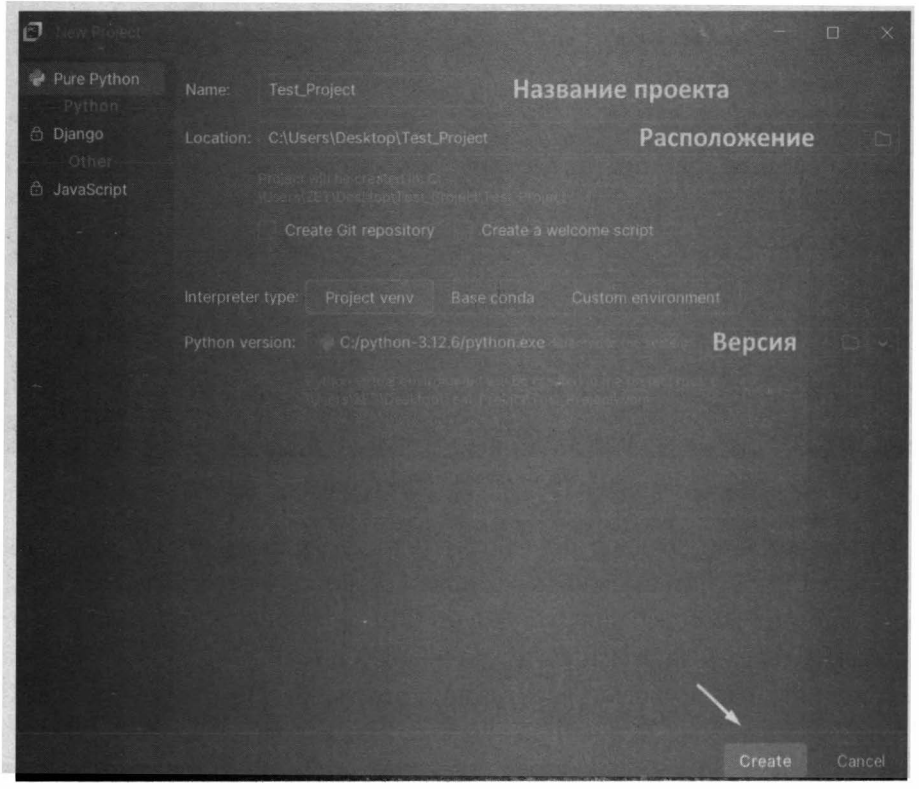
После скачивания установка происходит стандартным способом — двойным левым кликом мыши по exe-файлу. Выберите директорию установки и нажмите несколько раз клавишу **Next**.

При запуске нас встречает окно, в котором будет предложено пройти обучение. Рекомендуется сделать это, так как там описывается функционал для ежедневной работы.

Изображение 1.9.

Обучение проходит на английском языке, если вы его не знаете, ниже рассмотрим основные возможности. Для начала работы необходимо создать новый проект, задать ему понятное для вас название, выбрать директорию,

где будут храниться файлы, а также выбрать версию интерпретатора Python. Затем нажмите **Create** для завершения настроек.

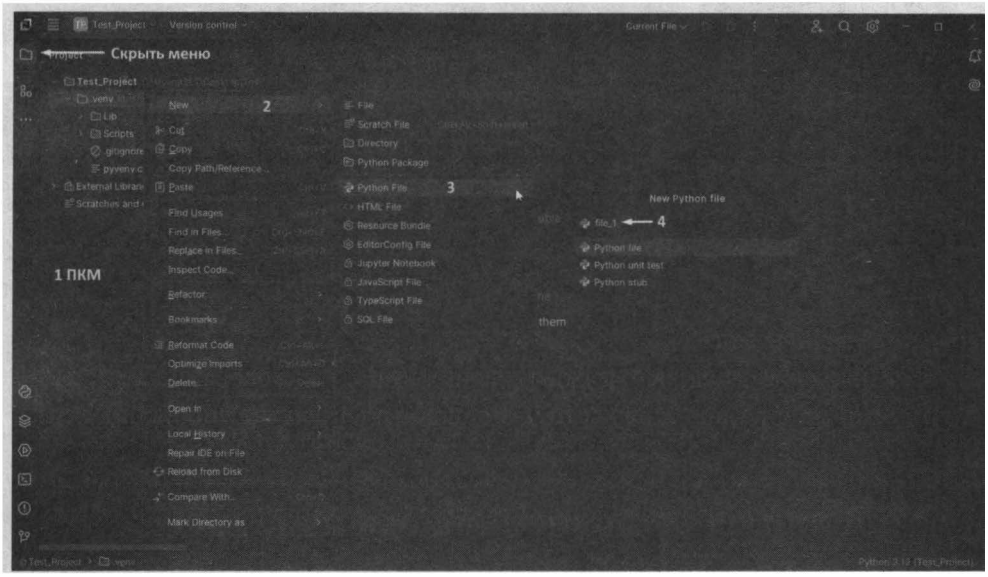


Изображение 1.10.

В открывшемся окне мы можем увидеть две области. Слева располагается меню со структурой файлов в нашем проекте, оно аналогично структуре папок в Windows. При нажатии на иконку папки меню можно скрывать и раскрывать.

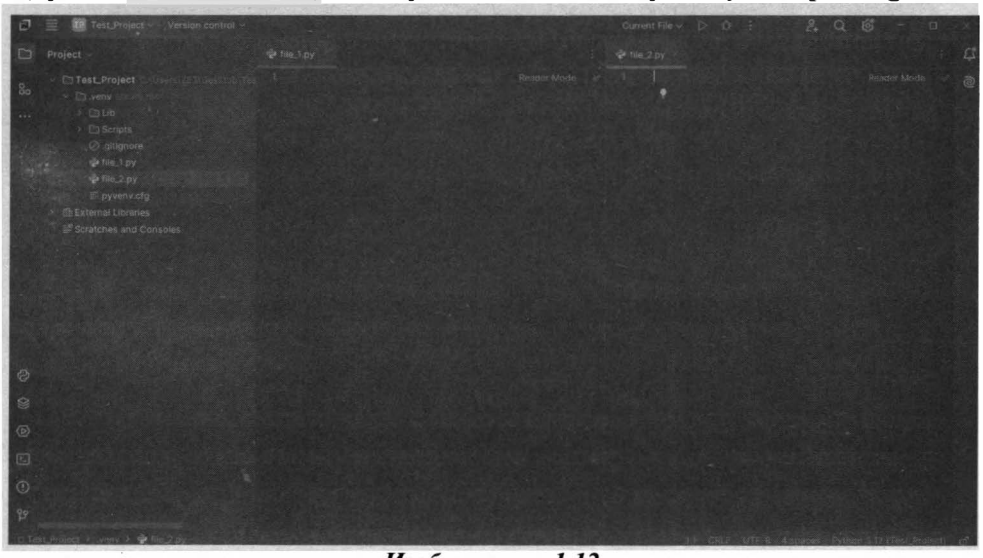
Чтобы создать новый файл, в котором мы будем писать код, нажмите правой кнопкой мыши по панели меню, выберите **New => Python File**. Далее откроется окошко, где нужно указать название файла.

Впишите любое название и нажмите **Enter**. В меню вы можете увидеть, что был создан файл с расширением ***.py***. Аналогичным образом создайте еще один файл.



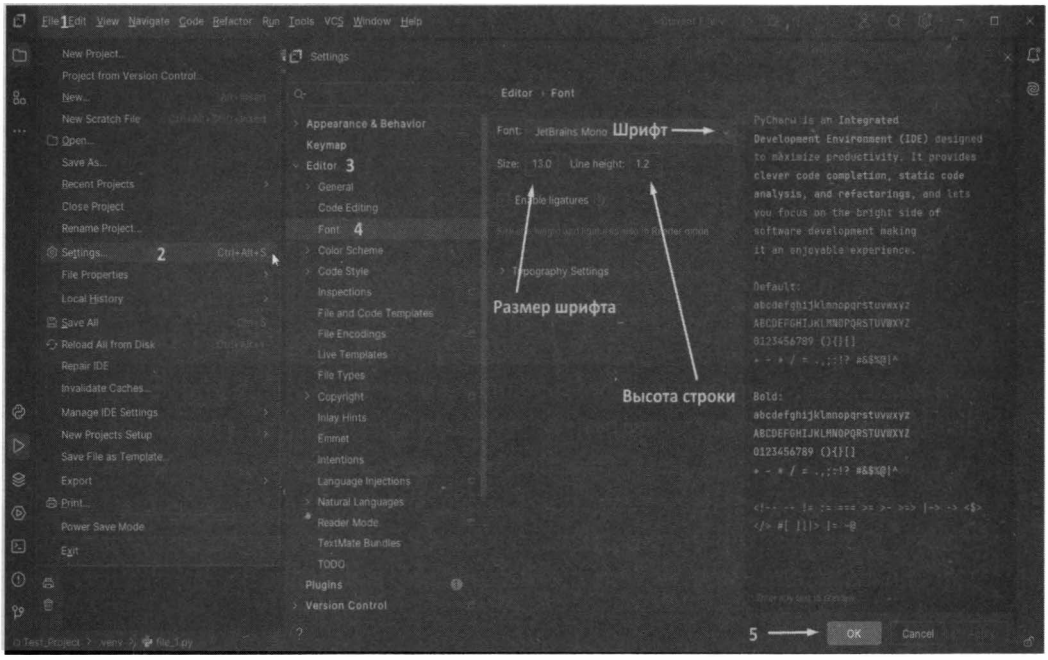
Изображение 1.11.

После создания файлов они автоматически открываются в отдельных вкладках. Между ними можно переключаться, а также перемещать их между собой для удобной очередности. Вместе с тем экран можно разделить, для отображения сразу двух файлов и одновременной работы с ними. Это может быть полезно, например, для анализа двух похожих скриптов. Нажмите ПКМ (правой кнопкой мыши) по второй вкладке и выберите пункт **Split Right**.



Изображение 1.12.

Для изменения размера шрифта или самого шрифта раскройте вкладку **File** на верхней панели (по умолчанию она может быть скрыта и доступна по нажатии иконки с четырьмя полосками). Перейдите в раздел **Setting => Editor => Font**.

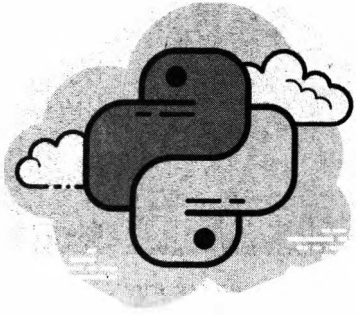


Изображение 1.13.

Также PyCharm хорош тем, что он подсвечивает допущенные нами ошибки и имеет функцию автодополнения. При вводе первого символа появляется всплывающая подсказка со всеми возможными инструкциями. Однако на этапе обучения лучше все писать самостоятельно.

Кроме того, при нажатии комбинации **Ctrl+Alt+L** редактор отформатирует ваш код согласно стандарту PEP 8. В данном стандарте регламентируются все правила написания программ на Python. То есть если вы пропустили пробел между символами или сделали неправильный отступ, при нажатии этой комбинации PyCharm наведет порядок.

Также большим преимуществом редактора является возможность отладки кода, но об этом мы поговорим подробнее в соответствующем разделе



Глава 2.

Основные принципы. Пишем первую простую программу

2.1. Что такое программа на Python?

Программа на **Python** — это набор инструкций, написанных на языке программирования Python, которые затем выполняет компьютер. Эти инструкции описывают последовательность действий, которые программа должна совершить для достижения определенной цели.

Это можно представить как рецепт для приготовления блюда. Рецепт содержит пошаговые инструкции, которые необходимо выполнить для приготовления конкретного блюда. Точно так же программа на Python содержит инструкции, которые компьютер должен выполнить, чтобы решить поставленную задачу.

2.2. Как выполняется программа?

Выполнение программ на **Python** — это процесс, в ходе которого компьютер преобразует написанный код в последовательность действий, которые он может понять и выполнить.

Всего можно выделить пять этапов:

1. Написание кода — мы создаем текстовый файл с расширением .py и пишем в нем свой код на языке Python, используя синтаксис и ключевые слова этого языка.

2. Запуск интерпретатора — далее мы запускаем интерпретатор Python из командной строки или среды разработки (IDLE, PyCharm и др.), указав путь к нашему файлу.

3. Компиляция в байт-код — перед выполнением кода интерпретатор Python обычно компилирует его в промежуточный формат, называемый байт-кодом. Байт-код — это низкоуровневая версия программы, которая ближе к машинному коду, но все еще может быть выполнена только интерпретатором Python. Компиляция в байт-код позволяет ускорить последующие запуски программы, так как его уже не нужно перекомпилировать каждый раз.

4. Выполнение байт-кода — интерпретатор последовательно выполняет инструкции байт-кода. Каждая инструкция выполняется в виртуальной машине Python (PVM), которая имитирует поведение реального компьютера.

5. Вывод результатов — если программа выводит какие-либо данные на экран или в файл, интерпретатор выполняет соответствующие инструкции для отображения этих данных.

Продолжая аналогию с рецептом, происходят следующие события:

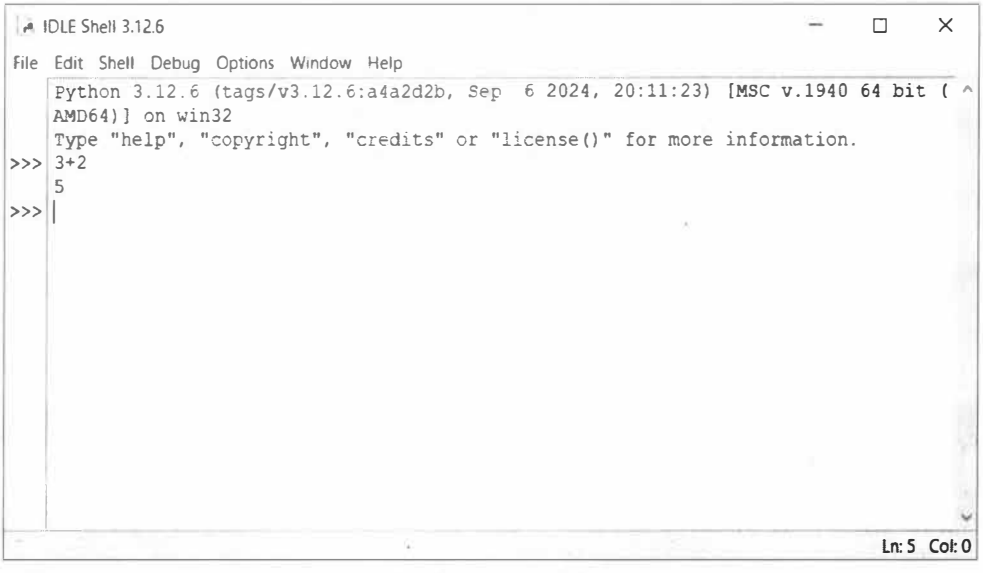
1. **Написание кода** — это создание рецепта.
2. **Интерпретатор** — это повар, который читает рецепт и выполняет указанные в нем действия.
3. **Байт-код** — это промежуточный этап приготовления, когда ингредиенты уже подготовлены и лежат в определенном порядке.
4. **Выполнение байт-кода** — это сам процесс приготовления блюда.
5. **Результат** — готовое блюдо в тарелке (результат работы программы на экране пользователя).

2.3. Варианты выполнения кода на Python

При работе с Python у нас есть два основных способа выполнения кода: интерактивный и файловый. Каждый из них имеет свои особенности и подходит для разных задач.

Интерактивный режим — в нем мы взаимодействуем с интерпретатором напрямую, вводя команды по одной строке и сразу получая результат. Это похоже на разговор с компьютером. Мы задаем вопрос, он сразу отвечает.

Напишите в поиске Windows аббревиатуру *idle* и откройте консоль, как это описывалось в предыдущей главе. Введите *3+2* и нажмите **Enter**. Мы тут же получим результат вычисления. То есть мы задали вопрос компьютеру: "Сколько будет 3+2?", и он сразу выдал ответ.



Изображение 2.1.

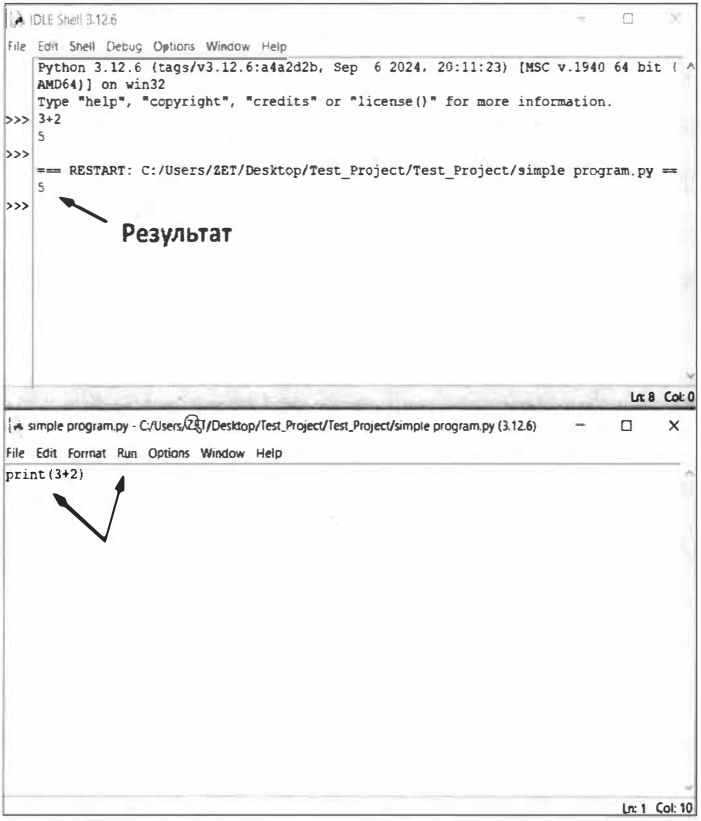
По сути, мы сейчас написали свою первую простейшую программу, которая указывает компьютеру прибавить к первому числу второе. Обратите внимание, что мы не создавали никаких файлов, поэтому наша "программа" и результат ее вычислений нигде не будут сохранены.

Интерактивный режим полезен для быстрого тестирования идей, обучения новым функциям и отладки небольших фрагментов кода. Его также можно использовать как мощный калькулятор для выполнения математических операций.

Файловый режим — в нем мы пишем весь код в текстовом файле с расширением .py и затем запускаем его. Это более структурированный вариант, который подходит для создания полноценных программ.

Создайте новый файл, нажав на верхней панели консоли **File => New file**. Откроется файловый редактор, в котором мы можем написать свою программу. Напишите *print(3+2)*, затем на верхней панели редактора нажмите **File => Save As**, чтобы сохранить, и выберите место на диске, где будет храниться файл. В таком случае наша программа может быть выполнена многократно, когда нам это понадобится.

Нажмите на верхней панели **Run => Run Module**, чтобы выполнить программу. Результат вычислений мы можем увидеть в консоли.

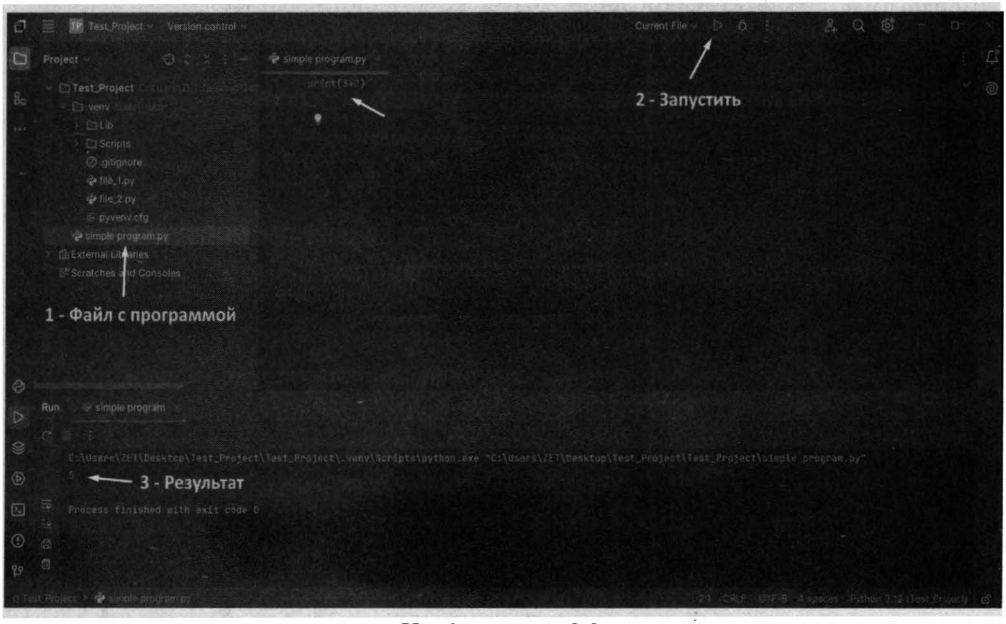


Изображение 2.2.

Файловый режим используется программистами на постоянной основе, так как это позволяет не только сохранять программу, но и передавать файл с ней другим людям, а также совместно работать над проектами. Кроме того, код в этом режиме лучше организован и понятен.

2.4. Запуск нашей первой программы в PyCharm

Создавать большие программы в среде IDLE может быть не очень удобно, поэтому многие разработчики используют такие среды разработки, как PyCharm и аналогичные. Для примера, вы можете запустить нашу первую программу в PyCharm. Откройте редактор и перетащите в него созданный файл. Затем нажмите на зеленый треугольник вверху редактора, похожий на кнопку **Play**, после этого внизу откроется окошко с результатом.



Изображение 2.3.

Глава 3.

Типы объектов и операции с ними

3.1. Объекты в Python

Объекты в **Python** — это фундаментальная концепция, которая лежит в основе всего в этом языке программирования. Представьте, что все в Python — это какие-то вещи, которые имеют свои свойства и умеют выполнять определенные действия. Эти "вещи" и есть объекты.

Для понимания, что такое объекты, можно сравнить их с реальными вещами. Например, автомобиль — это объект. У него есть свойства (цвет, модель, количество дверей) и возможности (ехать, тормозить, сигналить). В Python автомобиль также можно представить как объект, у которого будут свои атрибуты (цвет, модель) и методы (ехать, тормозить).

Атрибуты — это свойства объекта, которые описывают его состояние. Например, у объекта "автомобиль" атрибутами могут быть цвет, марка, год выпуска.

Методы — это действия, которые объект может выполнять. Например, у объекта "автомобиль" методы могут быть такие: ехать, тормозить, открывать дверь.

Такой подход делает код более гибким и позволяет использовать одни и те же механизмы для работы с различными типами данных. Код становится более понятным, когда мы представляем все элементы программы как объекты с определенными свойствами и поведением. Благодаря этому легко создавать собственные типы данных, определяя их атрибуты и методы.

Любые данные являются объектами и разделяются на типы:

Типы объектов*	Название типа объекта в Python	Пример объекта
Целое число	int (от слова «integer»)	43, -56, 0, 2548
Число с плавающей точкой (вещественные числа)	float	1.5, -43.0, 32.5
Строка	str (от слова «string»)	'hello', "Hello"
Логический тип	bool (от слова «boolean»)	True, False
Список	list	[43, -54.5, True, 'hello']
Словарь	dict (от слова «dictionary»)	{'a':123, 'b':456}
Кортеж	tuple	(1, 2, 3, 4, 5)
Множество	set	{1,2,3,4}

Изображение 3.1.

3.2. Операции с числами

Первые два типа объектов, с которыми мы познакомимся ближе, это числовые объекты **int** и **float**.

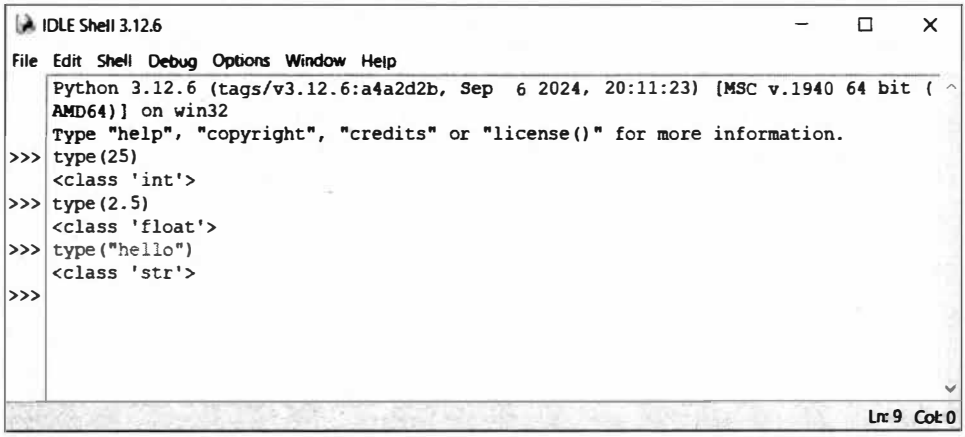
Тип **int** включает в себя целые числа без дробной части. Например: -2, 0, 100, 12345.

Тип **float** включает вещественные числа, то есть числа с дробной частью. Также их называют числами с плавающей точкой. Например: 3.14, -0.5, 2.71828. Обратите внимание, что дробная часть числа отделяется точкой. Если разделение будет через запятую, Python примет их за целые числа, написанные через запятую.

Для понимания, с каким типом объекта мы имеем дело, существует функция ***type***, которой мы можем передать объект, и она сообщит нам его тип.

- Откройте консоль IDLE и введите *type(25)*, затем нажмите **Enter**. Пайтон сообщит нам, что число 25 — это ***int***, то есть целое число.

- Введите в консоль *type(2.5)*, и на выходе получите вещественное число ***float***.
- Таким же образом вы можете ввести какое-нибудь слово, например *hello*. Пайтон сообщит нам, что это тип "строка" (str).



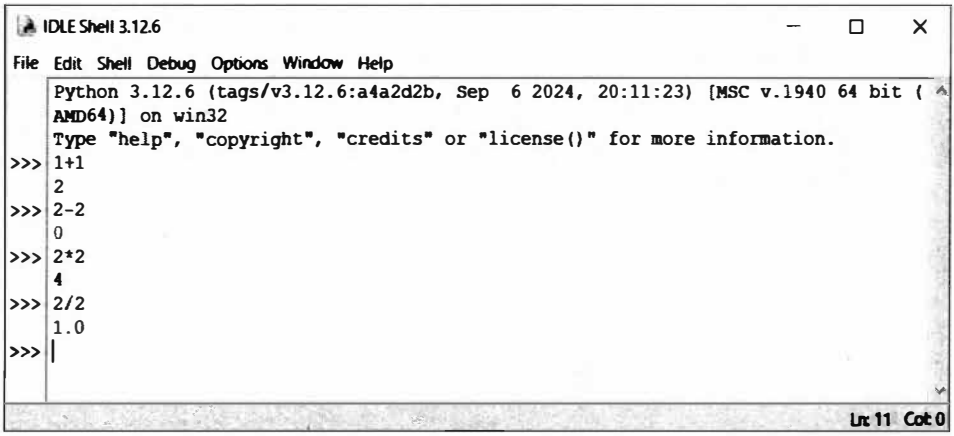
Изображение 3.2.

В программировании, как и в математике, над числами можно проводить арифметические операции.

Название операции	Используемый операнд
Сложение	+
Вычитание	-
Умножение	*
Деление	/
Возведение в степень	**
Целочисленное деление	//
Остаток от деления	%

Изображение 3.3.

Проведите в консоли простые операции сложения, вычитания, умножения и деления.

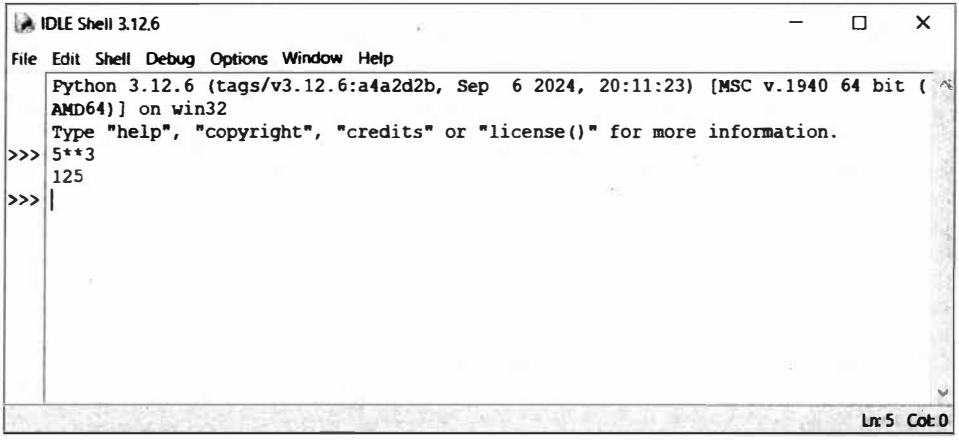


Изображение 3.4.

Обратите внимание, что в результате вычислений тип объекта может измениться. При сложении целых чисел (int) на выходе мы получим целое число. Если складывать целое число с дробным, всегда ответом будет дробное число (float).

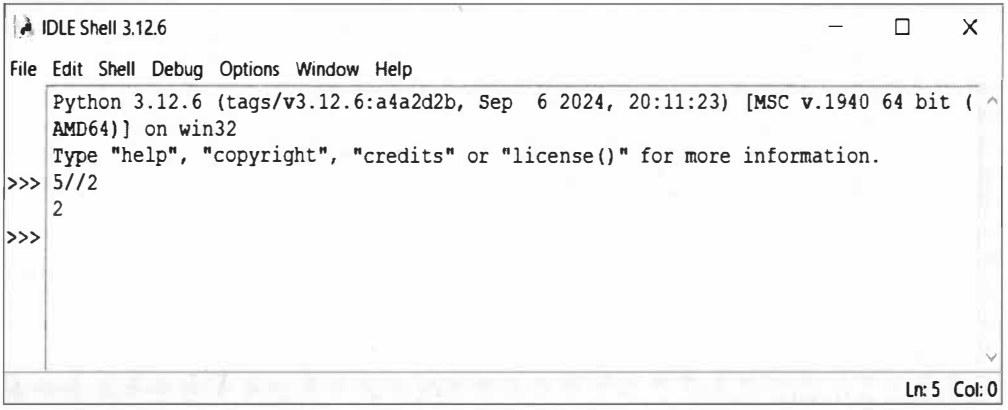
Также при делении, даже если оба числа являются целыми, ответом будет дробное. Это сделано потому, что в большинстве случаев при делении числа не делятся без остатка.

Чтобы возвести число в степень, используется двойной символ звездочки (**). Например, возведите 5 в степень 3. Для этого в консоли нужно написать: 5**3.



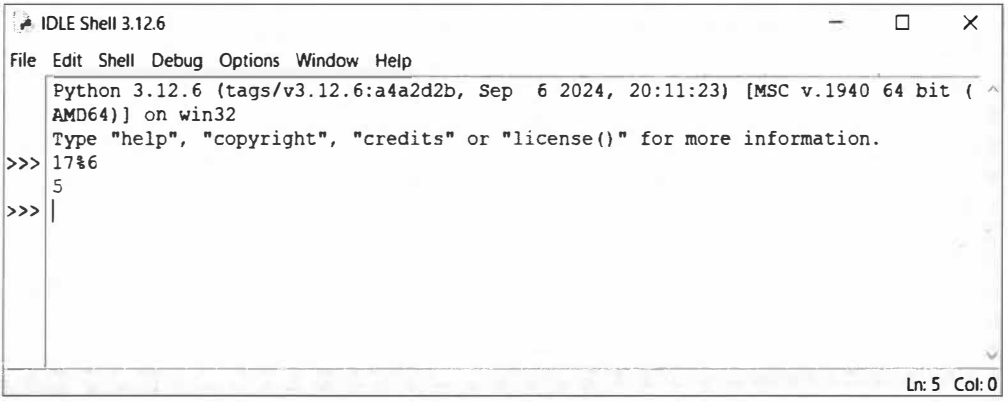
Изображение 3.5.

Целочисленное деление, или деление нацело, производится с помощью двойного слеша **(//)**. Деление без остатка всегда возвращает целое число (int). Например, введите в консоли 5//2. Ответом будет целое число 2, а остаток отбросится.



Изображение 3.6.

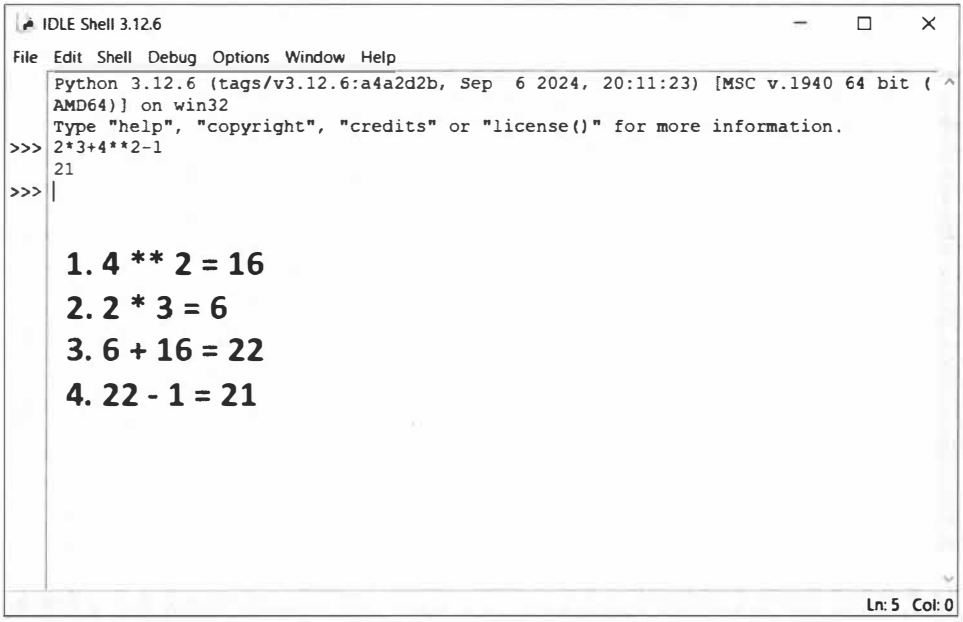
Остаток от деления использует символ процента **(%)** и в результате вычислений выводит в ответ только остаток. То есть, если мы разделим 17 на 6, ответом будет 5. В данном примере шестерка может дважды уместиться в семнадцати, а оставшаяся пятерка станет нашим ответом.



Изображение 3.7.

Так же, как и в математике, существует приоритет операций вычислений. Сначала происходит подсчет того, что указано в скобках, затем возведение в степень, далее умножение и деление, а в конце — вычитание и сложение:

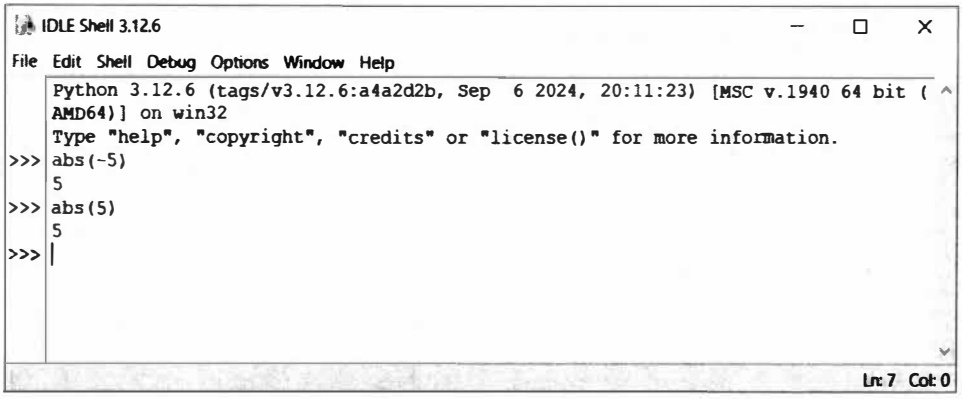
- **( )** — скобки (изменяют порядок вычислений);
- ****** — возведение в степень;
- *** / // %** — умножение, деление, целочисленное деление, остаток от деления;
- **+ -** — сложение, вычитание.



Изображение 3.8.

Python предоставляет множество встроенных функций, которые упрощают работу с числами. Их довольно много, поэтому рассмотрим самые основные и часто используемые.

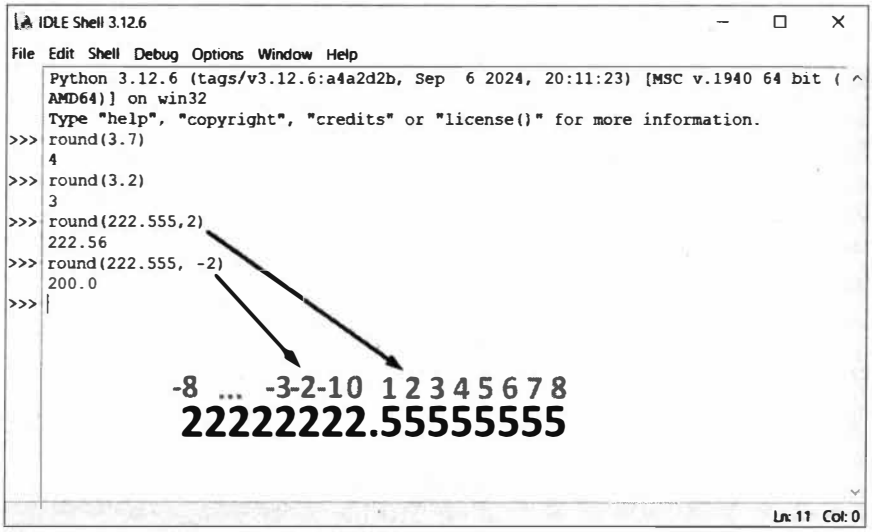
Функция **abs(x)** — возвращает абсолютное значение числа. Например, abs(-5) вернет 5. Данная функция может пригодиться, когда вводные данные неизвестны, но на выходе должно быть положительное число.



Изображение 3.9.

Функция round(x) — округляет число до ближайшего целого. Например, round(3.7) вернет 4, а round(3.2) вернет 3. Также можно округлить до десятых, сотых, тысячных и т.д. Для этого в скобках после запятой необходимо указать число, указывающее, сколько символов после запятой нужно оставить.

Например, round(222.555, 2) вернет 222,56. То есть число будет округлено до двух символов после запятой (до сотых). Если указать отрицательное значение, округление будет происходить до символов перед запятой. Например, round(222.555, -2) вернет 200.0.



Изображение 3.10.

Функция **int(x)** — преобразует число или строку в целое число. Например, int("42") вернет 42.

Функция **float(x)** — преобразует число или строку в число с плавающей точкой. Например, float("3.14") вернет 3.14.

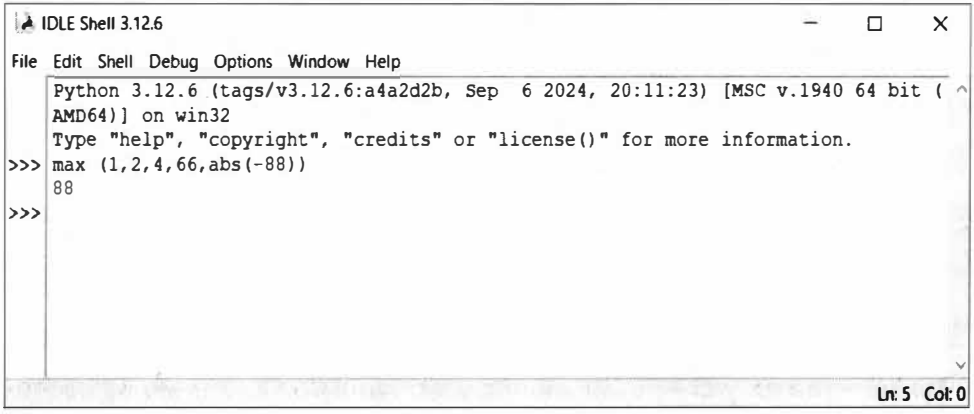
Функция **max(x,y,z)** — возвращает максимальное значение из последовательности чисел. Например, max(2, 3, 1) вернет 3.

Функция **min(x,y,z)** — возвращает минимальное значение из последовательности чисел. Например, min(2, 3, 1) вернет 1.

Функция **sum([x,y,z])** — вычисляет сумму всех элементов в последовательности. Например, sum([1, 2, 3]) вернет 6. Обратите внимание, что числа заключены в квадратные скобки, обозначающие список из чисел. Подробнее о списках мы узнаем немного позже.

Функция **pow(x,y)** — используется для возведения числа в степень. Она принимает два аргумента — основание и показатель степени — и возвращает результат возведения основания в эту степень. Например, pow(2,3) вернет 8.

Функции можно помещать внутри других функций. Например, ***max (1,2,4,66,abs(-88))*** вернет 88. По правилам приоритетов сначала произойдет операция во вложенных скобках, где функция ***abs*** преобразует -88 в 88, а затем функция ***max*** найдет максимальное число.



Изображение 3.11.

Для более сложных математических операций с целыми числами можно использовать модуль math. Он предоставляет функции для вычисления факториала, синуса, косинуса и других. Подробнее об этом мы поговорим в соответствующем разделе.

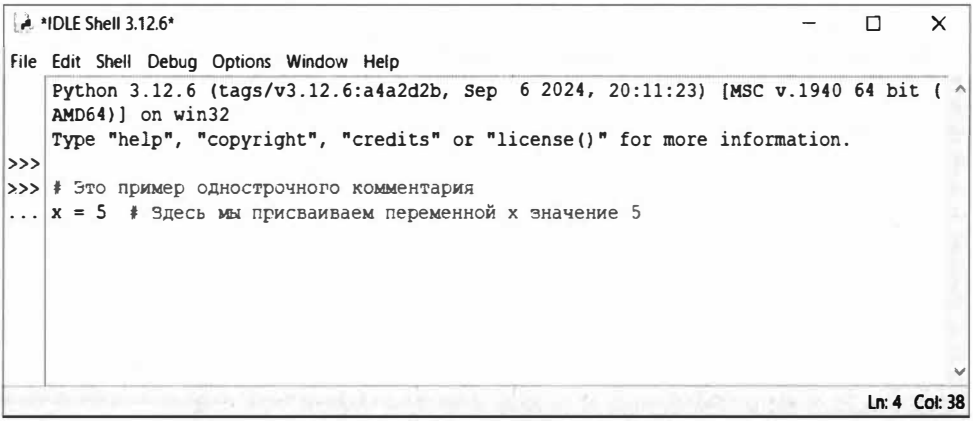
3.3. Комментирование кода

Комментирование кода — это описание работы различных участков программы, что делает код более понятным как для самого программиста, так и для других разработчиков, которые могут работать с ним в будущем.

Комментарии служат как документация. Они могут описывать алгоритмы, функции, переменные и другие элементы кода. Даже простой код, спустя некоторое время, может стать сложным для понимания. Комментарии помогают вспомнить, что делает каждая часть скрипта и почему она написана именно так.

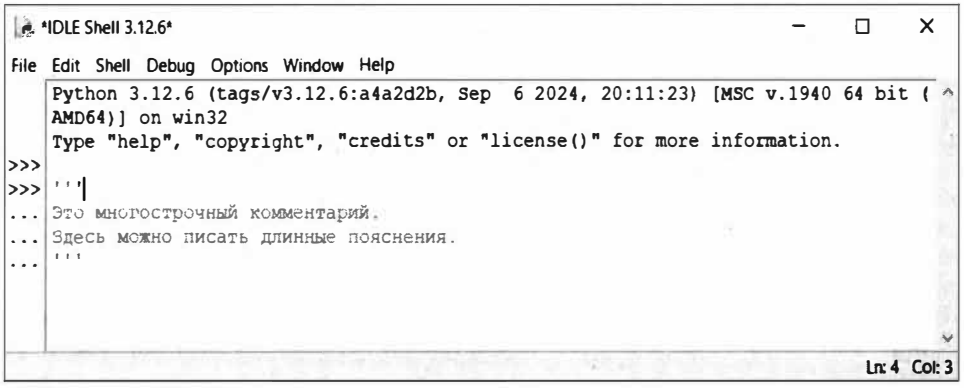
В Python для создания комментариев используется символ #. Все, что находится после этого символа в строке, игнорируется интерпретатором.

```
>>> # Это пример однострочного комментария
>>> x = 5  # Здесь мы присваиваем переменной x значение 5
```



Изображение 3.12.

Для многострочных комментариев можно использовать тройные кавычки """ или ''' (в английской раскладке).



Изображение 3.13.

Однако не стоит перегружать код излишними комментариями, описывая, например, очевидные вещи. Комментировать следует нетривиальные алгоритмы, условия и решения. Пояснения должны быть понятны не только вам, но и другим разработчикам. Избегайте сленга и жаргона. Если вы изменяете код, не забывайте обновлять соответствующие комментарии.

3.3.1. Практические задания

Для закрепления пройденного материала решите несколько простых задач в консоли IDLE. Ниже вы сможете найти ответы для их проверки.

1. Найдите абсолютное значение числа -5.27.
2. Округлите число 3.14159 до двух знаков после запятой.
3. Преобразуйте строку "684" в целое число.
4. Найдите максимальное и минимальное значения в ряду чисел: 3, 7, 2, 9, 5.
5. Найдите сумму всех элементов списка: [5, 4, 32, 18, 841].
6. Возведите число 2 в степень 5.

3.4. Что такое переменная?

Переменная в **Python** — это ячейка или коробка, в которой мы можем хранить какие-либо значения (данные). Например, число или слово. Чтобы Пайтон мог найти нужную информацию, каждой переменной мы даем название и сообщаем программе, что нам нужен объект из коробки с определенным названием.

Откройте консоль IDLE и создайте переменную с любым названием. Чтобы поместить в эту "коробку" информацию, используется оператор присвоения "=".

Обратите внимание, что в Python это не символ равенства, а символ присвоения. Для равенства используется двойной символ "==".

```
>>>
>>>
>>> name = "Иван"
>>> age = 30
>>> price = 19.99
```

Изображение 3.14.

>>> name = "Иван" # Переменная **name** хранит строчное значение **"Иван"**

>>> age = 30 # Переменная **age** хранит целое число **30**

>>> cost = 19.99 # Переменная **cost** хранит вещественное число **19.99**

Переменные в Python состоят из двух основных частей:

- Левая часть — это имя переменной (идентификатор);
- Правая часть — данные, которые хранятся в переменной.

Значение может быть любого типа: число (целое или с плавающей точкой), строка, список, словарь и т.д.

То есть, написав **x = 5**, мы сообщаем Пайтону, что создали контейнер с названием x, который хранит в себе число 5. Давайте создадим еще одну переменную **y** и присвоим ей значение **4**.

```
>>>
>>> name = "Иван"
>>> age = 30
>>> price = 19.99
>>>
>>> x=5
>>> y=4
```

Изображение 3.15.

Теперь, имея две переменные, мы можем оперировать ими вместо чисел:

Листинг 3.1

```
x + y   # Сложили значения в переменных
x + x   # Прибавили значение переменной к этой же переменной
x * 4 + y / 2  # Пример более сложной задачи
```

Таким образом, если в наших контейнерах числа изменятся, программа все равно решит задачу **x + y** и выдаст новый результат.

Например, записав следующие инструкции:

Листинг 3.2

```
x = 5  # Присвоили значение 5 переменной x
y = 4
x = 6  # Присвоили новое значение переменной x
x + y
10
```

То есть изначальное значение 5 было перезаписано на 6, в результате чего ответ был скорректирован.

3.5. Ряд правил при создании переменных

- Имя переменной может состоять из любого количества символов, но начинаться оно должно обязательно с буквы или нижнего подчеркивания. То есть переменные 1x или 2x создать не получится. Однако x1 или _2x можно создавать.
- Python чувствителен к регистру букв. То есть, если мы создадим переменную **X = 5**, но обратимся к ней с маленькой буквы, интерпретатор сообщит нам значение другой переменной, которую мы создавали ранее.

Изображение 3.16.

- Также стоит давать переменным осмысленные названия, чтобы самому в итоге в них не запутаться. Например, переменную стоимости предмета лучше назвать *cost* или *stoimost*, а не *x* или *y* (stoimost = 25).
- Переменная не может состоять из нескольких слов, но, если возникает такая необходимость, слова соединяют в одно. Например, *cost_item*, или *stoimost_predmeta*, или *StoimostPredmeta*. В последнем случае каждое слово начинается с большой буквы, это называется "Верблюжьей записью".

Такой способ не очень удобен для восприятия, поэтому большинство программистов предпочитают разделять слова нижним подчеркиванием.

- Переменную нельзя называть ключевыми словами, которые зарезервированы системой.

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

С большинством из них мы постепенно познакомимся, поэтому зубрить их необязательно. Кроме того, они подсвечиваются оранжевым цветом в IDLE, вы всегда сможете их отличить. В других редакторах кода они также будут иметь свой собственный цвет, он зависит от настроек стиля вашего редактора. Как правило, это желтый, оранжевый или синий цвет, в зависимости от фона.

Также имеются слова, подкрашенные фиолетовым, обозначающие какие-либо функции. С некоторыми из них мы уже познакомились ранее (abs, round, min, max и т.д.). Теоретически можно создать переменные с такими же именами, но в таком случае они потеряют свои первоначальные функции и будут хранить ваше значение.

Нескольким переменным мы можем присваивать одинаковое значение одной строкой. Это называется **массовое присвоение**.

```
>>>
>>> x=y=z=5
>>> x
5
>>> y
5
>>> z
5
>>>
```

Изображение 3.17.

Вместе с тем существует **множественное присвоение**, когда мы записываем через запятую несколько переменных и также через запятую несколько значений. Каждое из значений присваивается в порядке очередности.

```
>>>
>>> x, y, z = 5, 3.14, 'Hi'
>>> x
5
>>> y
3.14
>>> z
'Hi'
>>>
```

Изображение 3.18.

3.5.1. Практические задания

1. Создайте переменную ***name*** и присвойте ей ваше имя.

2. Создайте три переменных и присвойте им числовые значения. Затем получите сумму всех чисел в переменных.

3. Найдите максимальное и минимальное значение этих переменных.

4. С помощью различных операций поменяйте местами значения первых двух переменных. Например, x=5, y=7 нужно изменить на x=7, y=5.

Глава 4.

Ввод и вывод данных

4.1. Функция *input*

Функция **Input()** в Python служит для получения данных от пользователя во время выполнения программы.

С примером такой функции вы могли встречаться на многих интернет-ресурсах, где требуется вводить логин и пароль. Она работает следующим образом:

- Когда программа доходит до строки с функцией input(), она приостанавливается и выводит на экран то, что указано в скобках (если что-то указано).
- Далее программа ожидает, пока пользователь введет данные с клавиатуры и нажмет клавишу **Enter**.
- Введенные пользователем данные возвращаются в виде строки и могут быть присвоены переменной.

В коде это выглядит так:

Листинг 4.1

```
>>> x=input()
# Здесь программа ожидает ввода данных от пользователя
```

Проводя аналогию, представьте, что input() — это пустая коробка. Вы просите пользователя положить что-то в эту коробку. Когда пользователь положил туда предмет (ввел данные), вы заклеиваете коробку и пишете на ней "x". Теперь, когда вам нужно узнать, что находится внутри коробки, вы просто смотрите на этикетку "x".

```
>>>
>>> x=input() # В переменную x сохранятся введённые данные
    684
>>> x # обращаемся к переменной, чтобы увидеть её значение
    '684'
>>>
```

Изображение 4.1.

Обратите внимание, что введенные данные заключены в кавычки. Это означает, что тип данных **строка** (str). Если ввод требует имени или любого другого текста, то это нормально. Однако, если пользователь вводит числовые значения, необходимо конвертировать их в соответствующий тип, например, **int** или **float**.

Конвертация типов (или приведение типов) — это процесс преобразования данных из одного типа в другой. Например, вы можете преобразовать число в строку, строку в число и так далее. Это полезно, когда вам нужно выполнить операции с данными разных типов или когда вы получаете данные из внешних источников в неподходящем формате.

В Python для явной конвертации типов используются встроенные функции:

- int() — преобразует значение в целое число.
- float() — преобразует значение в число с плавающей точкой.
- str() — преобразует значение в строку.
- bool() — преобразует значение в логическое значение (True или False).

То есть в настоящий момент введенное число является текстом. Это можно проверить с помощью проверки типа:

Листинг 4.2

```
>>> type(x) # Выясняем тип данных в переменной x
<class 'str'>
```

Для конвертации в целое число или вещественное необходимо выполнить соответствующие действия:

Листинг 4.3

```
>>> int(x) # Преобразуем содержимое переменной x в целое число
684

>>> float(x) # Преобразуем содержимое переменной x в вещественное
число
684.0
```

Для конвертации в момент ввода запишем следующую конструкцию:

```
x=int(input())
```

Сначала выполняется действие в скобках, где происходит ожидание ввода информации, затем функция ***int*** преобразует данные в целое число. Аналогично этому можно записать конвертацию при вводе числа с плавающей точкой:

```
x=float(input())
```

```
>>>
>>> x=int(input())
    684
>>> x=float(input())
    684
>>> x
    684.0
>>>
```

Изображение 4.2.

Стоит иметь в виду, что если в конструкцию x=int(input()) будет введено дробное число, например 2.5, интерпретатор сообщит об ошибке, так как из трех введенных символов точка не может быть преобразована в число. То же самое касается введенных букв. Вместе с тем мы можем оставлять для пользователя текстовые подсказки, чтобы он понимал, что именно нужно ввести.

Листинг 4.4

```
>>> age=int(input('Сколько вам полных лет? '))
Сколько вам полных лет? 20
>>> age # Узнаем содержимое переменной
20
```

4.2. Практический пример "Социальная сеть"

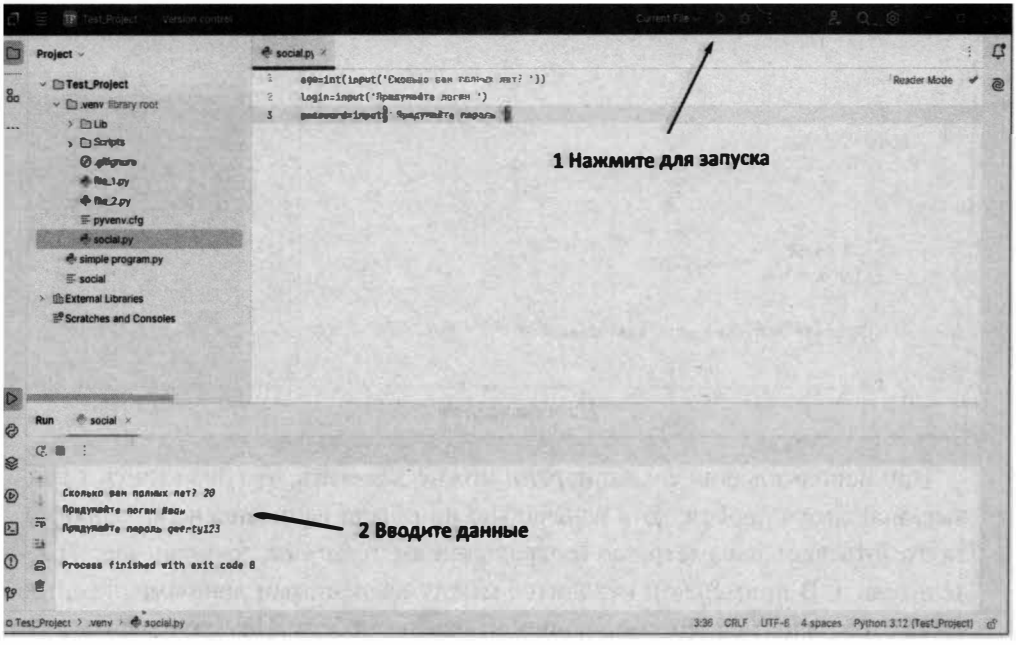
В начале раздела мы вспоминали о веб сайтах, на которых требуется вводить данные. В качестве закрепления материала создайте небольшую форму ввода при регистрации в социальной сети.

Откройте редактор кода Pycharm или аналогичный. На основе пройденного материала напишите сценарий, в котором у пользователя запрашивается возраст, логин и пароль. Также не забудьте добавить конвертацию типов, где это необходимо, и подсказки для пользователя, что ему нужно ввести. Проверьте работоспособность. Сохраните файл с программой.

Решение:

Листинг 4.5

```
age = int(input('Сколько вам полных лет? '))
login = input('Придумайте логин ')
password = input('Придумайте пароль ')
```

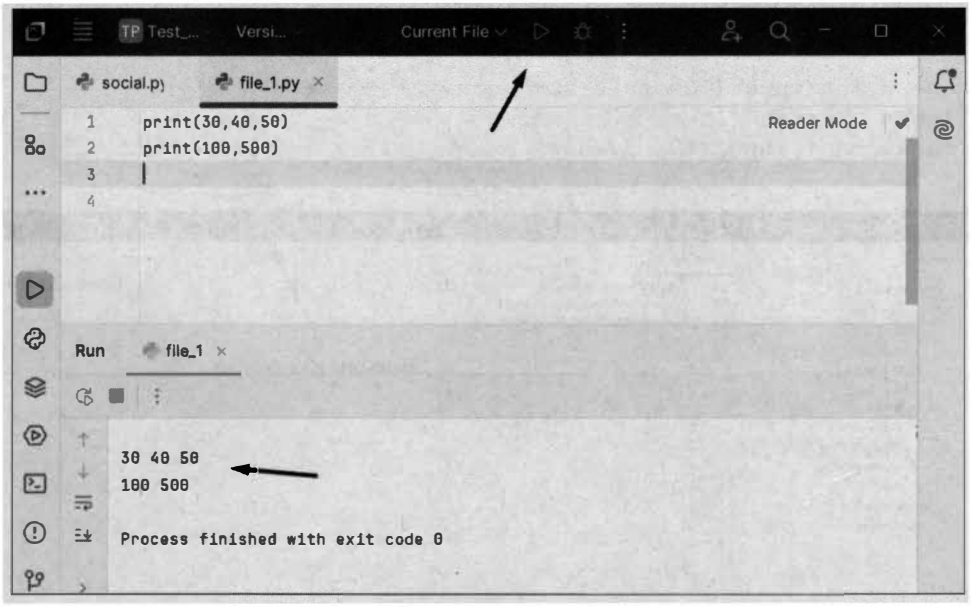


Изображение 4.3.

Если у вас получилось написать программу без подглядывания в ответ, могу вас поздравить, уже совсем скоро вы сможете создать свою социальную сеть, как Павел Дуров или Марк Цукерберг.

4.3. Функция *print*

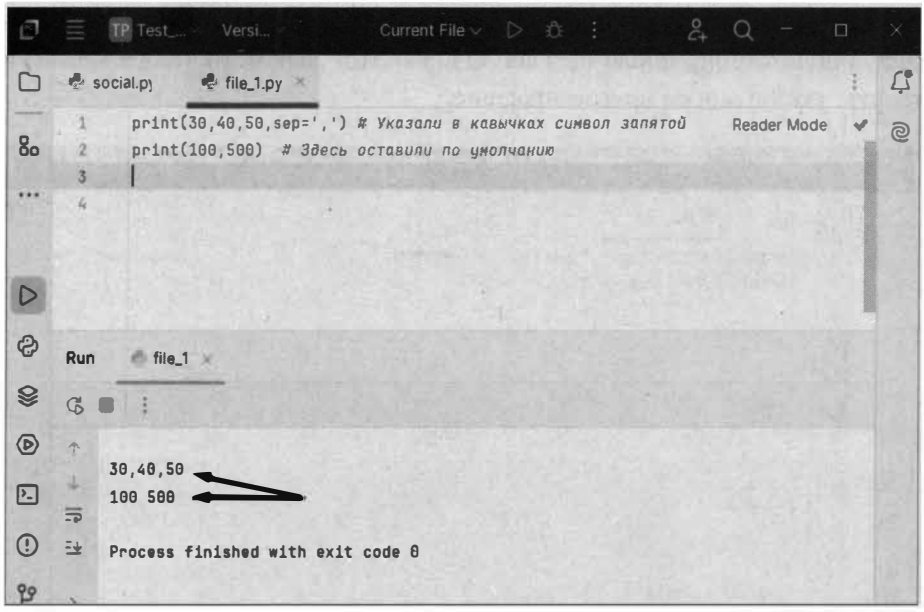
Функция **print()** в Python используется для вывода различных данных на консоль (экран). Это один из самых базовых и часто используемых инструментов. Он будет полезен для вывода результатов работы программы, для создания логов, а также для показа какой-либо информации пользователю.



Изображение 4.4.

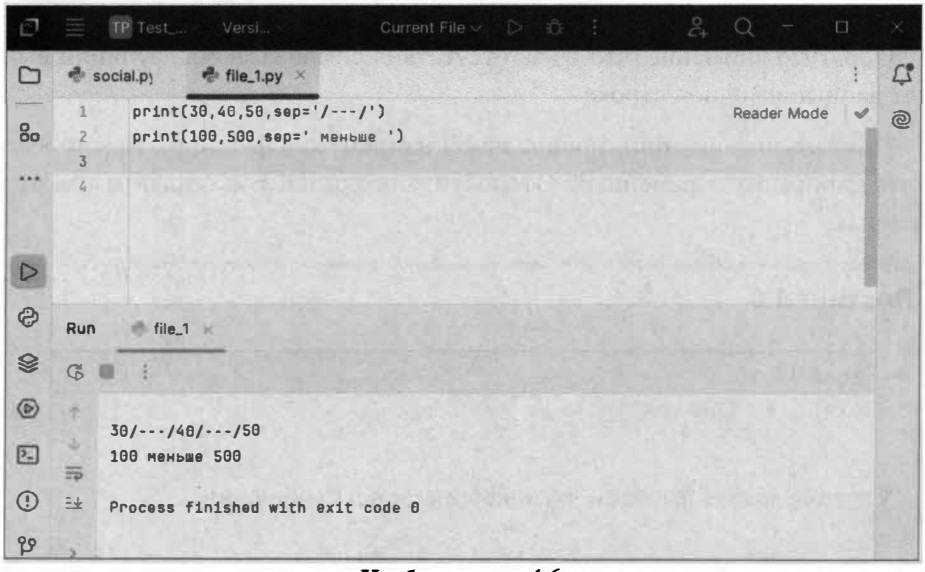
При использовании функции ***print*** можно заметить, что она выводит данные, используя пробел, хотя изначально они были написаны через запятую. За это отвечает параметр *sep* (сокращение от *separator*, что означает "разделитель"). В примере он находится между введенными данными, там, где у нас стоит запятая. По умолчанию он равен пробелу. Поэтому запятые заменяются на пробел.

Чтобы изменить параметр на другой символ, нужно указать его в кавычках, через запятую, прописав новый разделитель:



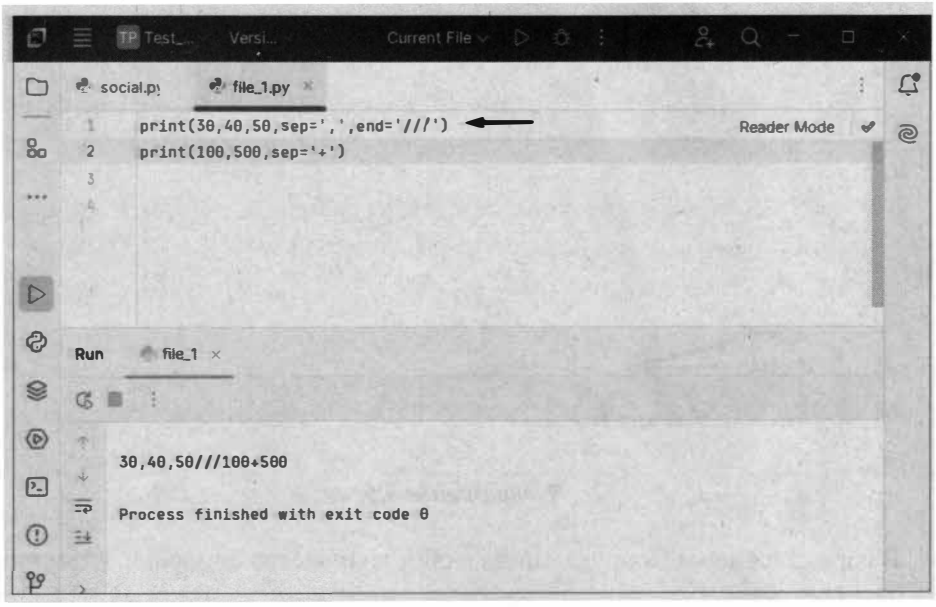
Изображение 4.5.

В параметре sep можно указывать любое количество символов, в том числе и целые слова.



Изображение 4.6.

Еще один параметр, находящийся внутри функции ***print*** — это параметр *end*. Он определяет, какой символ будет добавлен в конце выводимой строки. По умолчанию это символ новой строки *\n*, который переводит курсор на следующую строку после вывода. Однако этот параметр также можно изменить, указав любое другое значение.



Изображение 4.7.

Обратите внимание, что из-за отсутствия символа \n обе функции выводят данные на одной строке.

Помимо информации, прописанной напрямую в функции, можно выводить данные из переменных. Откройте консоль IDLE и создайте три переменных:

Листинг 4.6

```
>>> day = 'Понедельник'
>>> date = 30
>>> month = 'Января'
```

Теперь давайте выведем эту информацию в сообщении:

```
>>> print("Сегодня", day, date, month)
```

Таким образом, мы поместили в кавычки текст для вывода, а также переменные, данные из которых отобразятся в той же строке. Заметьте, что переменные находятся за кавычками, иначе они отобразятся как простой текст day, date, month.

Еще один вариант вывода с помощью символа **%s**, который также подставляет определенные значения вместо себя. В данном случае вся информация помещается в кавычки, а за ними в скобках указываются переменные.

```
>>> print("Сегодня %s %s %s" %(day, date, month))
```

Каждый символ *%s* в кавычках соответствует своей переменной в скобках. То есть первый символ соответствует первой переменной, второй символ — второй переменной и т.д.

Следующий вариант форматирования с помощью **F-строк.** Он является наиболее удобным для написания и легкочитаемым. F-строка начинается с буквы *f* перед кавычками. Внутри кавычек прописывается вся информация, а переменные просто заключаются в фигурные скобки.

```
>>> print(f"Сегодня {day}, {date} {month}.")
```

Для вывода многострочных данных можно использовать дополнительную переменную:

Листинг 4.7

```
# Записываем в переменную today многострочные данные
today= f'''Сегодня кажется {day},
примерно {date} {month}'''
print(today) # Выводим содержимое переменной
```

Также для вывода можно использовать метод **.format()**. Он представляет собой комбинированный вариант предыдущих примеров.

При написании текста в кавычках используются фигурные скобки {} для обозначения мест, куда будут вставлены значения. Эти скобки могут содержать индексы, чтобы указать, какое значение необходимо подставить.

```
print("Сегодня {}, примерно {} {}.".format(day,date,month))
```

4.3.1. Практические задания

1. Откройте файл с формой входа в "Социальную сеть" из предыдущего задания. Допишите приветственное сообщение в конце скрипта, в котором сообщается пользователю, какие данные он ввел.
2. Создайте скрипт, который будет считать периметр четырехугольника. С помощью функции ***input*** запросите у пользователя размеры для каждого стороны фигуры. Затем, с помощью математической операции, подсчитайте общую длину всех сторон. С помощью функции ***print*** выведите результат.

Глава 5.

Математические функции и операции сравнения

5.1. Арифметические операции

С простыми арифметическими операциями (+, -, *, /) мы уже знакомы. В данной теме мы разберем деление нацело и по остатку, а также рассмотрим примеры, в которых, помимо стандартных вычислений, эти операции могут пригодиться.

Деление нацело в Python обозначается оператором // (двойной слеш). Оно выполняет деление двух чисел и отбрасывает дробную часть результата, возвращая только целую часть.

Это полезно, когда вам нужно получить целое число в результате деления, например при расчете количества полных групп элементов или индексации элементов в списках.

Пример:

```
>>> 7//3
2
```

Результат 2, дробная часть 0.333... отбрасывается. Тогда как при обычном делении мы получим результат 2.333... Деление нацело возвращает целое число, округляя результат вниз. Таким образом мы можем не только делить без остатка, но и округлять любые числа, разделяя их на единицу (//1).

Пример:

Листинг 5.1

```
number = 3.7
rounded_down = number // 1
print(rounded_down) # Результат 3.0
```

При делении отрицательных чисел результат будет округляться в сторону минус бесконечности. Например, -7 // 3 равно -3. Если один из операндов является вещественным числом (float), результат также будет вещественным, но с отброшенной дробной частью.

Остаток от деления обозначается знаком процента (%). Он позволяет определить остаток, который получается при делении одного числа на другое.

Пример:

Листинг 5.2

```
x = 17
y = 5
ostatok = x % y
# ostatok будет равен 2, так как 17 = 5 * 3 + 2
```

Операцию остаток от деления можно использовать для определения четности числа. Если число делится на 2 без остатка (т.е. остаток равен 0), то оно четное. Кроме того, деление любых чисел на 10 имеет одну закономерность. Ответом всегда является последняя цифра введенного числа. То есть число 4567 состоит из 456 десятков, а семерка станет остатком от деления. Число 9876 состоит из 987 десятков, а шестерка является остатком от деления.

Таким образом можно возвращать последнюю цифру из неизвестного числа, данного на вход. Например, при вводе случайного числа 2345, мы можем отделить пятерку и проводить с ней дальнейшие операции.

Листинг 5.3

```
x = int(input()) # Введите любое целое число
y = x % 10 # В переменной y сохранили остаток от деления
print(y)
```

Аналогично этому мы можем взять у неизвестного числа две последние цифры, разделив его на %100. Например, 2345 состоит из 23 сотен и остатка от деления 45.

Листинг 5.4

```
x = int(input())
y = x % 100
print(y)
```

Если нужно получить три последние цифры, делим число на %1000 и т.д.

При делении без остатка (//) происходит ровно противоположная ситуация. Отбрасываются последние цифры, а оставшиеся являются ответом.

Листинг 5.5

```
4567 // 10 = 456
4567 // 100 = 45
4567 // 1000 = 4 и т.д.
```

Таким образом, комбинируя остаток от деления и деление без остатка, мы можем вычленить из любого числа одну из центральных цифр.

Листинг 5.6

```
x = 98765
y = x // 100 % 10 # x делим нацело, отбрасывая 65,
# а затем вычисляем остаток от числа 987 (%), которое будет равно 7
print(y) # Достали цифру из середины
```

5.1.1. Практические задания

1. Некоторое количество яблок необходимо поровну разложить по нескольким коробкам и узнать, сколько яблок в каждой коробке. С помощью функции ***input*** дайте пользователю ввести нужное количество яблок и количество коробок. Затем, с помощью арифметических операций, вычислите распределение яблок по коробкам. Функцией ***print*** выведите результат.
2. Отредактируйте вышеописанную программу таким образом, чтобы вычислить остаток яблок, которые невозможно поделить поровну на все коробки.

3. С помощью функции ***input*** дайте пользователю ввести любое целое число. В функции ***print*** выведите последнюю цифру числа.
4. С помощью функции ***input*** дайте пользователю ввести трехзначное целое число. Затем, с помощью арифметических операций, найдите вторую справа. Функцией ***print*** выведите результат.
5. С помощью функции ***input*** предложите пользователю ввести трехзначное число. Найдите сумму всех цифр и выведите результат.

5.2. Стандартные математические функции

Python предоставляет широкий набор встроенных функций для выполнения различных математических операций. Они позволяют выполнять сложные вычисления без необходимости писать собственные алгоритмы. Основная часть этих функций содержится в модуле **math**.

> Чтобы импортировать модуль **math**, необходимо написать соответствующее действие:
>
> **import math**

В некоторых случаях вы можете увидеть запись **from math import...** (из математического модуля импортировать функцию, где вместо многоточия имя функции). Это два способа импортировать математический модуль **math**. Каждый из них имеет свои особенности и применяется в разных ситуациях.

> **import math** является полным импортом. Происходит импорт всего модуля **math** в текущее пространство имен.

Для вызова функций из модуля **math** необходимо указывать имя модуля перед именем функции.

Например:

Листинг 5.12

```
import math # Импортировали модуль
result = math.sqrt(16) # Вычислит квадратный корень из 16
print(result)
# Используем функцию .sqrt() из модуля math
```

Импорт всего модуля подходит, когда требуется использовать множество функций из модуля **math** или когда необходимо явно указывать происхождение функции для повышения читаемости кода.

> **from math import...** является частичным импортом. Это позволяет импортировать только конкретные функции или константы из модуля **math**.

Импортированные функции можно вызывать непосредственно, без указания имени модуля.

Например:

Листинг 5.13

```
from math import sqrt, pi # Импортировали функцию sqrt и константу pi
result = sqrt(16) # Не указываем math., так как импортировали sqrt из
модуля
radius = 5
area = pi * radius**2 # Используем pi без указания math.
print(area)
```

Частичное импортирование подходит, когда необходимо использовать только несколько функций из модуля **math** и нет необходимости постоянно указывать имя модуля (math.). Однако при таком импорте существует риск конфликта имен, если в вашем коде уже определены переменные с такими же именами, как импортированные функции.

Помимо числа Пи и квадратного корня, модуль **math** включает в себя множество других функций. Давайте познакомимся с ними.

5.2.1. Функции округления

ceil() — округляет число вверх до ближайшего целого числа. Используется, например, при расчете стоимости, когда нужно округлить сумму вверх.

Листинг 5.14

```
import math
number = 3.14
rounded_up = math.ceil(number)
print(rounded_up) # Ответ 4
```

Переменная *number* содержит вещественное число 3.14. Функция math.ceil(number) округляет это число вверх до ближайшего целого, то есть до 4.

floor() — округляет число вниз до ближайшего целого числа. Работает противоположно **ceil**.

Листинг 5.15

```
import math
number = 3.7
rounded_down = math.floor(number)
print(rounded_down) # Ответ 3
```

round() — округление числа до ближайшего целого (по правилам математического округления).

Листинг 5.16

```
import math
number = 3.14159
rounded_number = round(number, 2)
print(rounded_number) # Ответ 3.14
```

В переменную *rounded_number* сохраняется округленное число из переменной *number*. Число после запятой в скобках указывает количество знаков после запятой, до которых требуется сделать округление. Если это число не указано, округление произойдет до целого числа.

Листинг 5.17

```
import math
number = 3.14159
rounded_number = round(number)
print(rounded_number) # Ответ 3
```

5.2.2. Экспоненциальные и логарифмические функции

exp() — вычисляет экспоненту числа, то есть возводит число Эйлера (e ~ 2.71828) в заданную степень. Данная функция часто используется для моделирования процессов, которые характеризуются экспоненциальным ростом или убыванием, например роста популяции или радиоактивного распада.

Листинг 5.18

```
import math
number = 2 # Число, применяемое в качестве показателя степени
result = math.exp(number)
print(result) # Ответ 7.38905609893065
```

log() — вычисляет натуральный логарифм (логарифм по основанию e) числа. Другими словами, она отвечает на вопрос: "В какую степень нужно возвести число Эйлера, чтобы получить заданное число?". Данная функция применяется для решения уравнений, содержащих логарифмы.

Листинг 5.19

```
import math
number = 10
```

```
result = math.log(number)
print(result) # Ответ 2.302585092994046
# Функция math.log(number) вычисляет натуральный логарифм
числа 10
```

5.2.3. Тригонометрические функции

sin() — вычисляет синус угла. Однако важно помнить, что угол должен быть задан в радианах, а не в градусах. Функция **sin** широко используется в задачах, связанных с тригонометрией, например при расчете длин сторон треугольников, движении по круговой траектории и т.д.

Листинг 5.20

```
import math
angle_radians = math.pi / 4 # 45 градусов в радианах
sine_value = math.sin(angle_radians)
print(sine_value) # Ответ 0.7071067811865476
```

Переменная *angle_radians* содержит значение угла 45 градусов, переведенное в радианы (число Пи, деленное на 4). Функция math.sin(angle_radians) вычисляет синус этого угла. Результат сохраняется в переменной *sine_value* и выводится на экран.

cos() — вычисляет косинус угла. Аналогично функции **sin**, угол должен быть задан в радианах.

Листинг 5.21

```
import math
angle_radians = math.pi / 3 # 60 градусов в радианах
cosine_value = math.cos(angle_radians)
print(cosine_value) # Ответ 0.5
```

tan() — вычисляет тангенс угла. Используется в задачах при расчете наклона прямых, вычислении высот и расстояний.

Листинг 5.22

```
import math
angle_radians = math.pi / 4 # 45 градусов в радианах
tangent_value = math.tan(angle_radians)
print(tangent_value) # Ответ 1.0
```

asin() — вычисляет арксинус (обратный синус) числа. Функция может быть применима для решения задач, связанных с гармоническими колебаниями в физике.

Листинг 5.23

```
import math
sine_value = 0.5 # Синус угла 30 градусов
angle_radians = math.asin(sine_value)
print(angle_radians) # Ответ 0.5235987755982989
```

acos() — вычисляет арккосинус (обратный косинус) числа.

Листинг 5.24

```
import math
cosine_value = 0.5 # Косинус угла 60 градусов
angle_radians = math.acos(cosine_value)
print(angle_radians) # Ответ 1.0471975511965976
```

atan() — вычисляет арктангенс (обратный тангенс) числа.

Листинг 5.25

```
import math
tangent_value = 1 # Тангенс угла 45 градусов
angle_radians = math.atan(tangent_value)
print(angle_radians) # Ответ 0.7853981633974483
```

5.2.4. Другие полезные функции

Функция **radians()** предназначена для прямого преобразования угла, выраженного в градусах, в радианы. Это полезная функция, когда нужно использовать тригонометрические функции, которые ожидают аргумент в радианах.

Листинг 5.26

```
import math
degrees = 45 # Угол в градусах
angle = math.radians(degrees) # Преобразуем в радианы
print(angle) # Ответ 0.7853981633974483
```

В математике и многих областях науки, таких как физика и инженерия, углы часто измеряются в двух основных единицах: градусах и радианах.

Градусы — более привычная для нас единица измерения углов, делящая круг на 360 равных частей.

Радианы — менее интуитивная, но более удобная для математических вычислений единица, определяемая как длина дуги окружности, деленная на ее радиус.

Радианы связаны с геометрическими свойствами окружности и поэтому являются более "естественной" единицей измерения углов в математических формулах и уравнениях. Многие математические функции, такие как синус, косинус и тангенс, имеют более простые производные и интегралы, когда аргумент выражен в радианах. Именно поэтому есть необходимость переводить градусы в радианы и обратно.

Для обратного преобразования радианов в градусы используется формула:

```
degrees = radians * 180 / math.pi
```

В формуле происходит умножение радианов на 180 и деление на число Пи. Результат присваиваем переменной *degrees*.

Листинг 5.27

```
import math
radians = 0.7854 # Угол в радианах (около 45 градусов)
degrees = radians * 180 / math.pi # Преобразуем в градусы
print(degrees) # Ответ 45.00010522957485 (примерно 45 градусов)
```

factorial() — принимает на вход целое неотрицательное число и возвращает его факториал. Факториалы используются в комбинаторике для вычисления числа перестановок, сочетаний и размещений. А также в формулах для вычисления вероятностей различных событий.

Листинг 5.28

```
import math
number = 5
result = math.factorial(number) # Вычисляем факториал числа 5
print(result) # Ответ 120
```

Важно помнить, что факториалы быстро растут с увеличением числа. Даже для относительно небольших чисел факториал может стать очень большим. Для очень больших чисел рекомендуется использовать специальные библиотеки, которые могут работать с большими числами.

Обратите внимание, что в каждом примере мы импортируем модуль **math**. Это было сделано, чтобы каждый из примеров работал в отрыве от других. Однако, если все эти примеры записать в один файл, импортировать модуль достаточно один раз в начале файла.

Если у вас есть несколько файлов, которые составляют одну программу, то в каждый файл, где используются функции из модуля **math**, необходимо добавить строку *import math*. Это связано с тем, что каждый файл имеет свое собственное пространство имен и импортирование модуля происходит в рамках этого пространства.

5.2.5. Практические задания

1. У нас есть какое-то количество яблок, которые необходимо расфасовать по коробкам. Каждая коробка может вместить 5 яблок. Напишите программу, которая предложит пользователю ввести количество яблок. С помощью математических операций и функции ceil выясните, сколько потребуется коробок, чтобы поместить все яблоки.
2. У нас есть какое-то количество яблок, которые необходимо упаковать в коробки по 5 штук. Напишите программу, которая предложит пользователю ввести количество яблок. С помощью математических операций и функции floor выясните, сколько коробок будет точно заполнено.
3. Три магазина заказали разное количество яблок с нашего яблочного склада. Выясните, сколько нам потребуется коробок, чтобы доставить каждому магазину его заказ, учитывая, что в одной коробке не могут быть два заказа. Каждая коробка вмещает 5 яблок. Напишите программу, которая предложит ввести количество яблок для каждого магазина. Затем, с помощью математических операций и функций, подсчитайте нужное количество коробок и выведите результат на экран.

5.3. Логический тип *Bool* и операторы сравнения

Логический тип данных **bool** (сокращение от Boolean) используется для представления двух возможных состояний: истина (True) или ложь (False).

Этот тип играет ключевую роль в условных выражениях, циклах и других логических операциях. Обратите внимание, что они всегда пишутся с заглавной буквы и подсвечиваются желтым. То есть являются ключевыми словами, а значит, эти имена нельзя использовать для названия переменных, функций и пр.

Например, открыв консоль IDLE и введя значение 5 > 4 (пять больше четырех), мы получим ответ True, потому что выражение верно (истина). Если ввести 5 < 4, ответ будет False, потому что выражение неверно (ложь).

```
>>>
>>> 5 > 4
    True
>>> 5 < 4
    False
>>>
```

Изображение 5.1.

При сравнении мы можем использовать следующие операторы:

- **>** (больше) — проверяет, больше ли левое значение правого. Например: x > y.
- **<** (меньше) — проверяет, меньше ли левое значение правого. Например: x < y.
- **>=** (больше или равно) — проверяет, больше ли левое значение правого или равно ему. Например: x >= y.
- **<=** (меньше или равно) — проверяет, меньше ли левое значение правого или равно ему. Например: x <= y.
- **==** (равно) — проверяет, равны ли два значения. Например: x == 5. Символ равно обозначается двойным символом "==", не путайте его с одиночным символом "=", который означает присвоение.
- **!=** (не равно) — проверяет, не равны ли два значения. Например: x != y (*x* не равен *y*).

> Заметьте, что операторы с двойным символом >=, <=, ==, != пишутся слитно, без пробела между символами.

Примеры использования:

Листинг 5.32

```
# Простые сравнения
x =    # Присваиваем переменным значения
y = 5
print(x == y) # Выведет False (10 не равно 5)
print(x > y) # Выведет True (10 больше 5)

# Цепочки сравнений
print(1 < 2 < 3) # Выведет True (2 больше 1 и меньше 3)
print(10 >= 5 == 5) # Выведет True
# Центральная пятерка меньше или равна 10 и равна 5 справа

# Сравнение строк
a = "hello" # Присваиваем переменным значения
b = "world"
print(a == b) # Выведет False (символы слов отличаются)

# Регистр тоже важен
a = "hello"
b = "Hello"
print(a == b) # Выведет False (первый символ отличается)
```

5.3.1. Логические операторы

> Операторы сравнения часто используются совместно с логическими операторами **and** (и), **or** (или) и **not** (не) для создания более сложных условий.

Например, мы можем проверить сразу два сравнения 1 < 2 **and** 2 < 3. Если оба сравнения верны, ответ получим True. В противном случае — False. Данное выражение можно прочитать так: "Один меньше двух **и** два меньше трех".

Оператор **or** используется, когда верным должно быть хотя бы одно сравнение: 1 < 2 **or** 2 > 3. В данном случае ответ также будет True, так как одно из сравнений верно.

Читается это так: "Один меньше двух **или** два больше трех". То есть первое сравнение верно (True), и это отвечает условию.

Оператор **not** просто "переворачивает" логическое значение. Если выражение было истинным, то **not** делает его ложным, и наоборот.

Листинг 5.33

```
x = True
print(not x) # Выведет False

y = False
print(not y) # Выведет True

a = 5 > 3 # Сравнение верно True
print(not a) # Выведет False, переворачивая результат
```

Применение оператора **not** дважды возвращает исходное значение. Также стоит иметь в виду приоритеты операторов. В выражениях, где используются несколько операторов, они выполняются в определенном порядке. Операторы с более высоким приоритетом выполняются первыми. Для изменения порядка используются скобки.

not имеет самый высокий приоритет.

and имеет более низкий приоритет, чем **not**, но более высокий, чем **or**.

or имеет самый низкий приоритет из этих трех.

Листинг 5.34

```
x = 10 # Присваиваем переменным значения
y = 5
z = 2
result = x > y and y < z or not x == 10
print(result) # Выведет False
```

В этом случае из-за приоритетов операторов вычисления будут происходить в следующем порядке:

1. **x > y** — происходит первое сравнение (10 > 5 — это True).

2. **y < z** — второе сравнение (5 < 2 — это False).

3. **not x == 10** — оператор **not** переворачивает результат (10 == 10 — это False).

4. **x > y and y < z** — оператор **and** применяется к первым двум пунктам. Один из них является ложным, поэтому общий результат False (**and** подразумевает, что оба варианта верны).

5. Далее применяется **or** для результатов четвертого и третьего шагов. Он подразумевает, что хотя бы одно из них должно быть True, но это не так. Поэтому общий ответ уравнения False.

Пример со скобками и сменой приоритетов:

Листинг 5.35

```
x = 10 # Присваиваем переменным значения
y = 5
z = 2
# Выражение с явным указанием приоритетов скобками
result = (x > y) and (y < z or not (x == 10))
print(result) # Выведет False
```

1. **x > y** — сравнение выполняется первым, так как скобки указывают на приоритет (10 > 5 — это True).

2. **y < z** — следующее сравнение (5 < 2 — это False).

3. **not (x == 10)** — здесь оператор **not** переворачивает результат (10 == 10 — это False).

4. **(y < z or not (x == 10))** — теперь выполняется операция or между результатами второго и третьего шагов. Так как ни один из них не является верным (True), общий ответ для содержимого скобок — False.

5. Оператор **and** подразумевает, что выражения слева и справа от него должны быть True, но это не так. Поэтому общий ответ уравнения False.

5.3.2. Преобразование любого значения в булево значение

Python автоматически преобразует различные типы данных в булево (bool) значение в контекстах, где ожидается логическое значение, например в условных выражениях или в логических операциях.

1. Пустые списки, кортежи, множества, словари, строки и число 0 преобразуются в **False**.
2. Непустые структуры данных и ненулевые числа преобразуются в **True**.
3. Значение None преобразуется в **False**.

Листинг 5.36

```
# Пустые структуры
bool([]) # False
bool("") # False
bool({}) # False

# Непустые структуры
bool([1, 2, 3]) # True
bool("hello") # True
bool(" ") # True (пробел также является символом)

# Числа
bool(0) # False
bool(42) # Любые числа True

# None
bool(None) # False
```

5.3.3. Определение четности числа с помощью деления по остатку

Благодаря операторам сравнения можно выполнять множество различных проверок и операций. Например, является ли число четным. Ранее мы познакомились с делением по остатку. Четное число всегда делится на 2 без остатка: 4 % 2 == 0 (двойка дважды помещается в четырех, остаток 0). То есть любое число при делении по остатку на 2 всегда даст нам ответ 0.

Нечетные числа при делении по остатку на 2 всегда дают единицу: 15 % 2 == 1 (двойка семь раз поместится в пятнадцати, остаток 1). Таким образом мы можем выяснить, является ли неизвестное число четным или нет.

```
>>>
>>> number = 10
>>> number % 2 == 0 # Проверяем на четность
    True
>>>
>>> number = 11
>>> number % 2 == 0
    False
>>>
```

Изображение 5.2.

5.3.4. Определение кратности числа

Когда мы говорим, что одно число кратно другому, мы подразумеваем, что первое число делится на второе без остатка. Например, число 12 кратно 3, потому что 12 делится на 3 без остатка (12 / 3 = 4 # остаток 0).

Для определения кратности одного числа другому в программировании используется операция взятия остатка от деления. При делении на это число, если ответ равен 0, число является кратным — True, иначе — False. То есть, имея ряд чисел от 1 до 12, мы можем выяснить, какие из них являются кратными трем. Или, например, достать из этого ряда все числа, кратные 3.

Листинг 5.37

```
number = 3
number % 3 == 0 # Если ответ True, число делится без остатка на 3

number = 4
number % 3 == 0 # Ответ False

number = 6
number % 3 == 0 # Ответ True

number = 9
number % 3 == 0 # Ответ True

number = 12
number % 3 == 0 # Ответ True
```

Если нам нужно определить числа не кратные нашему числу, происходит противоположное сравнение. То есть в нашем случае все числа, не равные 0, являются некратными.

Листинг 5.38

```
number = 7
number % 3 != 0 # Ответ True
```

Деление по остатку 7 % 3 == 1, то есть результат **не равен** (!=) нулю.

5.3.5. Определение, находится ли число в определенном диапазоне

Чтобы определить, находится ли число в нужном диапазоне чисел, мы также можем использовать операции сравнения.

Листинг 5.39

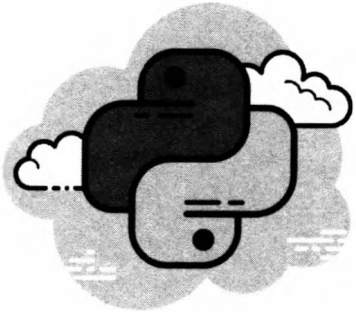
```
number = 15
print(number >= 10 and number <= 20) # Ответ True
```

В примере выше мы определяем, находится ли число в переменной number в диапазоне от 10 до 20. Сначала вычисляется ответ, является ли пятнадцать больше или равно десяти (True), затем является ли пятнадцать меньше или равно двадцати (True). Если ответы слева и (and) справа верны, общий ответ True.

5.3.6. Практические задания

1. Напишите программу, которая предлагает пользователю ввести любое целое число. Затем проверьте, является ли это число кратным пяти. Выведите ответ на экран. True является кратным 5, False не является кратным 5.

2. Напишите программу, которая предлагает пользователю ввести любое целое число. Затем проверьте, является ли это число двузначным. Выведите ответ на экран. True — число двухзначное, False — число не является двузначным.



Глава 6.

Условные операторы

6.1. Оператор *if*

Условный оператор If — это одна из основных конструкций в большинстве языков программирования. Он позволяет программе принимать решения в зависимости от того, истинно или ложно некоторое условие.

Представьте, что, выходя на улицу вам нужно решить, брать зонт или нет. Вы примете это решение в зависимости от погодных условий. Если идет дождь — берете зонт. В программировании за это условие отвечает оператор *if*, который с английского переводится как "если".

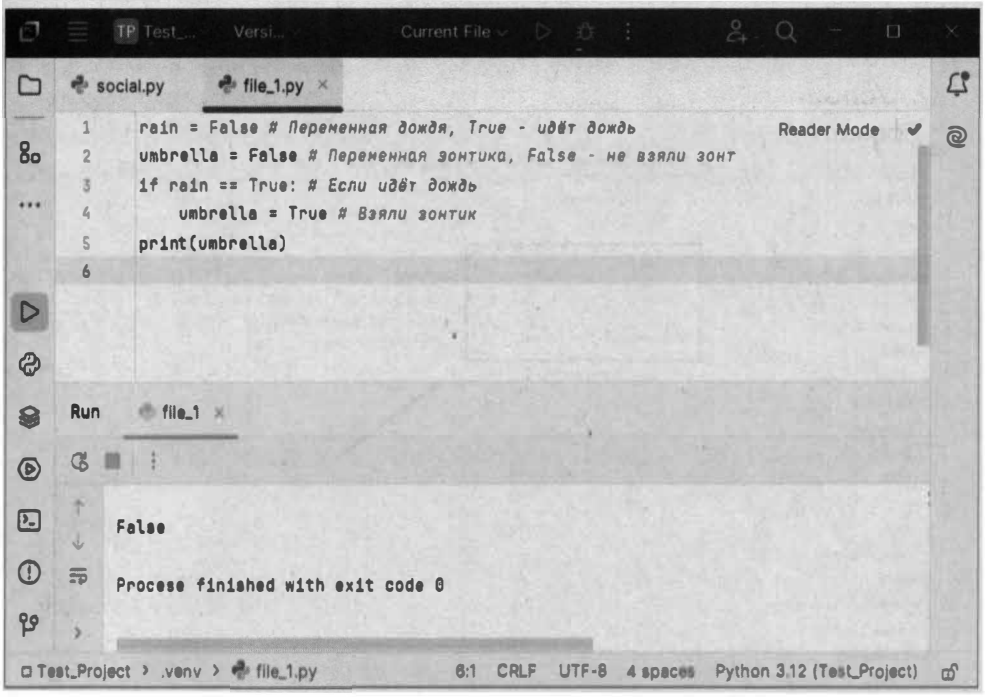
Листинг 6.1

```
rain = True # Переменная дождя, True — идет дождь
umbrella = False # Переменная зонтика, False — не взяли зонт
if rain == True: # Если дождь идет, то:
    umbrella = True # взять зонтик
print(umbrella) # Ответ True
```

В примере выше мы создали две переменные, которые имеют определенные значения. Далее идет условие *if rain == True:*, которое дословно можно перевести как "Если дождь идет, то:". Если значение переменной дождя равно булеву значению True, выполняются инструкции после двоеточия. То есть что нужно сделать? Взять зонт.

Переменная зонта (umbrella) по умолчанию установлена в значение False. Выходя на улицу, мы не берем зонт. Но, если дождь идет, переменная переключается в значение True. В последней строке мы выводим на экран состояние зонта — взяли или нет.

Напишите вышеописанный пример в редакторе кода и измените значение переменной *rain* в состояние False (дождь не идет). После запуска скрипта будет получен ответ umbrella = False. То есть в строке проверки *if rain* произойдет сравнение, равно ли значение дождя (False) булеву значению == True. Ответ: нет, не равно, значит, инструкции после двоеточия не будут выполнены. Эта часть кода будет пропущена, и произойдет переход к строке *print(umbrella)*, где будет отражено значение False, зонт не взяли.

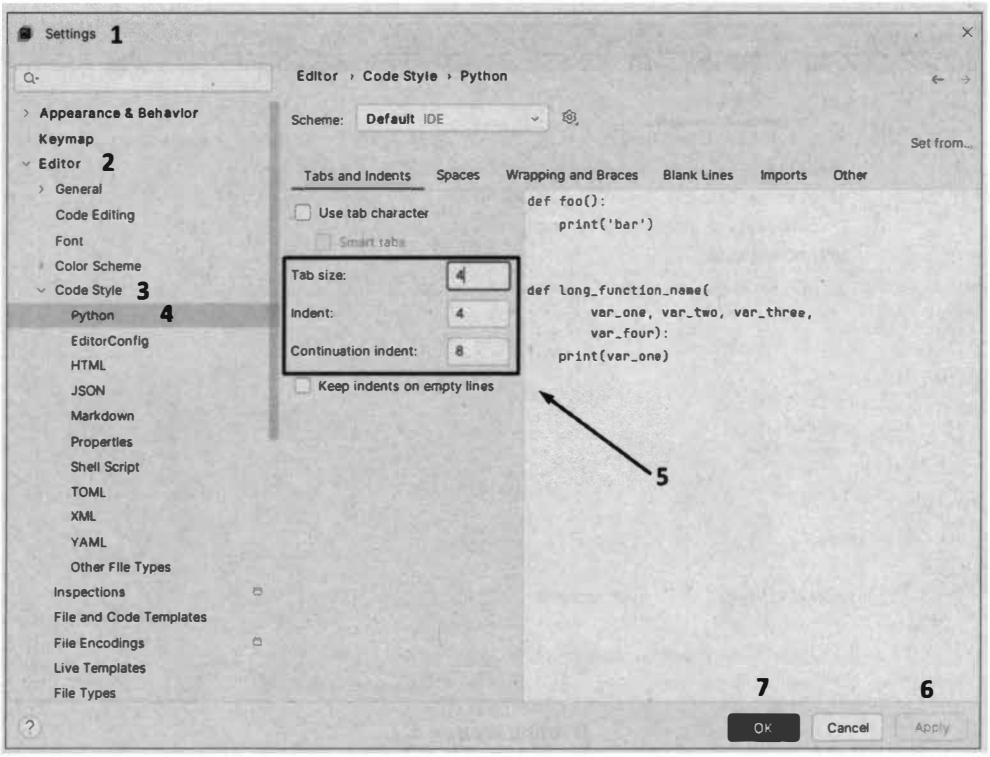


Изображение 6.1.

Обратите внимание, что после строки *if rain == True:* следующая строчка имеет отступ вправо. Этот отступ называется **табуляцией**. По умолчанию он равен 4 пробелам, это общепринятый стандарт, который следует соблюдать. Один отступ можно сделать нажатием клавиши **Tab**, что автоматически выполнит 4 пробела.

Если в вашем редакторе табуляция делает другое количество пробелов, это можно настроить в параметрах. В случае PyChapn данная настройка находится по пути: File => Setting => Editor => Code Stile => Python.

> Параметр **Tab size** отвечает за количество пробелов при отступе от края строки.
>
> Параметр **Indent** — количество пробелов при следующей табуляции.
>
> Параметр **Continuation indent** — при последующих отступах.



Изображение 6.2.

Следует запомнить, что один отступ делается всегда после каждого двоеточия (:). Таким образом мы указываем набор действий, которые необходимо выполнить при определенном условии.

Схематично это выглядит так:

```
Если идет дождь, то: # Ниже набор инструкций после двоеточия
____взять зонтик
____надеть резиновые сапоги
____надеть плащ

Если холодно, то: # Набор инструкций после двоеточия
____надеть шапку
____надеть шарф
____надеть перчатки
```

Такая конструкция с отступами позволяет интерпретатору определить, где начинается и заканчивается один блок, где начинается следующий. Также это помогает визуально видеть границы разных блоков, что делает код легко читаемым и редактируемым.

Пример выше можно записать в коде следующим образом:

Листинг 6.2

```
# Вводим переменные
rain = True # Дождь, True — идет дождь
umbrella = False # Зонтик
boots = False # Сапоги
raincoat = False # Плащ
cold = False # Холод, если True — холодно
cap = False # Шапка
scarf = False # Шарф
gloves = False # Перчатки

# Условия
if rain == True: # Если идет дождь, то:
    umbrella = True # Взять зонтик
    boots = True # Надеть сапоги
    raincoat = True # Надеть плащ

if cold == True: # Если холодно, то:
    cap = True # Надеть шапку
    scarf = True # Надеть шарф
    gloves = True # Надеть перчатки

# Выводы
print(umbrella, boots, raincoat)
print(cap, scarf, gloves)
```

Заметьте, что оба блока будут работать не связанно друг с другом. Независимо от того, идет ли дождь, произойдет проверка на температуру. И если будет холодно, то соответствующие переменные переключаться в состояние True. То же самое с дождем, независимо от того, холодно или нет, проверка дождя переключает свой набор одежды и аксессуаров.

В примере переменная *rain* имеет значение True по умолчанию, это значение совпадает со значением True в условии (if), поэтому выполняются инструкции блока после двоеточия. Переменная холода по умолчанию находится в состоянии False, это не совпадает со значением True в условии (if), соответственно, инструкции после двоеточия будут пропущены. Пайтон перейдет к следующей строке после завершения этого блока.

После запуска скрипта мы можем убедиться, что первый блок был выполнен и все переменные этого блока были переключены в значение True. Переменные второго блока остались в состоянии False, так как условие не было выполнено.

Еще раз обратите внимание на важность отступов. Если мы сместим на один шаг одну строчку *print*, она станет частью блока второго условия и не будет выполнена.

Листинг 6.3

```
if cold == True: # Если холодно, то:
    cap = True # Надеть шапку
    scarf = True # Надеть шарф
    gloves = True # Надеть перчатки

    print(umbrella, boots, raincoat) # Сместили строку
print(cap, scarf, gloves)
```

То есть логика программы изменится, первый *print* будет пропущен, и отобразится только второй результат. В данном примере это может показаться некритичным, так как все переменные в условиях отработают правильно. Но в более сложных скриптах из-за ошибки в табуляции может быть пропущена важная часть кода, в результате чего программа отработает с ошибками.

Условия могут включать в себя несколько сравнений. Например, мы одновременно можем проверить температуру и наличие дождя в одном условии:

Листинг 6.4

```
if rain == True and cold == True: # Если дождь и холодно, то:
    umbrella = True
    boots = True
    raincoat = True
    cap = True
    scarf = True
    gloves = True
print(umbrella, boots, raincoat)
print(cap, scarf, gloves)
```

Теперь все инструкции стали частью одного блока и будут выполнены, только если оба условия будут верны (идет дождь и холодно).

Также условие можно записывать без явного сравнения:

Листинг 6.5

```
if 12:
    umbrella = True
print(umbrella)
```

Из прошлого раздела мы помним, что любое непустое значение преобразуется в значение логического типа bool (12). То есть предыдущая запись может выглядеть так:

Листинг 6.6

```
if True:
    umbrella = True
print(umbrella)
```

6.2. Оператор *else*

Оператор **else** используется в условных конструкциях вместе с оператором **if**. Он позволяет выполнить определенный блок кода в том случае, если все предыдущие условия оказались ложными.

Листинг 6.7

```
temperature = 10 # Температура 10 градусов
if temperature < 0: # Если температура меньше ноля, то:
    cap = True # Надеть шапку
    print("Холодно, я надел шапку")
else: # Иначе
    print("Сегодня достаточно тепло")
```

В примере выше у нас есть переменная температуры, в зависимости от которой будет принято решение: надевать шапку или нет. Сначала происходит первая проверка: "Если температура меньше ноля", то следует надеть шапку. И в этом же блоке мы выводим сообщение о выполненном действии.

Далее идет блок **else**, который можно перевести как "иначе". То есть если температура ниже ноля, одеваем шапку, иначе — идем без шапки. Переменная не изменяет свое значение, поэтому ее нет смысла прописывать, и далее выводим только соответствующее сообщение.

Таким образом, мы создали развилку в программе, которая выполнит одно из условий и в зависимости от этого запустит соответствующие действия. Обратите внимание, что условие **else** никогда не включает в себя дополнительных сравнений. После ключевого слова всегда ставится двоеточие, и с отступом вправо прописывается набор инструкций этого блока.

Листинг 6.8

```
else: # Правильный вариант
    print("Сегодня достаточно тепло")

else temperature > 0: # << Получим ошибку синтаксиса!
    print("Сегодня достаточно тепло")
```

6.3. Оператор *elif*

Оператор **elif** (сокращение от **else if**) можно воспринимать как "иначе если". Оператор используется в условных конструкциях, когда необходимо проверить несколько условий последовательно. Он позволяет создать более сложные ветвления в программе, чем простое **if / else**.

Листинг 6.9

```
temperature = int(input("Какая сейчас температура?: "))
if temperature >= 30:
    print("Сегодня довольно жарко")
elif temperature >= 20:
    print("Сегодня теплая погодка")
elif temperature >= 10:
    print("Могло бы быть и лучше")
else: # Во всех остальных случаях выполнится это условие
    print("Нужно одеться потеплее")
```

Как и в случае с **if / else**, отступы определяют, какой код принадлежит к какому блоку. Обратите внимание, что порядок условий важен. Python проверяет условия по порядку сверху вниз, и как только найдет первое истинное условие, выполнит соответствующий блок кода и пропустит все остальные.

В нашем примере после ввода любого значения температуры происходят следующие события:

1. В условии **if** проверяется значение введенной температуры. Если она больше или равна 30, отображается соответствующее сообщение, и остальной код пропускается. Если температура меньше 30 градусов, Пайтон переходит к следующей проверке.
2. В строке *elif temperature >= 20* происходит сравнение, является ли введенная температура больше или равной 20 градусам. Если происходит эта проверка, значит результат предыдущей оказался ложным, то есть мы точно знаем, что температура меньше 30. Поэтому наше сообщение в *print* будет соответствовать температуре от 20 до 30. Если введенной значение температуры оказалось меньше 20, происходит следующая проверка.
3. Здесь проверяется диапазон температуры от 10 до 20 градусов, так как это условие выполняется только в том случае, если предыдущее оказалось ложным (температура меньше 20). Если введенное значение больше или равно 10, отображается сообщение из этого блока. Иначе Пайтон переходит к следующей строке, где понятно, что температура ниже 10 градусов и нужно одеться потеплее.

Блок **else** является необязательным блоком, он выполняется только в том случае, если ни одно из предыдущих условий не было истинным. Он может отсутствовать, если предыдущие блоки проверяют все возможные варианты. Или если нужно проверить только определенные значения, а все непроверенные значения попадут в блок **else**.

Листинг 6.10

```
temperature = int(input("Какая сейчас температура?: "))
if temperature == 30: # Если температура равна 30, то:
    print("Сегодня 30 градусов")
elif temperature == 22: # Если температура равна 22, то:
    print("Сегодня 22 градуса")
elif temperature == 7: # Если температура равна 7, то:
    print("Сегодня 7 градусов")
else: # Во всех остальных случаях выполнится это условие
    print(f"Температура: {temperature}")
```

Иногда может возникнуть вопрос: "Зачем использовать **elif**, если можно несколько раз прописать **if**?"

Листинг 6.11

```
temperature = int(input("Какая сейчас температура?: "))
if temperature == 30:
    print("Сегодня 30 градусов")
if temperature == 22:
    print("Сегодня 22 градуса")
if temperature == 7:
    print("Сегодня 7 градусов")
```

При использовании множества операторов **if** для проверки одной переменной будет происходить каждая из проверок последовательно. То есть, если пользователь введет число 30, что совпадает с первым вариантом, Пайтон все равно будет проверять все последующие условия. Так как они по сути являются отдельными блоками с условиями. Тогда как вариант с использованием **if-elif** позволяет пропустить бессмысленные проверки после совпадения одного из вышестоящих условий.

6.4. Вложенные условия

Вложенные условия — это конструкция, когда внутри одного условного оператора находится другой условный оператор. Это позволяет создавать более сложные логические выражения и принимать решения на основе нескольких условий.

Например, при необходимости проверить несколько отдельных параметров, чтобы принять окончательное решение. Также вложенные условия часто используются для создания алгоритмов, которые обрабатывают различные сценарии. А при анализе данных они позволяют фильтровать данные по нескольким критериям.

```
1   # Вводим переменные
2   age = int(input("Сколько вам лет?: "))
3   has_license = (input("Есть ли у вас права (Да/Нет)?: "))
4
5   # Вложенные условия
6   if age >= 18: # Если меньше 18 переходим к 11-ой строке
7       if has_license == "Да" or has_license == "да": # Если ответ не "Да" переходим к 9-ой строке
8           print("Можно водить машину")
9       else:
10          print("Можно получить водительские права")
11  else:
12      print("Слишком молод для вождения")
13
```

Изображение 6.3.

На изображении выше можно увидеть один блок с условиями, внутри которого располагается вложенный блок с дополнительными условиями.

- После ввода возраста и ответа на наличие водительских прав программа проверяет, является ли ваш возраст больше или равным 18, если нет, **вложенный блок** пропускается, и выполнение сценария переходит к 11-й строке, где сообщается, что пользователю недостаточно лет.
- Если возраст больше или равен 18, происходит следующая проверка, где программа выясняет, есть ли у пользователя водительское удостоверение. Если введенный ответ совпадает со строкой "Да" (равен == "Да"), то ему

разрешается водить автомобиль. Иначе скрипт переходит к 9-й строке, где сообщается, что пользователь может получить права, так как ему уже есть 18 лет.

Пример с классификацией чисел:

Листинг 6.12

```
number = int(input("Введите число: "))
if number > 0:
    print("Число положительное")
else:
    if number == 0:
        print("Число равно нулю")
    else:
        print("Число отрицательное")
```

В данном примере программа проверяет, какое число ввел пользователь:

- Если число больше ноля, отображается соответствующее сообщение, а весь остальной код пропускается.
- Если число не больше ноля, скрипт переходит к блоку **else**, который включает в себя вложенное условие с дополнительным уточнением. Является ли число равным нулю? Если число равно 0, программа выводит сообщение об этом, а остальной код этого вложенного блока пропускается. Иначе становится понятно, что число не больше и не равно нулю, а значит, оно отрицательное.

Каждое вложенное условие также может иметь свои вложенные условия. А те, в свою очередь, свои вложенные условия. И так до бесконечности. Однако старайтесь избегать большого каскада вложенных условий, так как такой код тяжело воспринимать визуально и он трудночитаем.

В большинстве случаев вложенные условия можно заменить более компактными конструкциями, используя оператор **elif**, тернарный оператор или словари.

6.5. Тернарный оператор

Тернарный оператор — это компактный способ записи условного выражения в одну строку. Он позволяет присвоить значение переменной в зависимости от истинности или ложности условия. По сути это сокращенная форма конструкции **if-else**.

Листинг 6.13

```
# Переменные
x = 5
y = 10
# Тернарный оператор (запись условия в одну строку)
result = "x больше y" if x > y else "y больше или равно x"
print(result) # Вывод: y больше или равно x
```

- В примере выше в переменную *result* сохраняется определенная фраза.
- Сохраняется фраза "x больше y", если переменная x > y (if 5 > 10). Если сравнение верно, условие после else пропускается.
- Если сравнение ложно, переменной присваивается фраза после оператора else "y больше или равно x".

Преимущество такой записи в ее лаконичности, она может использоваться, когда есть только два варианта: либо это, либо то. Однако при более сложных условиях, включающих логические операторы (and, or, not), лучше использовать стандартную конструкцию с **if-elif-else**.

6.5.1. Практические задания

1. Напишите программу, где пользователю предлагается ответить на вопрос: "Сколько персонажей тянуло репку в сказке про репку?". Если ответ правильный, отобразите сообщение, что это правильный ответ. Иначе сообщите, что ответ неверный.

2. Напишите программу, где пользователю предлагается ввести число. С помощью тернарного оператора и математических операций определите, является ли оно четным или нечетным. Выведите ответ на экран.

3. Напишите программу, которая будет определять оценку школьника по количеству набранных баллов. На вход принимается целое число. На выходе отображается сообщение с оценкой, в зависимости от введенного числа, где:

- Числа больше 91 — оценка 5;
- Числа 81–90 — оценка 4;
- Числа 71–80 — оценка 3;
- Числа 61–70 — оценка 2;
- Число меньше 61 — оценка 1.

4. Напишите программу, где пользователю предлагается ввести логин. Если логин неверный, отображается сообщение об ошибке. Если логин верный, пользователю предлагается ввести пароль. Если пароль верный, отображается сообщение о входе в свой профиль. Если пароль неверный, отображается сообщение об ошибке.

Глава 7.

Работа с текстом

7.1. Строки, основные операции

Строка (str) в Python — это тип объекта, который представляет собой последовательность символов, заключенную в кавычки. Это один из самых распространенных типов данных, который используется для представления текста, имен, адресов и любой другой информации, которую можно выразить в виде последовательности символов.

Ранее мы уже поверхностно познакомились со строками в разделе 4.3 Функция print, когда учились выводить на экран различную информацию.

Строки могут выводиться на экран с помощью **двойных** и **одинарных кавычек**.

Листинг 7.1

```
print("Строка в двойных кавычках")
print('Строка в одинарных кавычках')
```

Обратите внимание, что если вывести текст без кавычек, мы получим ошибку, так как интерпретатор воспринимает такие слова как имена переменных, функций и пр., но поскольку таких данных ранее не было создано, мы получаем сообщение об ошибке. Также будет получена ошибка, если смешивать между собой тип кавычек 'одинарные и двойные".

Как и в случае с комментированием кода, для многострочного текста используются тройные кавычки ''' ''' или """ """.

Листинг 7.2

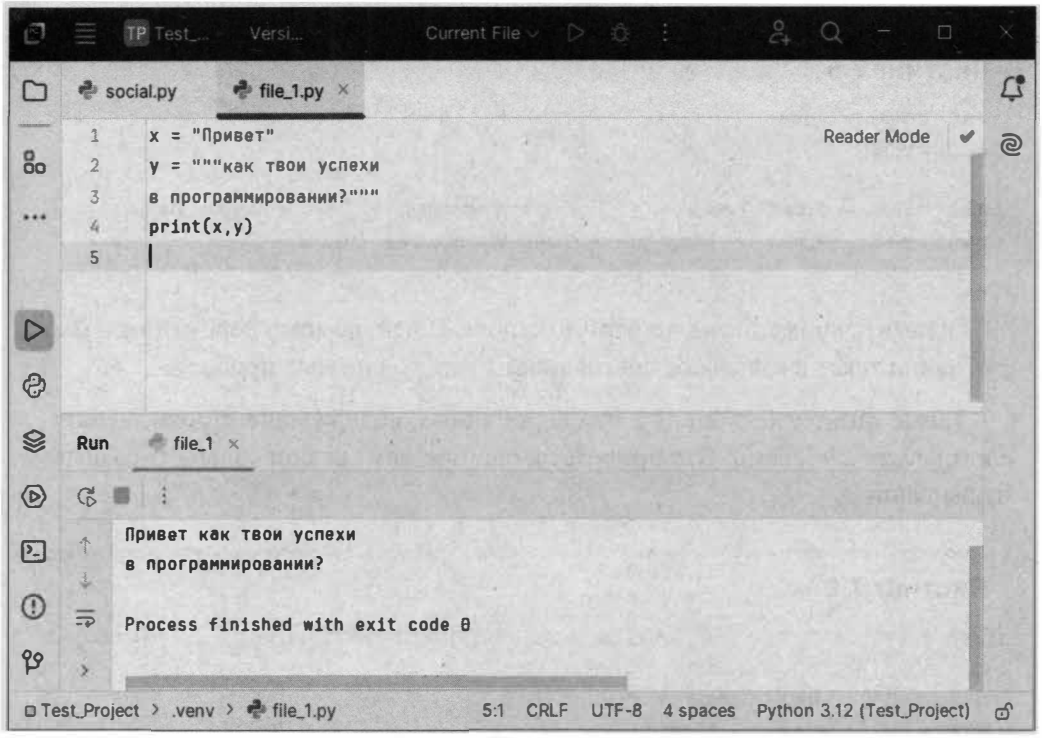
```
print("""Пример
многострочного
текста""")

print('''Еще пример
в одинарных кавычках''')
```

Любой текст можно сохранять в переменных:

Листинг 7.3

```
x = "Привет"
y = """Как твои успехи
в программировании?"""
print(x,y)
```

Результат:



Изображение 7.1.

В примере выше при выводе переменных в функции ***print*** запятая по умолчанию заменяется пробелом, поэтому все сообщение выглядит аккуратно и после слова "Привет" ставится пробел. Однако, если содержимое нескольких переменных сохранить в одной общей переменной, текст будет записан слитно.

Листинг 7.4

```
first_name = "Иван"
patronymic = "Иванович"
last_name = "Иванов"
full_name = first_name + patronymic + last_name
print(full_name) # Выведет: ИванИвановичИванов
```

Оператор **"+"** в данном контексте означает **конкатенацию** (от англ *concatenation*) — процесс соединения двух и более строк в одну. Чтобы исправить наш пример с помощью конкатенации, необходимо добавить символ пробела в кавычках.

Листинг 7.5

```
first_name = "Иван"
patronymic = "Иванович"
last_name = "Иванов"
full_name = first_name + " " + patronymic + " " + last_name
print(full_name) # Выведет: Иван Иванович Иванов
```

Обратите внимание на четвертую строку. В ней, помимо переменных, мы добавили текст в кавычках, состоящий из одного символа пробела.

Также стоит иметь в виду, что переменные, содержащие строки, нельзя складывать с числами. Это приведет к ошибке, так как они содержат разные типы данных.

Листинг 7.6

```
first_name = "Иван" # Тип данных - строка (str)
age = 25 # Тип данных - целое число (int)
full_name = first_name + age
print(full_name) # Ошибка!
```

Чтобы вывести имя с возрастом, необходимо преобразовать число в строку, аналогично тому, как ранее мы преобразовывали текст в число.

Листинг 7.7

```
first_name = "Иван"
age = 25
full_name = first_name + str(age)
print(full_name) # Выведет: Иван25
```

В третьей строчке произошло преобразование переменной *age* в тип **строка** (str). Напомню, что при пользовательском вводе данные принимают тип **строка**.

x = input() — любые данные получают тип строки;

x = int(input()) — преобразовали в целое число при вводе пользователем;

x = float(input()) — преобразовали в вещественное число при вводе.

В нашем примере выше число не вводилось пользователем, а было записано в переменную сразу как целое число (age = 25). Поэтому оно требовало преобразования. Если записать его в кавычках, оно будет иметь тип данных строка (age = "25").

Листинг 7.8

```
first_name = "Иван"
age = "25" # Тип - строка
full_name = first_name + " " + age
print(full_name) # Выведет: Иван 25
```

Итак, оператор "+" сцепляет две строки в одну. Однако, если использовать оператор умножения "*", строка будет написана определенное количество раз:

Листинг 7.9

```
name = "Иван"
print(name * 4) # Выведет: ИванИванИванИван
```

С помощью функции **len** (сокращение от length) мы можем узнать *длину строки*.

Листинг 7.10

```
first_name = "Иван"
print(len(first_name)) # Выведет: 4
```

Помимо длины строк, данная функция определяет длину различных объектов, таких как списки, кортежи и пр. Таким образом, мы можем обновить наш код "социальной сети", добавив условие на длину вводимых символов.

Листинг 7.11

```
password = input("Введите пароль: ")
if len(password) < 8: # Если длина пароля меньше 8 символов, то
    print("Пароль должен быть не менее 8 символов")
else: # Иначе
    print("Отлично, пароль сохранен")
```

Функция **in** используется для проверки, содержится ли определенный элемент в последовательности (например, строке, списке, кортеже, словаре). Она возвращает True, если элемент найден, и False, если нет.

Листинг 7.12

```
name = "Иван"
print('И' in name) # Выведет True
print('я' in name) # Выведет False
```

В примере выше мы выяснили, что большая буква "И" содержится в переменной name, а маленькая буква "я" не содержится. Обратите внимание, что регистр важен. Если бы мы искали маленькую букву "и", ответ был бы отрицательным (False).

Данную функцию мы также можем использовать при регистрации в социальной сети на предмет запрещенных символов:

Листинг 7.13

```
password = input("Введите пароль: ")
if "@" in password: # Если символ @ содержится в пароле, то
    print("Вы ввели запрещенный символ")
else:
    print("Регистрация прошла успешно")
```

Мы можем проверять целую последовательность символов, если они идут в том же порядке:

Листинг 7.14

```
name = "Иван"
print('ва' in name) # Выведет True
print('ав' in name) # Выведет False
```

Как только порядок символов был изменен, мы получаем отрицательный ответ. Поэтому на текущий момент мы можем проверить пароль на запрещенные символы, перебирая каждый из них по отдельности, используя длинную конструкцию **if-elif...elif-else**. Далее, когда мы познакомимся со списками, подобную проверку можно будет сократить.

Вместе с тем мы можем сравнивать строки между собой, они сравниваются так же, как слова в словаре: сначала сравниваются первые символы, затем вторые и так далее.

Листинг 7.15

```
print("qwerty" == "qwerty") # True
print("qwerty" < "z") # True
```

В первом случае все символы совпадают, в ответ мы получаем True, то есть две строки равны между собой. Во втором случае мы также получаем положительный ответ при сравнении, где *z* больше целого ряда символов. Это происходит потому, что при посимвольном сравнении сравниваются не сами символы, а их числовой код. Данный код можно узнать с помощью функции **ord**:

Листинг 7.16

```
print(ord("q")) # 113
print(ord("z")) # 122
```

Как видим, 113 (q) меньше, чем 122 (z). Коды всех символов можно найти в Интернете по запросу *ascii code table*. Для языков, отличных от латиницы, например кириллицы или китайских иероглифов, существуют свои таблицы.

0 -	16 - ▶	32 -	48 - 0	64 - @	80 - P	96 - '	112 - p
1 - ☺	17 - ◀	33 - !	49 - 1	65 - A	81 - Q	97 - a	113 - q
2 - ●	18 - ↕	34 - "	50 - 2	66 - B	82 - R	98 - b	114 - r
3 - ♥	19 - ‼	35 - #	51 - 3	67 - C	83 - S	99 - c	115 - s
4 - ♦	20 - ¶	36 - $	52 - 4	68 - D	84 - T	100 - d	116 - t
5 - ♣	21 - §	37 - %	53 - 5	69 - E	85 - U	101 - e	117 - u
6 - ♣	22 - ▬	38 - &	54 - 6	70 - F	86 - V	102 - f	118 - v
7 -	23 - ↕	39 - '	55 - 7	71 - G	87 - W	103 - g	119 - w
8 -	24 - ↑	40 - (	56 - 8	72 - H	88 - X	104 - h	120 - x
9 -	25 - ↓	41 -)	57 - 9	73 - I	89 - Y	105 - i	121 - y
10 -	26 - →	42 - *	58 - :	74 - J	90 - Z	106 - j	122 - z
11 -	27 - ←	43 - +	59 - ;	75 - K	91 - [	107 - k	123 - {
12 -	28 - ∟	44 - ,	60 - <	76 - L	92 - \	108 - l	124 - |
13 -	29 - ↔	45 - -	61 - =	77 - M	93 - j	109 - m	125 - }
14 - ♫	30 - ▲	46 - .	62 ->	78 - N	94 - ^	110 - n	126 - ~
15 - ☼	31 - ▼	47 - /	63 - ?	79 - O	95 - ÷	111 - o	127 - ⌂
16 - ▶	32 -	48 - 0	64 - @	80 - P	96 -	112 - p	

128 - А	144 - Р	160 - а	176 - ▒	192 - └	208 - ╨	224 - р	240 - Ё
129 - Б	145 - С	161 - б	177 - ▓	193 - ┴	209 - ╤	225 - с	241 - ё
130 - В	146 - Т	162 - в	178 - │	194 - ┬	210 - ╥	226 - т	242 - Є
131 - Г	147 - У	163 - г	179 - ┤	195 - ├	211 - ╙	227 - у	243 - є
132 - Д	148 - Ф	164 - д	180 - ┐	196 - ─	212 - ╘	228 - ф	244 - Ї
133 - Е	149 - Х	165 - е	181 - ┼	197 - ┼	213 - ╒	229 - х	245 - ї
134 - Ж	150 - Ц	166 - ж	182 - ╞	198 - ╞	214 - ╓	230 - ц	246 - Ў
135 - З	151 - Ч	167 - з	183 - ╟	199 - ╟	215 - ╫	231 - ч	247 - ў
136 - И	152 - Ш	168 - и	184 - ╢	200 - ╚	216 - ╪	232 - ш	248 - °
137 - Й	153 - Щ	169 - й	185 - ╣	201 - ╔	217 - ┘	233 - щ	249 - ∙
138 - К	154 - Ъ	170 - к	186 - ║	202 - ╩	218 - ┌	234 - ъ	250 - ·
139 - Л	155 - Ы	171 - л	187 - ╗	203 - ╦	219 - ▒	235 - ы	251 - √
140 - М	156 - Ь	172 - м	188 - ╝	204 - ╠	220 - ▄	236 - ь	252 - №
141 - Н	157 - Э	173 - н	189 - ╜	205 - ═	221 - ▌	237 - э	253 - ¤
142 - О	158 - Ю	174 - о	190 - ╛	206 - ╬	222 - ▐	238 - ю	254 - ■
143 - П	159 - Я	175 - п	191 - ┐	207 - ╧	223 - ▀	239 - я	255 -
144 - Р	160 - а	176 - ▒	192 - └	208 - ╨	224 - р	240 - Ё	

Изображение 7.2.

Таким образом, с помощью сравнения двух строк можно проверить правильность введенного логина или пароля:

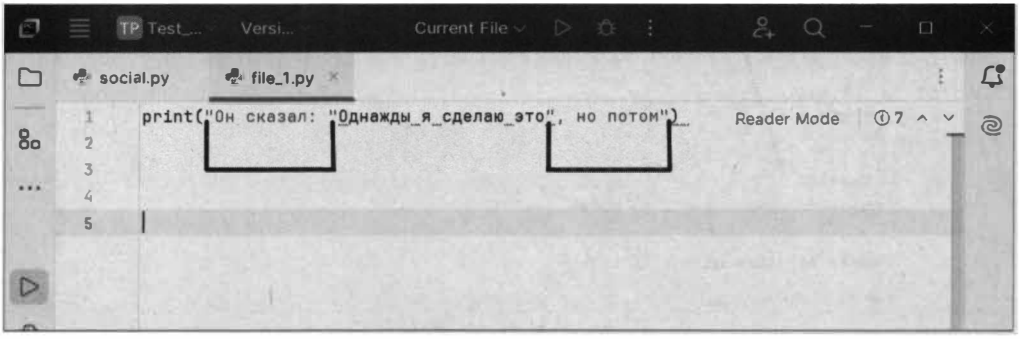
Листинг 7.17

```
login = "admin"
login_input = input("Введите ваш логин: ")
if login_input == login:
    print("Доступ разрешен")
else:
    print("Доступ запрещен")
```

7.2. Эскейп-последовательности

Эскейп-последовательности (или экранированные последовательности) — это специальные комбинации символов, начинающиеся с обратного слеша \, они используются для представления символов, которые сложно или невозможно ввести напрямую, или для выполнения определенных действий при форматировании текста.

Например, при написании цитаты в кавычках эти кавычки воспринимаются интерпретатором как новые кавычки для обозначения начала и конца строки. В итоге мы получаем "строку в строке", которую Пайтон понимает как две отдельные строки с переменной между ними, что в итоге приводит к ошибке.



Изображение 7.3.

Добавив обратный слеш к внутренним кавычкам, мы сообщаем интерпретатору, что они являются внутренними, а не закрывающими и открывающими кавычками.

Листинг 7.18

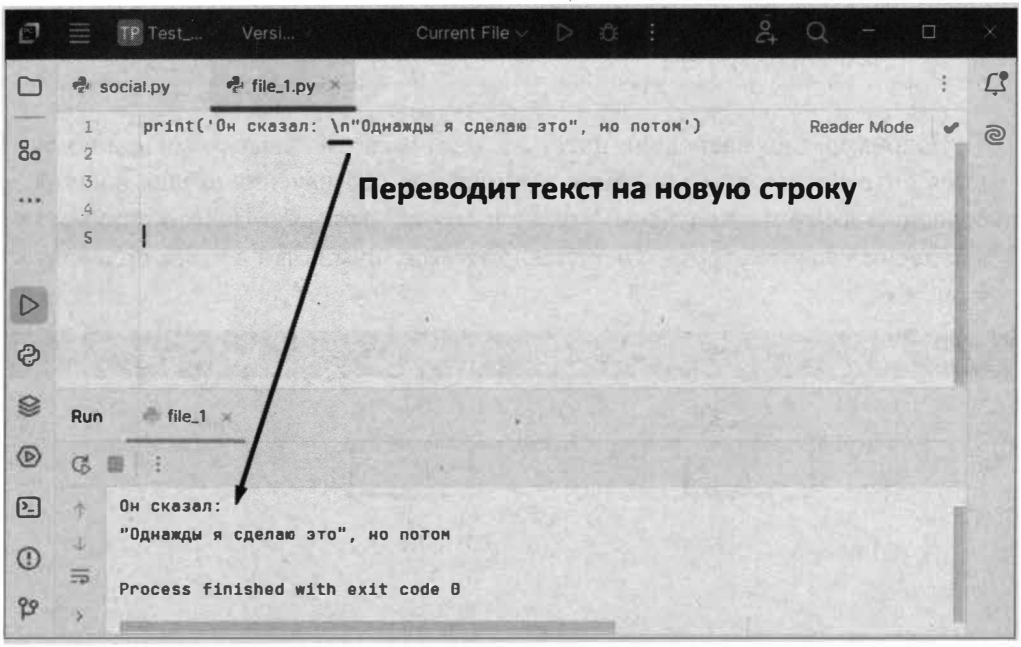
```
print("Он сказал: \"Однажды я сделаю это\", но потом")
```

Кроме того, можно чередовать двойные и одинарные кавычки, чтобы также указать Пайтону, что одни располагаются внутри других.

Листинг 7.19

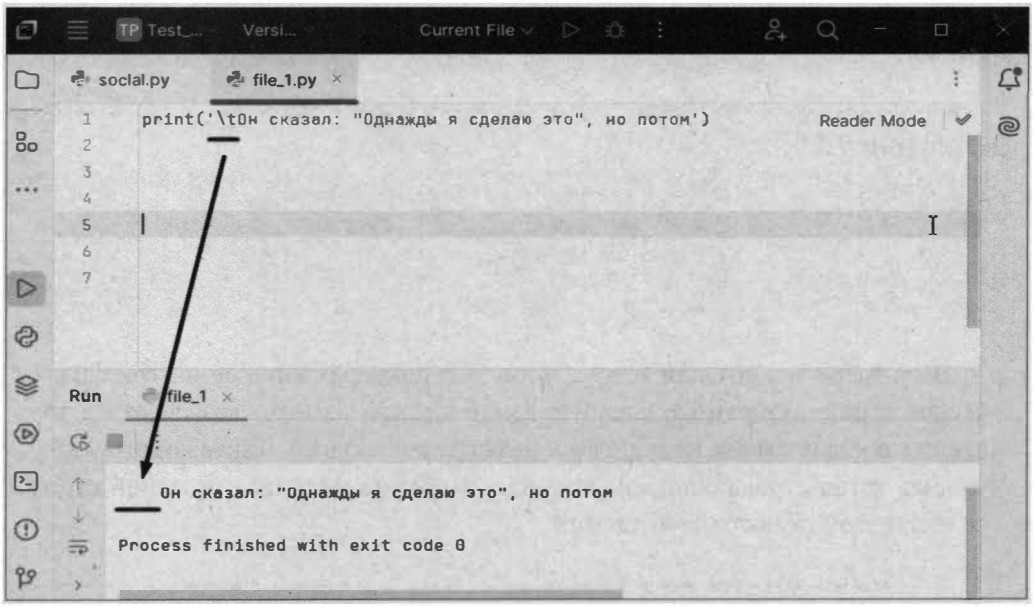
```
print('Он сказал: "Однажды я сделаю это", но потом')
```

Когда Пайтон встречает обратный слеш \, он понимает, что следующий символ является частью специальной последовательности, а не обычным символом. Например, **\n** не интерпретируется как два отдельных символа, а как один символ перевода строки.



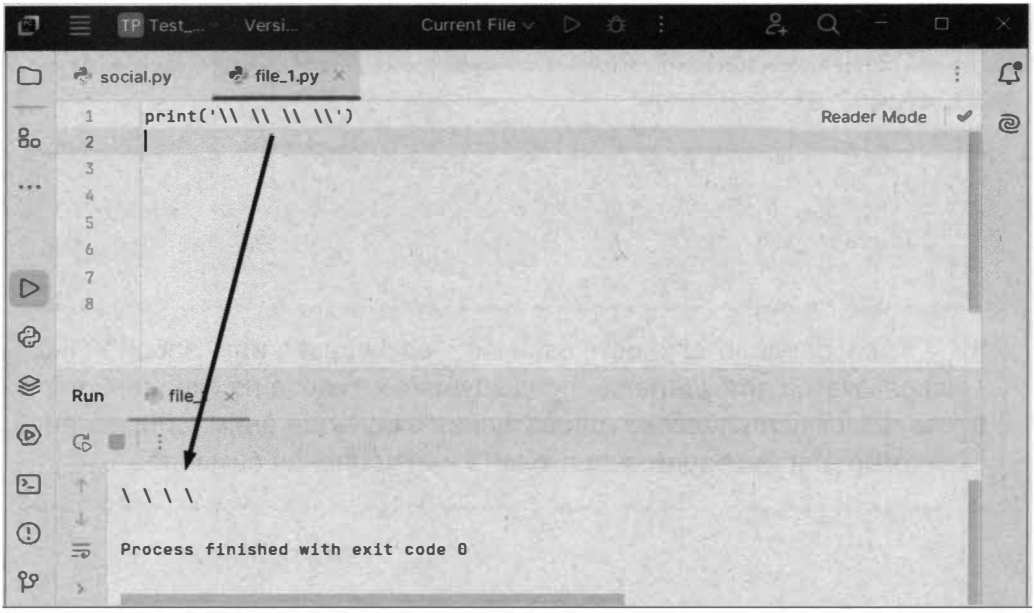
Изображение 7.4.

\t — горизонтальная табуляция. Добавляет несколько пробелов (обычно 4).



Изображение 7.5.

\ — обратный слеш. Используется для экранирования самого себя.



Изображение 7.6.

Если слеш вынести за кавычки, он становится символом продолжения строки.

Листинг 7.20

```
long_string = "Это очень длинная строка, " \
              "которую мы разбили на несколько частей " \
              "для лучшей читаемости."
print(long_string)
```

В примере мы создали одну длинную строку, разделив ее на три физические строки. Обратные слеши в конце каждой из них указывают на то, что следующая строка является продолжением текущей. Такая конструкция удобна, когда строка слишком длинная, чтобы поместиться на одной строке и при этом оставаться читаемой.

\r — возврат каретки. Перемещает курсор в начало текущей строки, не переходя на следующую. Это может быть полезно при создании эффектов анимации или прогресс-баров в консоли.

Для примера можете запустить этот код и увидеть его действие:

Листинг 7.21

```
import time
for i in range(101):
    print(f"\rПрогресс: [{'=' * i}{' ' * (100-i)}] {i}%", end='')
    time.sleep(0.1)
```

\b — эта последовательность означает "backspace", или "забой". Она используется для удаления предыдущего символа на экране. Это также может быть полезно для создания эффектов анимации, например для имитации ввода текста или удаления символов.

Листинг 7.22

```
import time
text = "Загрузка..."
for i in range(len(text)):
    print(text[:i+1], end='', flush=True)
    time.sleep(0.2)
    print("\b" * (i+1), end='', flush=True)
```

\f — "формальная подача" (form feed). Она используется для перехода на новую страницу в текстовых документах.

Сырые строки — иногда нужно, чтобы все символы в строке воспринимались буквально, без интерпретации эскейп-последовательностей. Для этого перед открывающей кавычкой строки ставится буква **r**.

Листинг 7.23

```
path = r"C:\Users\ИмяПользователя\Documents"
```

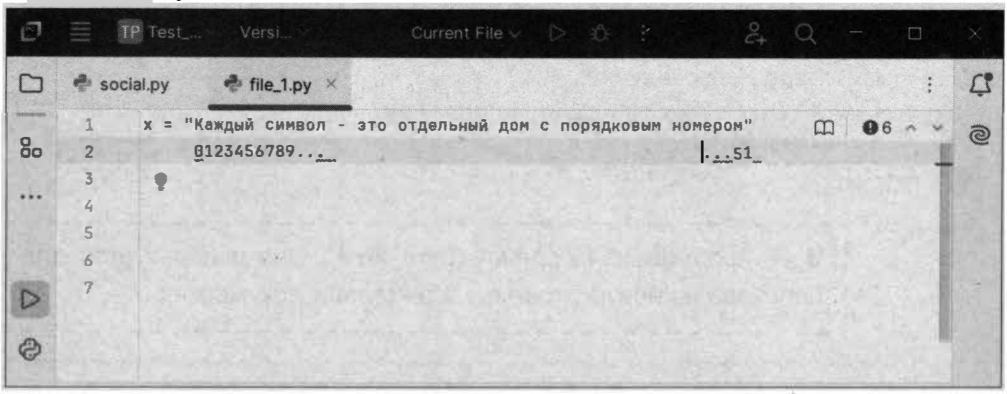
7.2.1. Практические задания

1. Предложите пользователю ввести любой текст, а затем выведите на экран количество символов в этом тексте.
2. Предложите пользователю ввести логин длиной не менее трех и не более десяти символов. В случае успеха или ошибки выведите на экран соответствующее сообщение.

7.3. Индексы и срезы

Индекс в Python — это порядковый номер, который указывает на положение конкретного элемента в упорядоченной коллекции данных, такой как список, строка или кортеж.

Это как номер дома на улице; зная его, вы можете точно определить, где находится нужный вам дом.



Изображение 7.7.

Стоит иметь в виду, что почти все языки программирования, включая Python, начинают отсчет индексов с нуля, а не с единицы. Это может показаться немного непривычным, так как в жизни мы обычно начинаем считать с единицы. Но у такого подхода есть свои веские причины.

В основе его лежит то, как компьютеры хранят данные в памяти. Когда мы создаем список или массив, компьютер выделяет для него непрерывный участок памяти.

- Первый элемент этого участка имеет адрес, который можно считать началом отсчета;
- Индекс элемента — это просто смещение от этого начального адреса;
- Таким образом, индекс 0 указывает на сам начальный адрес, а индекс 1 — на элемент, расположенный на одну ячейку памяти дальше.

Когда мы используем индексы для доступа к элементам, компьютер выполняет простые арифметические операции. Например, чтобы получить адрес элемента с индексом x, компьютер просто складывает начальный адрес массива и значение x, умноженное на размер элемента. Если бы индексация начиналась с 1, пришлось бы каждый раз вычитать 1 из индекса, что добавило бы лишние вычисления.

Итак, с помощью индекса можно получить доступ к любому элементу последовательности по его позиции. На изображении выше в переменной

x хранится строка, где каждый символ пронумерован (имеет свой индекс). Благодаря этому мы можем манипулировать одним или несколькими символами, вызывая их по индексу.

Листинг 7.26

```
x = "Каждый символ - это отдельный дом с порядковым номером"
#    0123456789...                                  ...51
print(x) # Команда выведет на экран всю строку
print(x[5]) # Выведет только символ с индексом 5 (буква "й")
```

Вместе с тем расчет индекса происходит также справа налево, только отсчет начинается с -1. То есть последний символ предложения (буква "м") имеет индекс -1.

Листинг 7.27

```
print(x[-54]) # Выведет букву "К"
```

Таким образом, если неизвестна длина строки или она постоянно меняется, но есть необходимость узнать последний символ, можно вызвать его по индексу -1. Предпоследний символ — по индексу -2 и т.д. до первого символа.

Листинг 7.28

```
x = "Еще одно предложение"
print(x[-1]) # Вывели последний символ – букву "е"
```

Такой подход делает код более гибким, так как нам не нужно предварительно вычислять длину строки функцией len. Это особенно полезно при работе с динамическими данными, которые могут меняться во время выполнения программы.

Срезы — позволяют "отрезать" часть подпоследовательности из более крупных последовательностей. Это как вырезать кусок из пирога: вы выбираете начало и конец отрезка и получаете именно эту часть.

То есть из предыдущего примера мы можем отрезать небольшую часть и продолжать работать с ней:

Листинг 7.29

```
x = "Еще одно предложение"
#      0123456789..
print(x[4:8]) # Выведет: одно
```

В данном примере из содержимого переменной мы отображаем символы от 4 до 7. Обратите внимание, что последний символ никогда не включается в последовательность (отсчет до него). По факту отображаются символы с 4 по 7 включительно.

Если не указывать один из параметров среза, то будут выводиться символы до конца строки или с самого начала.

Листинг 7.30

```
x = "Еще одно предложение"
#    0123456789..
print(x[4:]) # Выведет весь текст, начиная с 4 символа
print(x[:15]) # Выведет весь текст с 0 до 15 символа
```

Если не указывать ни один из параметров, отобразится вся строка.

Листинг 7.31

```
x = "Еще одно предложение"
#    0123456789..
print(x[:]) # Отобразится вся строка
```

Вместе с тем можно извлекать символы с определенным шагом, например каждый второй или третий. Для этого в квадратных скобках нужно указать третий параметр после двоеточия.

[::2] — с начала до конца строки будет взят каждый второй символ;

[3::4] — с третьего символа до конца строки отобразится каждый четвертый;

[:13:3] — с начала строки до 13-го символа будет взят каждый третий символ;

[::-1] — отобразим строку наоборот.

Листинг 7.32

```
x = "Еще одно предложение"
print(x[::-1]) # ежнежолдерп ондо  ещЕ
```

Благодаря срезам мы можем заменить часть содержимого переменной на новое:

Листинг 7.33

```
x = "Еще одно предложение" # Старое содержимое
x = x[:4] + "какое-то" + x [8:]
print(x) # Выведет: Еще какое-то предложение
```

В примере выше мы берем срезы из начала и конца предложения и с помощью конкатенации составляем новое предложение, вставляя между ними слово "какое-то".

7.4. Методы строк

Методы — это функции, связанные с определенным типом объекта. Они являются специфичными для каждого отдельного типа. Методы строк — это функции, встроенные в объекты типа строка.

Методы строк позволяют выполнять различные операции над строками, такие как поиск подстрок, преобразование регистра, форматирование и многое другое. Каждый метод выполняет определенную задачу и возвращает новый объект строки с внесенными изменениями.

Методов существует довольно большое количество, поэтому ниже мы рассмотрим только наиболее часто используемые и полезные. С полным списком методов можно ознакомиться в официальной документации Python.

Чтобы применить метод к объекту, нужно записать данный объект, поставить точку и написать метод, который следует применить. Так как метод — это, по сути, функция, многие из них могут принимать аргументы, которые записываются в скобки. Даже если мы не собираемся вводить аргументы, нужно не забывать ставить пустые скобочки.

Метод **upper()** — используется для преобразования всех букв в строке в верхний регистр.

Листинг 7.34

```
text = "привет, мир!"
print(text.upper()) # Результат: ПРИВЕТ, МИР!
```

Стоит иметь в виду, что методы не изменяют исходную строку, а отображают ее с внесенными изменениями. То есть, если мы снова обратимся к переменной, то увидим, что в ней по-прежнему хранится первоначальный текст с маленькой буквы.

Листинг 7.35

```
text = "привет, мир!"
print(text.upper()) # Результат: ПРИВЕТ, МИР!
print(text) # Результат: привет, мир!
```

Если вы хотите изменить исходную строку, нужно присвоить результат метода к этой переменной.

Листинг 7.36

```
text = "привет, мир!"
text = text.upper() # Перезаписали переменную с применением метода
print(text) # Результат: ПРИВЕТ, МИР!
```

Метод **lower()** — работает противоположно предыдущему методу. Он преобразует верхний регистр в нижний.

Листинг 7.37

```
text = "ПРИВЕТ, МИР!"
text = text.lower() # Перезаписали переменную с применением метода
print(text) # Результат: привет, мир!
```

Метод **title()** — делает первую букву каждого слова заглавной.

Листинг 7.38

```
text = "ПРИВЕТ, МИР!"
text = text.title()
print(text) # Результат: Привет, Мир!
```

Метод **capitalize()** — делает заглавной только первую букву строки.

Листинг 7.39

```
text = "ПРИВЕТ, МИР!"
text = text.capitalize()
print(text) # Результат: Привет, мир!
```

Метод **find()** — используется для поиска элемента внутри строки. Он возвращает индекс первого вхождения найденного элемента. Если элемент не найден, возвращается значение -1. В качестве аргумента в скобках необходимо указать, что именно нужно искать.

Листинг 7.40

```
text = "Это пример текста, в котором ищем элемент"
index = text.find("т") # Ищем букву "т"
print(index) # Результат: 1
```

В данном примере первое совпадение буквы "т" было найдено на первой позиции (индекс 1). Мы также можем искать совпадения на определенном отрезке текста, указав в качестве второго и третьего аргументов индексы начала и конца поиска.

Листинг 7.41

```
text = "Это пример текста, в котором ищем элемент"
index = text.find("т",16, 25) # Диапазон поиска — индексы с 16 по 25
print(index) # Результат: 23 — буква "т" в слове "котором"
```

Помимо одного символа, в качестве аргумента можно передать более длинный текст и узнать, где начинается совпадение.

Листинг 7.42

```
text = "Это пример текста, в котором ищем элемент"
index = text.find("текста")
print(index) # Результат: 11
```

Совпадение слова "текста" начинается на 11-й позиции.

Метод **rfind()** — делает то же самое, только ищет символы справа налево (*r* — сокращение от слова *right*). То есть выводит индекс первого совпадения от правого края строки.

Листинг 7.43

```
text = "Это пример текста, в котором ищем элемент"
index = text.rfind("е")
print(index) # Результат: 38 — буква "е" в слове "элемент"
```

Метод **index()** — работает так же, как и ***find***, однако, если символ не будет найден, то вместо ответа -1 будет ошибка, и программа завершит работу.

Листинг 7.44

```
text = "Это пример текста, в котором ищем элемент"
index = text.index("я")
print(index) # Результат: Ошибка! Подстрока не найдена
```

Метод **isalpha()** — проверяет, состоит ли строка только из букв. Возвращает ответ в виде булевых значений True/False. Если в строке есть хотя бы один пробел или цифра, ответ будет False.

Листинг 7.45

```
text = "Это пример текста, в котором ищем элемент"
index = text.isalpha()
print(index) # Результат: False (присутствуют пробелы)
```

Метод **isdigit()** — проверяет, состоит ли строка только из цифр. Также возвращает ответ True или False, если кроме цифр есть что-то еще. Данный метод может быть полезен, когда мы предлагаем пользователю ввести целое число и перед преобразованием (int) можно проверить, является ли введенное значение целым числом.

Листинг 7.46

```
text = "1234567890"
index = text.isdigit()
print(index) # Результат: True
```

Метод **isalnum()** — проверяет, состоит ли строка только из букв и цифр. Наличие пробелов и других символов вернет ответ False. Можно использовать, например, при проверке введенного пользователем значения, в котором запрещается вводить какие-либо спецсимволы.

Листинг 7.47

```
text = "qwerty & 12345"
index = text.isalnum()
print(index) # Результат: False — есть пробелы и спецсимвол
```

Метод **startswith(sub[, start[, end]])** — проверяет, начинается ли строка с указанного символа или нескольких символов. Также может включать в себя стартовый и конечный аргументы как метод *find*, для поиска совпадения в указанном отрезке.

Листинг 7.48

```
text = "Это пример текста, в котором ищем элемент"
index = text.startswith("пример",4,30)
print(index) # Результат: True
```

В примере выше берется отрезок начиная с 4-го индекса и заканчивая 29-м и проверяется, начинается ли отрезок со слова "пример".

Метод **endswith(suffix[, start[, end]])** — проверяет, заканчивается ли строка указанным символом. Тоже может принимать дополнительные аргументы в качестве начала и конца отрезка, в котором следует делать сравнение.

Листинг 7.49

```
text = "Это пример текста с расширением.txt"
if text.endswith(".txt"): # Если True
    print("Файл имеет расширение.txt")
```

Листинг 7.50

```
url = "https://www.example.com"
if url.endswith(".com", 8): # Начинаем проверку с 8-го символа
    print("Это веб-адрес")
```

Метод **replace(old, new[, count])** — заменяет все вхождения одной подстроки на другую. В качестве первого аргумента *old* нужно указать старый текст, во втором *new* — новый, который заменит собой старый. Если в тексте необходимо произвести несколько замен, в необязательном аргументе *count* указывается количество замен.

Листинг 7.51

```
text = "Привет, мир! Привет, Python!"
new_text = text.replace("Привет", "Здравствуйте")
print(new_text) # Результат: Здравствуйте, мир! Здравствуйте,
Python!
```

Заменяем два слова из трех:

Листинг 7.52

```
text = "Привет, мир! Привет, Python! Привет, Python!"
new_text = text.replace("Привет", "Здравствуйте", 2 )
print(new_text) # Результат: Здравствуйте, мир! Здравствуйте, Python!
Привет, Python!
```

Метод **strip()** — используется для удаления пробельных символов (пробелы, табуляции, символы новой строки) в начале и конце строки. Это полезно для очистки данных, особенно когда вы работаете с пользовательским вводом или текстовыми файлами.

Листинг 7.53

```
text = "  Привет, мир!  " # Лишние пробелы с двух сторон
clean_text = text.strip()
print(clean_text) # Результат: Привет, мир!
```

Для удаления конкретных символов необходимо указать их в скобках:

Листинг 7.54

```
text = "###Привет, мир!###"
clean_text = text.strip("#")
print(clean_text) # Результат: Привет, мир!
```

Существуют также дополнительные методы **lstrip()** и **rstrip()**, которые удаляют пробелы только слева или справа.

Метод **split()** — используется для разделения строки на список подстрок по указанному разделителю. Это очень полезный инструмент, когда нужно разбить строку на отдельные слова, фразы или элементы. Метод также включает в себя два необязательных аргумента **sep** и **maxsplit**, которые отвечают за разделитель и максимальное количество разделений.

Листинг 7.55

```
text = "Привет, мир! Привет, Python!"
words = text.split()
print(words) # Результат: ['Привет,', 'мир!', 'Привет,', 'Python!']
```

Вспомним про функцию len(), которая позволяет подсчитывать количество символов в строке. Совместив ее с методом ***split***, мы можем посчитать количество слов в тексте. Для этого метод ***split*** нужно обернуть в функцию ***len***.

Листинг 7.56

```
text = "Привет, мир! Привет, Python!"
words = len(text.split())
print(words) # Результат: 4
```

Если не указывать аргумент ***sep***, метод разделяет текст на отдельные слова, когда встречает пробел. Однако, если мы укажем другой символ, разделение будет происходить по этому символу:

Листинг 7.57

```
text = "Привет, мир! Привет, Python!"
words = text.split("и") # Разделяем текст, когда встречается буква "и"
print(words) # Результат: ["Пр", "вет, м", "р! Пр", "вет, Python!"]
# Разделено на 4 блока в кавычках через запятую
```

Второй аргумент ***maxsplit*** указывает на количество разделений, которое необходимо сделать по символу, указанному в первом аргументе ***sep***:

```
text = "Привет, мир! Привет, Python!"
words = text.split("и", 2) # Разделяем текст только на первых двух "и"
print(words) # Результат: ["Пр", "вет, м", "р! Привет, Python!"]
# Разделено на 3 блока в кавычках через запятую
```

Метод **join()** — выполняет обратную операцию к ***split***. Он объединяет элементы последовательности (например, списка) в одну строку, используя указанный разделитель.

Листинг 7.59

```
words = ["Привет", "мир", "!"] # Список с тремя словами
sentence = " ".join(words) # Соединить список с помощью пробела
print(sentence) # Результат: Привет мир !
```

В примере выше у нас есть список, в котором через запятую хранятся три слова (подробнее со списками мы познакомимся в следующем разделе). Метод ***join***, используя указанный пробел "_", соединяет содержимое списка в одну строку. Вместо пробела можно указать запятую или другие символы.

Листинг 7.60

```
data = ['Иван', 'Петров', '30']
csv_data = ','.join(data)
print(csv_data) # Результат: Иван,Петров,30
```

Листинг 7.61

```
lines = ['Первый элемент', 'Второй элемент', 'Третий элемент']
multiline_text = '\n'.join(lines)
print(multiline_text)
# Эскейп-последовательность \n перенесет следующий текст на новую
строку
# Первый элемент
# Второй элемент
# Третий элемент
```

Метод **format()** — используется для форматирования строк, то есть для вставки значений переменных в строку по определенным шаблонам.

Листинг 7.62

```
name = "Иван"
age = 30
greeting = "Привет, {}! Тебе {} лет.".format(name, age)
print(greeting) # Результат: Привет, Иван! Тебе 30 лет
```

В фигурные скобки будет подставлено содержимое переменных, которые указаны в качестве аргументов метода.

Метод **count()** — используется для подсчета количества вхождений определенного элемента в списке, строке или другой последовательности.

Листинг 7.63

```
text = "Это пример текста"
count_letter = text.count("т") # Количество букв "т" в строке
print(count_letter) # Результат: 3
```

Метод ***count*** обязательно должен иметь в круглых скобках аргумент (что именно искать и считать). Также можно указать диапазон поиска с помощью индексов:

Листинг 7.64

```
text = "Это пример текста"
count_letter = text.count("т", 3, 14) # Количество букв "т" в строке
print(count_letter) # Результат: 1 вхождение буквы "т" в диапазоне 3-14
```

Метод **center()** — используется для центрирования строки внутри поля заданной длины, заполняя оставшееся пространство указанным заполнителем (по умолчанию пробелом).

Листинг 7.65

```
text = "Привет, мир!"
centered_text = text.center(20)
print(centered_text) # Результат: Привет, мир!
```

В этом примере строка "Привет, мир!" центрируется в поле шириной 20 символов, заполняя пустые места пробелами.

Листинг 7.66

```
text = "Python"
centered_text = text.center(10, '*')
print(centered_text) # Результат: **Python**
```

Здесь строка "Python" центрируется в поле шириной 10 символов, а пустые места заполняются звездочками.

Если ширина меньше длины исходной строки, метод вернет исходную строку без изменений. Заполнитель может быть любой строкой, но обычно используются одиночные символы. Будет полезно, когда необходимо выровнять текст или заголовок по центру в заданных границах.

Метод **rjust()** — используется для выравнивания строки вправо внутри заданного поля, заполняя пустые позиции слева указанным символом (по умолчанию пробелом).

Листинг 7.67

```
text = "Python"
right_aligned = text.rjust(10)
print(right_aligned) # Результат: Python
```

В примере выше строка "Python" выравнивается вправо в поле шириной 10 символов, а слева заполняется пробелами. Аналогично методу ***center*** пробел можно заменить на другой символ.

Метод **ljust()** — работает так же, как и ***rjust***, но заполняет пустое пространство поля с правой стороны.

Листинг 7.68

```
text = "Python"
left_aligned = text.ljust(10, '^')
print(left_aligned) # Результат: Python^^^^
```

Метод **expandtabs()** — используется для преобразования символов табуляции (\t) в строке на определенное количество пробелов. Это может быть полезно при работе с текстовыми файлами, где табуляция используется для выравнивания данных.

Листинг 7.69

```
text = "Имя\tВозраст\tГород"
expanded_text = text.expandtabs(2)
print(expanded_text) # Результат: Имя Возраст Город
```

В этом примере каждый символ табуляции будет заменен на 2 пробела, что приведет к более читаемому формату.

Метод **encode(encoding, errors)** — используется для преобразования строки (текста) в последовательность байтов. Это может понадобиться, когда есть взаимодействие с другими системами, которые работают с байтовыми данными.

Листинг 7.70

```
text = "Привет, мир!"
bytes_utf8 = text.encode('utf-8')
print(bytes_utf8) # Результат: b'\xd0\x9f\...\xd1\x80!'
```

1. Аргумент **encoding** указывает кодировку для преобразования. Если ничего не указывать, по умолчанию используется utf-8.
2. Второй аргумент **errors** отвечает за обработку ошибок.

Листинг 7.71

```
text = "Привет, мир!"
bytes_ascii = text.encode('ascii', errors='ignore')
print(bytes_ascii) # Результат: b', !' (запятая и восклицательный знак)
```

Здесь мы пытаемся закодировать текст в ASCII, но так как не все символы поддерживаются этой кодировкой, будет получена ошибка. С помощью аргумента ***ignore*** можно пропустить неподдерживаемые символы. Другие варианты поведения при ошибке: ***strict*** и ***replace***.

> Обратите внимание, что методы можно применять сразу к значениям, которые будут введены пользователем!

Листинг 7.72

```
text = input().upper() # Применяем метод к введенному значению
print(text)
```

В данном примере введенный текст с помощью метода ***upper*** будет сохранен в верхнем регистре.

Также мы можем применять последовательно серию из нескольких методов:

Листинг 7.73

```
text = input().capitalize().strip().center(14)
print(text)
```

Здесь введенный текст будет исправлен методом ***capitalize***, если он написан с маленькой буквы или капсом. Затем метод ***strip*** отбросит все пробелы слева и справа, если таковые имеются. Далее метод center центрирует текст в поле на 14 символов.

7.4.1. Практические задания

1. Предложите пользователю ввести любой текст. Затем с помощью определенного метода выясните, состоит ли введенный текст только из букв. Выведите полученный результат на экран.
2. Напишите аналогичную программу, которая будет проверять, состоит ли введенное значение только из цифр.
3. Предложите пользователю ввести логин, который должен состоять только из букв и цифр. Проверьте введенное значение и сообщите пользователю, принят его логин или нет.
4. У нас есть текст со следующим содержанием:

   "Мой кот очень любит играть с клубком. Но иногда кот спит весь день." Замените все вхождения слова "кот" на слово "пес" и отобразите результат.
5. Предложите пользователю ввести e-mail адрес и проверьте, корректно ли он введен. Это можно выяснить по наличию символа "@" и окончанию .com или .ru.
6. Предложите пользователю ввести любой текст, затем подсчитайте количество слов в тексте и выведите результат.

7.5. Регулярные выражения

Регулярные выражения — это инструмент для работы с текстовыми данными, благодаря которому можно производить парсинг по массиву текста, находить в нем определенные значения и проводить с ним различные операции, такие как поиск и сортировка по заданным параметрам, замена одних частей текста на другие и т.д.

Для применения регулярных выражений в Python используется встроенный модуль **re**. То есть для работы данного модуля его нужно так же импортировать, как модуль **math**, которым мы пользовались ранее для некоторых математических операций. Модуль **re** включает в себя методы, позволяющие эффективно искать и изменять текст, используя определенные шаблоны.

К примеру, у нас есть задача найти определенное слово или словосочетание в большом массиве текста. Например, необходимо выяснить, сколько раз упоминается персонаж Болконский в произведении "Война и мир". Само слово "Болконский" станет нашим регулярным выражением, которое можно найти, используя операторы и методы из модуля re, а затем подсчитать точное количество упоминаний.

Листинг 7.80

```
import re # Импортируем модуль
text = "Список: хлеб пшеничный, хлеб ржаной, хлебный продукт"
# Шаблон поиска
pattern = r"\bхлеб\b" # Ищем слово "хлеб" как отдельное слово
matches = re.findall(pattern, text)
count = len(matches) # Считаем количество совпадений
# Выводим результат
if count > 0:
    print(f"Слово 'хлеб' встречается {count} раз(а).")# 2 раза
else:
    print("Слово 'хлеб' не найдено.")
```

В примере выше в строке *matches = re.findall(pattern, text)* происходит следующее:

- В переменную *matches* сохраняется результат работы метода ***findall*** из модуля ***re***.
- Метод имеет шаблон, сохраненный в переменной *pattern*, по которому будут находиться совпадения.
- А сам поиск по шаблону будет производиться в тексте, который сохранен в переменной *text*.

Шаблон в переменной *pattern (r"\bхлеб\b")*, помимо искомого слова, включает в себя специальные символы (операторы), благодаря которым отсекаются похожие выражения. Например, прилагательное "хлебный" имеет корень "хлеб", что приведет к совпадению, поэтому, чтобы отсеять похожие слова, используется символ \b, означающий, что это граница слова.

Символ обратного слеша может восприниматься Пайтоном как экранирование (эскейп-последовательность), поэтому перед кавычками мы пишем букву *r*, сообщающую интерпретатору, что все символы в кавычках нужно воспринимать буквально (сырая строка).

7.5.1. Операторы в регулярных выражениях

Полный список операторов и их значений может варьироваться в зависимости от конкретной реализации регулярных выражений. Ниже приведен список наиболее распространенных и универсальных символов.

\b — обозначает границу слова. Это может быть его начало или конец, а также любой другой символ, не являющийся буквой, цифрой или подчеркиванием.

\B — противоположный предыдущему символу. То есть с помощью этого символа мы можем найти слово "хлебный" и другие однокоренные слова из примера выше.

. (точка) — любой символ, кроме новой строки. В примере ниже точка применяется три раза, чтобы найти последовательность из трех любых символов.

Листинг 7.81

```
import re
text = "Это пример текста с разными словами: abc, 123, xyz, а-я."
pattern = r"\b...\b"  # Ищем последовательности длиной в три символа
matches = re.findall(pattern, text)
print(matches) # Результат: ['Это', ' с ', 'abc', '123', 'xyz', ', а']
```

^ — начало строки. Символ используется для того, чтобы указать, что совпадение должно начинаться с самого начала строки. Это может понадобиться при поиске определенных шаблонов в начале текста или при валидации данных.

Листинг 7.82

```
import re
text = "Начало текста. Середина текста. Конец текста."
pattern = r"^Начало"
matches = re.findall(pattern, text)
print(matches) # Результат: ['Начало']
```

$ — конец строки. Противоположно предыдущему, ищет окончание строки.

Листинг 7.83

```
import re
text = "Начало текста. Середина текста. Конец текста."
pattern = r"текста.$"
matches = re.findall(pattern, text)
print(matches) # Результат: ['текста.']
```

\A — применяется только к самому началу массива текста, в отличие от ^, который может соответствовать началу каждой строки.

\Z — соответствует только самому концу строки. В отличие от $, который может также соответствовать позиции перед символом новой строки в режиме многострочного поиска (с флагом re.MULTILINE). \Z всегда соответствует абсолютному концу текста.

\d в регулярных выражениях обозначает сокращение для набора символов, соответствующих **любой цифре** от 0 до 9. Это полезный символ при поиске чисел в тексте.

Листинг 7.84

```
import re
text = "Почтовый индекс: 12345"
pattern = r"\d{5}" # Находим последовательность из 5 чисел
matches = re.findall(pattern, text)
print(matches) # Результат: ['12345']
```

Листинг 7.85

```
import re
text = "Телефонный номер: 123-456-7890"
pattern = r"\d+" # Находим одну или более цифр подряд
matches = re.findall(pattern, text)
print(matches) # Результат: ['123', '456', '7890']
```

\D — ищем любой символ, **кроме цифры**.

Листинг 7.86

```
import re
text = "Телефонный номер: 123-456-7890"
pattern = r"\D+" # Находим один или более нецифровых символов подряд
matches = re.findall(pattern, text)
print(matches) # Результат: ['Телефонный номер: ', '-', '-']
```

\s — ищем любой пробельный символ (пробел, табуляция и т.д.).

Листинг 7.87

```
import re
text = "  Привет, мир!  "
pattern = r"\s+" # Ищем один или более пробельных символов подряд
matches = re.findall(pattern, text)
print(matches) # Результат: ['  ', ' ', '  ']
```

\S — любой не пробельный символ.

Листинг 7.88

```
import re
text = "  Привет, мир!  "
pattern = r"\S+" # Находим один или более не пробельных
символов подряд
matches = re.findall(pattern, text)
print(matches) # Результат: ['Привет,', 'мир!']
```

\w — любой алфавитно-цифровой символ или подчеркивание (_).

```
import re
text = "Небольшой текст и цифры  123_."
pattern = r"\w+" # Ищем одну или более букв, цифр или подчеркиваний
подряд
```

```
matches = re.findall(pattern, text)
print(matches) # Вывод: ['Небольшой', 'текст', 'и', 'цифры',
'_123_']
```

\W — противоположно предыдущему выражению. Означает любой символ, кроме алфавитно-цифрового или подчеркивания.

Листинг 7.90

```
import re
text = "Небольшой текст и цифры _123_."
pattern = r"\W"
matches = re.findall(pattern, text)
print(matches) # Результат: [' ', ' ', ' ', ' ', '.']
```

***** — означает ноль или более повторений предыдущего символа или группы символов. Звездочка может быть полезна для создания шаблонов, при поиске повторений, количество которых может изменяться.

Листинг 7.91

```
import re
text = "551235"
pattern = r"5*" # Ищем пятерку или несколько пятерок подряд
matches = re.findall(pattern, text)
print(matches) # Результат: ['55', '', '', '', '5', '']
```

+ — находит одно или несколько повторений предыдущего элемента. То есть паттерн, которому предшествует +, должен встречаться в строке хотя бы один раз. Примеры его использования можно увидеть в нескольких примерах выше.

Отличие плюса (+) от звездочки (*) заключается в том, что совпадение должно присутствовать хотя бы один раз. Тогда как звездочка допускает **ноль** и более совпадений (то есть паттерн может встречаться любое количество раз, включая ни разу).

? — символ означает, что паттерн, которому предшествует знак вопроса (?), может встречаться в строке только один раз либо вовсе отсутствовать.

Листинг 7.92

```
import re
text = "солнце сонце"
pattern = r"сол?нце" # Буква 'л' может присутствовать или
отсутствовать
matches = re.findall(pattern, text)
print(matches) # Результат: ['солнце', 'сонце']
# Находим все совпадения слова,
# даже если текст написан с ошибками
```

{n} — указывает количество повторений для предыдущего символа.

Листинг 7.93

```
import re
text = "Номер телефона: 89123456789"
pattern = r"\d{11}" # Находим последовательность из 11 цифр
matches = re.findall(pattern, text)
print(matches) # Результат: ['89123456789']
```

{n,m} — от n до m повторений предыдущего элемента.

Листинг 7.94

```
import re
text = "IP-адрес: 192.167.0.1"
# Каждый октет IP-адреса должен включать в себя от 1 до 3 цифр
pattern = r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
matches = re.findall(pattern, text)
print(matches) # Результат: ['192.167.0.1']
```

[abc] — ищет любой символ из набора a, b или c. Квадратные скобки в регулярных выражениях определяют набор символов.

Листинг 7.95

```
import re
text = "Пример текста"
pattern = r"[иеа]" # Ищем гласные буквы
matches = re.findall(pattern, text)
print(matches) # Результат: ['и', 'е', 'е', 'а']
```

[^abc] — противоположно предыдущей конструкции находит любые символы, кроме указанных в квадратных скобках.

[a-z] — любая строчная буква от a до z.

Листинг 7.96

```
import re
text = "эюя wifi"
pattern = r"[a-z]" # Находим латинские буквы в диапазоне от a до z
matches = re.findall(pattern, text)
print(matches) # Результат: ['w', 'i', 'f', 'i']
```

[^a-z] — находим все, кроме указанного диапазона символов.
[0-9] — находим цифры в диапазоне от 0 до 9.

( ) — скобки в регулярных выражениях используются для группировки частей шаблона. Это позволяет извлекать отдельные части совпадения, где каждая группа в скобках становится отдельным элементом в результате поиска.

Листинг 7.97

```
import re
text = "Иван Петров, Москва"
pattern = r"(\w+) (\w+)" # Две группы: имя и фамилия
matches = re.search(pattern, text)
if matches:
    print("Имя:", matches.group(1))
    print("Фамилия:", matches.group(2))
```

(\w+) создает две группы, каждая из которых ищет одну или более букв, цифр или подчеркиваний. match.group(1) возвращает первое совпадение (имя), а match.group(2) — второе (фамилия).

Пример с извлечением даты из строки:

Листинг 7.98

```
import re
text = "Сегодня 2024-12-24"
# Три группы: первая из 4 цифр, вторая и третья из 2 цифр
pattern = r"(\d{4})-(\d{2})-(\d{2})"
matches = re.search(pattern, text)
if matches:
    # В каждую переменную сохраняется результат отдельной группы
    year, month, day = matches.groups()
    print(year, month, day) # Результат: 2024 12 24
```

\1, \2 — обратные ссылки в регулярных выражениях. Они применяются, чтобы повторно использовать части шаблона, находящиеся в скобках (группах). Данная конструкция может понадобиться при повторении подшаблонов. Например, для проверки на палиндромы или для поиска дубликатов слов.

Листинг 7.99

```
import re
text = "Дата: 2024-12-24"
pattern = r"(\d{4})-(\d{2})-(\d{2})"
new_format = r"\2.\1.\3" # Поменяли группы местами
new_text = re.sub(pattern, new_format, text)
print(new_text) # Результат: 12.2024.24
```

Комбинирование:

Листинг 7.100

```
import re
text = "Абвг Aab1s2d"
```

```
# Находим символы в диапазонах a-z,A-Z,0-9
pattern = r"[a-zA-Z0-9]"
matches = re.findall(pattern, text)
print(matches) # Результат: ['A', 'a', 'b', '1', 's', '2', 'd']
```

Обратите внимание, что при комбинировании нескольких вариантов поиска между регулярными выражениями не ставятся запятые. Вместе с тем стоит помнить о важности регистра. При необходимости найти все вхождения определенных слов в параметрах поиска нужно указать их в верхнем и нижнем регистре.

7.5.2. Основные методы модуля *re*

Модуль **re** включает в себя ряд методов, которые применяются для разных задач. В примерах выше, при разборе операторов, мы в основном использовали метод ***re.findall***. Давайте разберемся, в каких случаях стоит применять те или иные методы.

Метод **re.match()** используется для поиска совпадения шаблона в **начале** заданной строки. Если совпадение найдено, возвращается объект Match, с информацией о найденном совпадении. В противном случае возвращается None.

Листинг 7.101

```
import re
text = "Массив, в котором ищем совпадения"
pattern = r"Массив"
match = re.match(pattern, text)
if match:
    print("Совпадение найдено:", match.group())
else:
    print("Совпадение не найдено")
# Результат: Совпадение найдено: Массив
```

Метод **re.search()** используется для поиска первого вхождения шаблона в строке. В отличие от re.match(), совпадение ищется во всем массиве текста, пока не будет встречено первое совпадение.

Листинг 7.102

```
import re
text = "Массив, в котором ищем совпадения"
pattern = r"ищем"
match = re.search(pattern, text)
if match:
    print("Совпадение найдено:", match.group())
else:
    print("Совпадение не найдено")
# Результат: Совпадение найдено: ищем
```

Метод **re.findall()** используется для поиска **всех** непересекающихся совпадений шаблона в строке и возвращает список всех совпавших подстрок.

Листинг 7.103

```
import re
text = "Пшеничный хлеб, ржаной хлеб, отрубной хлеб"
pattern = r"хлеб"
matches = re.findall(pattern, text)
print(matches) # Результат: ['хлеб', 'хлеб', 'хлеб']
```

Метод **re.split()** применяется для разделения строки на отдельные подстроки по указанному выражению. На выходе получаем список из подстрок.

Листинг 7.104

```
import re
text = "Строка, разделенная по пробелам"
words = re.split(r"\s+", text)
print(words)
# Результат: ['Строка,', 'разделенная', 'по', 'пробелам']
```

Метод **re.sub()** применяется для замены всех непересекающихся совпадений в строке на указанный символ или подстроку. В результате получаем обновленную строку с заменой слов или символов.

Листинг 7.105

```
import re
text = "Телефонный номер: 123-456-7890"
new_text = re.sub(r"\d", "*", text)
print(new_text) # Результат: Телефонный номер: ***-***-****
# Произошла замена номера на звездочки
```

7.5.3. Практические задания

1. Дана строка, содержащая имя пользователя и его номер телефона: "Иванов Семен Петрович: 89123456789".

   Необходимо, используя методы из модуля re и регулярных выражений, извлечь из этой строки телефонный номер, а затем отобразить в консоли.

2. На вход получены данные, включающие в себя текст и дату в виде "число, месяц, год":

   "Это событие произошло: 17-07-1777".

   С помощью регулярных выражений извлеките дату и выведите ее на экран.

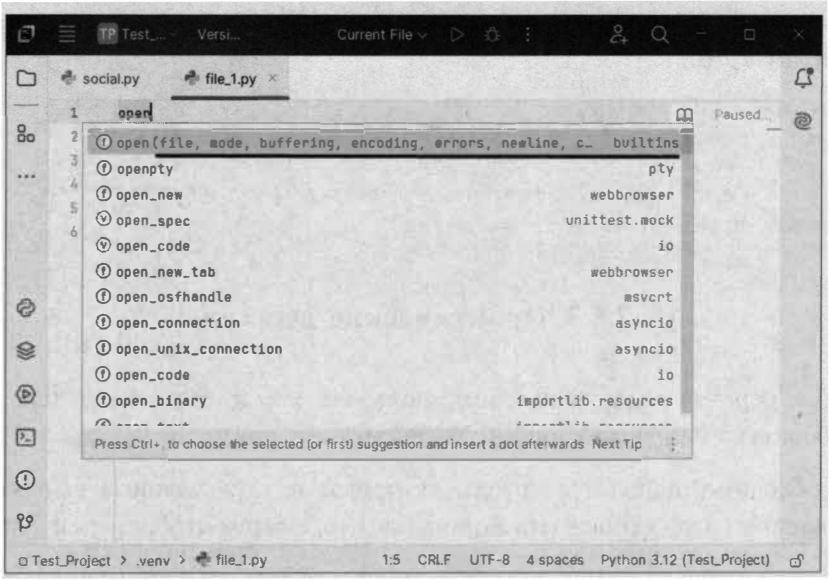
3. Замените все вхождения слова "кот" на слово "пес" в строке:

   "Наш кот любит сметану. Иногда кот спит на печи."

4. Создайте строку для ввода email-адреса. С помощью регулярных выражений проверьте на корректность введенную электронную почту. То есть почта должна содержать набор символов до знака @, а также набор символов после него. Далее должна присутствовать точка и набор символов для доменной зоны.

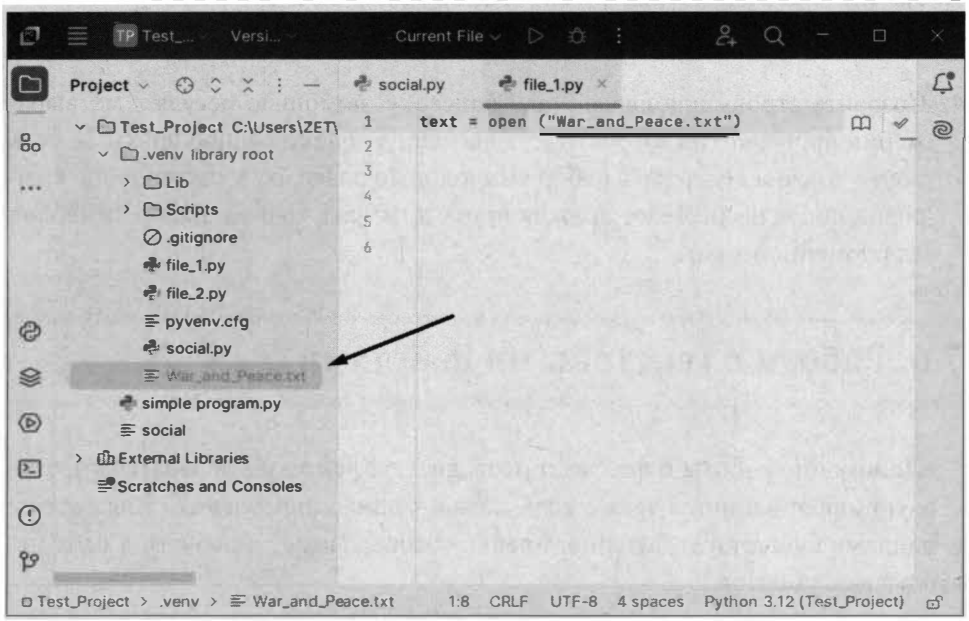
7.6. Работа с текстовыми файлами

Понимание работы с файлами позволяет эффективно обрабатывать текстовую информацию, а также создавать и модифицировать ее. Для работы с файлами существует функция **open()**, которая может включать в себя несколько аргументов.



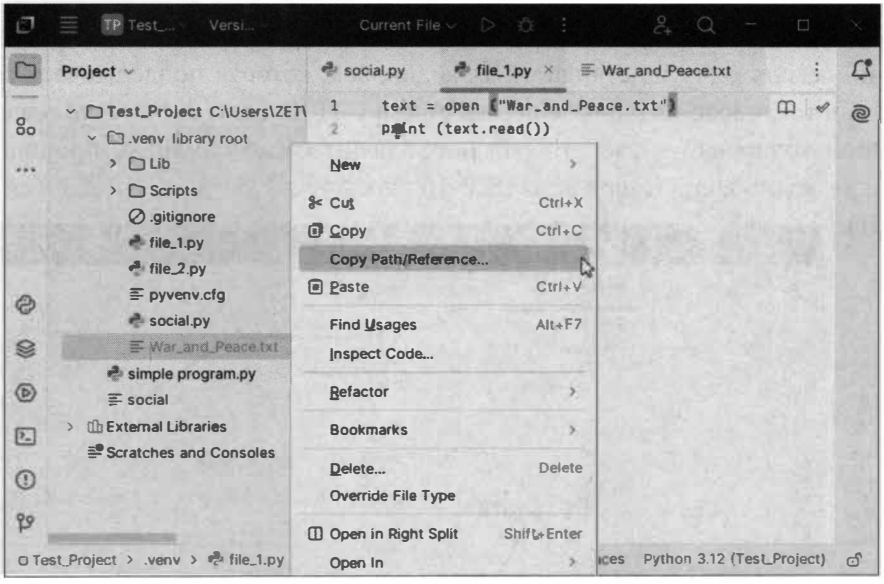
Изображение 7.8.

Первый аргумент **file** — позволяет прописать путь к файлу, с которым необходимо взаимодействовать. Если файл находится в той же директории, что и программа, можно писать не полный путь к файлу, а только его название и расширение.



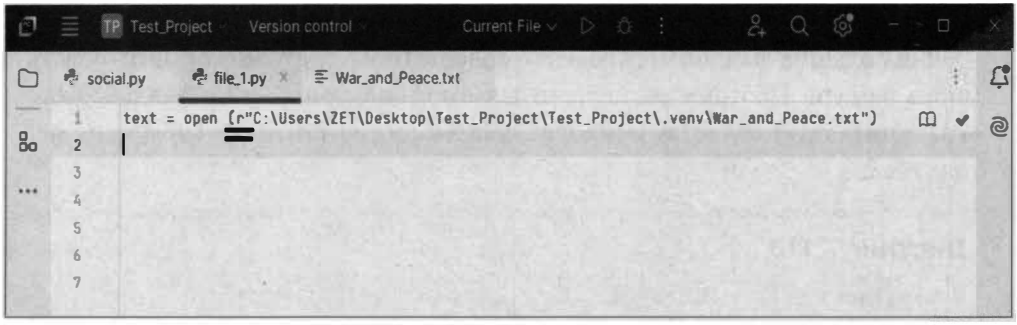
Изображение 7.9.

На изображении выше мы создали переменную ***text***, которая будет хранить результат работы функции ***open***. Функция, в свою очередь, обращается к указанному файлу. Кроме того, можно прописать полный путь к файлу. Для этого щелкните правой кнопкой мыши по файлу и выберите пункт Copy Path. После этого откроется дополнительное окно, из которого можно скопировать полный путь.



Изображение 7.10.

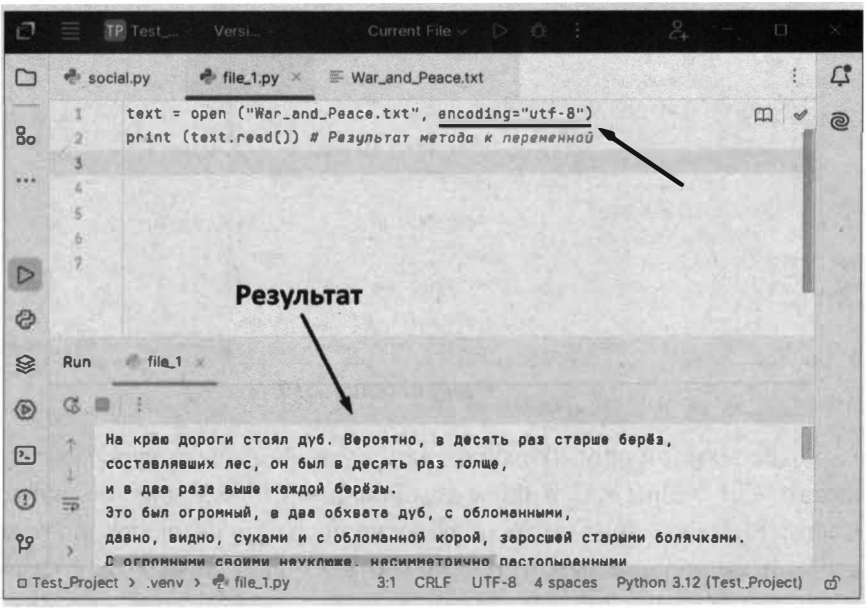
Более простой способ скопировать путь — навести курсором на файл и нажать Ctrl + Shift + C, а затем комбинацию Ctrl + V для вставки в нужное место. Также не стоит забывать об экранировании символов, поэтому перед строкой с прописанным путем нужно поставить символ *r*, создав сырую строку.



Изображение 7.11.

После открытия файла к нему можно применить разные методы. Метод **read** используется для чтения содержимого файла. Когда вы вызываете этот метод на объекте файла, он возвращает все содержимое в виде одной большой строки.

Стоит иметь в виду, что, если текст содержит символы, не входящие в латинский алфавит, мы можем получить ошибку или нечитаемый набор символов. Поэтому вторым аргументом **encoding** в функции open необходимо указать одну из стандартных кодировок, которая поддерживает символы, в том числе на кириллице, например UTF-7. Если при использовании данной кодировки у вас отображаются нечитаемые символы, пропишите другие кодировки. Например, UTF-16, Windows-1251 или cp1251 (си пи 1251).



Изображение 7.12.

Для больших файлов чтение всего содержимого сразу может потребовать много памяти. Поэтому размер считываемой информации из файла можно ограничить по количеству символов, указав соответствующий аргумент метода ***read***.

Листинг 7.110

```
text = open ("War_and_Peace.txt", encoding="utf-8")
print (text.read(100))  # Результат применения метода к переменной
```

В примере выше мы указали в скобках 100 символов, которые метод read должен прочитать. Обратите внимание, что Пайтон запоминает место, на котором остановилось чтение, поэтому если мы снова укажем отобразить 100 символов, будут выведены следующие символы после предыдущих.

```
text = open ("War_and_Peace.txt", encoding="utf-8")
print (text.read(100)) # Отобразятся первые 100 символов
print (text.read(100)) # Отобразятся следующие 100 символов
```

Если нам понадобится снова обратиться к первой части текста, необходимо вернуть чтение к началу с помощью метода **seek**.

Листинг 7.112

```
text = open ("War_and_Peace.txt", encoding="utf-8")
print (text.read(100)) # Отобразятся первые 100 символов
print (text.read(100)) # Отобразятся следующие 100 символов
text.seek(0) # Указали, к какой позиции откатиться
print (text.read(100)) # Снова вывели первые 100 символов
```

Как видно в примере выше, в скобках мы указали позицию возврата. Это может быть не только нулевая позиция, но и любая другая, в том числе та, до которой мы еще не добрались. В таком случае произойдет "скачок" вперед по тексту.

Следующий метод **readline** — считывает строку целиком.

Листинг 7.113

```
text = open ("War_and_Peace.txt", encoding="utf-8")
print (text.readline()) # Считывает строку целиком
print (text.readline()) # Следующая строка
```

В данном случае также сохранена позиция прочтения первой строки. Поэтому при повторном выводе функции ***print*** отобразится следующая строка.

Метод **readlines** — читает все строки файла и возвращает их в виде списка. Это может пригодиться при сортировке или форматировании каких-то данных, где каждая строка содержит определенную информацию, например в телефонной книге.

Листинг 7.114

```
text = open ("War_and_Peace.txt", encoding="utf-8")
spisok = text.readlines() # Сохраняем в переменную результат метода
print (spisok) # Отображаем список
```

Кроме чтения информации из файла, есть методы для записи новой информации или редактирования текущей. Для этих целей используется метод **write.** Но для начала давайте разберемся с режимами работы с файлом.

Ранее наш файл открывался в режиме чтения, поэтому что-то записать в него или удалить не получится. Для этого потребуется "перевести" его в соответствующий режим, указав аргумент **mode** в функции **open**.

Режимы определяют, каким образом будет открыт файл:

- **'r'** — чтение (сокращение от *read*), установлен по умолчанию, даже если не прописывать.
- **'w'** — запись (*write*) создает новый файл или перезаписывает существующий.
- **'a'** — добавление (*add*), добавляет данные в конец файла.
- **'x'** — создание, создает новый файл.
- **'t'** — открывает файл как текстовый документ.
- **'b'** — открывает файл в бинарном/двоичном режиме (например, для чтения изображений).
- **'r+'** — чтение и запись.
- **'a+'** — чтение и добавление.

Таким образом, для записи новой информации в файл нам потребуется следующий код:

Листинг 7.115

```
text = open ("War_and_Peace.txt", "w", encoding="utf-8")
text.write("Новый текст, который будет записан в файл")
```

Обратите внимание, что новый текст перезапишет старый. Если есть необходимость добавить новые данные к текущим, режим следует переключить, прописав аргумент "a". Он добавит указанный текст в самый конец файла.

Листинг 7.116

```
text = open ("War_and_Peace.txt", "a", encoding="utf-8")
text.write("Новый текст, который будет записан в файл")
```

Также стоит иметь в виду, что при работе в режиме записи или добавления невозможно работать с **режимом чтения** (read). Для его использования потребуется прописать соответствующие режимы r+ или a+.

Следующий метод **writelines** — используется для записи стразу целого списка строк в файл, без необходимости вызывать метод ***write*** для каждой строки по отдельности. Это значительно упрощает и ускоряет процесс записи больших объемов данных.

```
text = open ("War_and_Peace.txt", "a", encoding="utf-8")
# Сисок с д  а  дл эаиси
spisok = ['Первая строка\n', 'Вторая строка\n', 'Третья строка']
text.writelines(spisok)
```

В примере выше открывается файл в режиме добавления ("a"), в переменной *spisok* содержится список, состоящий из нескольких элементов. Затем файл из переменной *text* редактируется методом ***writelines***, что добавляет в него данные из списка.

Обратите внимание, что каждый элемент списка дополнен экранированным символом (\n), благодаря чему каждый из них будет размещен на новой строке.

Метод **tell** — используется для получения текущей позиции указателя (курсора) внутри файла. То есть того места, до которого был прочитан файл.

Листинг 7.118

```
text = open ("War_and_Peace.txt", "r", encoding="utf-8")
text.read(100) # Читаем первые 100 символов
position = text.tell() # Сохраняем в переменную позицию,
определенную методом tell
print(f"Текущая позиция: {position}") # Выводим результат
```

Метод **close** применяется для закрытия открытого файла. Если не закрыть файл после перезаписи, есть вероятность, что не освободится память, которую занимала эта информация. То есть произойдет утечка памяти. Вместе с тем могут быть проблемы при повторном открытии данного файла.

Листинг 7.119

```
text = open ("War_and_Peace.txt", "a+", encoding="utf-8")
text.write("Отредактированный текст")
text.close() # Закрыли файл
```

Также может произойти какая-то ошибка или сбой во время работы с файлом, и до команды ***close*** интерпретатор может не дойти. То есть файл останется открытым. Чтобы избежать подобных ситуаций, применяется альтернативный вариант.

Листинг 7.120

```
with open("War_and_Peace.txt", "a+", encoding="utf-8") as text:
    text.write("Это новая строка\n") # Добавили несколько строк
    text.writelines(["Строка 2\n", "Строка 3"]) # разными методами
```

Контекстный менеджер **with** автоматически закрывает файл после завершения блока, даже если возникнет исключение.

Это гарантирует, что файл будет закрыт корректно. То есть при открытии файла интерпретатор сразу узнает, что по завершении работы с ним его следует закрыть.

Удаление файлов осуществляется с помощью стандартной библиотеки **os**. Для этого существует несколько функций, каждая из которых предназначена для определенных случаев.

Функция **os.remove** используется для удаления обычных файлов. Она принимает в качестве аргумента путь к файлу.

Листинг 7.121

```
import os # Импортировали модуль
file_path = "путь/к/вашему/файлу.txt" # Переменная хранит путь к файлу
os.remove(file_path) # Удаляем файл по указанному пути
```

Функция **os.rmdir** используется при удалении пустых директорий. Если директория не пустая, будет выброшено исключение.

Листинг 7.122

```
import os
directory_path = "путь/к/пустой/директории"
os.rmdir(directory_path) # Удалили директорию
```

Перед удалением файла рекомендуется проверить его существование с помощью функции **os.path.exists**, чтобы избежать ошибок. Кроме того, у вашей программы должны быть необходимые права доступа для удаления директории или файла.

В большинстве операционных систем удаленные файлы помещаются в корзину. Для полного удаления из корзины могут потребоваться дополнительные действия. Если файл используется другим процессом, его удаление может быть невозможно.

Код с проверкой выглядит следующим образом:

Листинг 7.123

```
import os
file_path = "путь/к/вашему/файлу.txt"
if os.path.exists(file_path): # Если файл существует, то:
    os.remove(file_path) # Удаляем файл
    print(f"Файл {file_path} успешно удален.") # Подтверждение
else: # Если не найден
    print(f"Файл {file_path} не обнаружен.")
```

7.6.1. Практическое задание

1. Создайте новый текстовый файл и поместите в него любую информацию. Далее перенесите файл в любую выбранную папку и запомните путь к ней. Напишите скрипт, который откроет файл по указанному пути и допишет в конце, что файл был отредактирован. Затем можете снова открыть файл, чтобы убедиться в изменениях.

7.7. Работа с двоичными файлами

Двоичный файл — это файл, содержащий данные в нетекстовом формате, то есть в виде последовательности байтов. В отличие от текстовых файлов, где каждый символ представлен определенным кодом, в двоичных файлах данные хранятся в более компактном и эффективном виде, непосредственно отражая внутреннее представление данных в памяти компьютера.

Двоичные файлы бывают следующих типов:

1. Исполняемые файлы (EXE, BIN).
2. Изображения (JPEG, PNG).
3. Аудио- и видеофайлы (MP3, MP4).

4. Сжатые архивы (ZIP, RAR).

5. Сохраненные игры.

6. Базы данных.

Двоичные файлы обычно занимают меньше места на диске и быстрее загружаются, чем текстовые файлы, так как не содержат избыточной информации. Многие типы данных (изображения, звук, видео) имеют свои собственные форматы, которые оптимизированы для хранения и обработки конкретного типа информации. Кроме того, двоичные файлы позволяют сохранять сложные структуры данных, такие как массивы, объекты и пр., в их исходном виде.

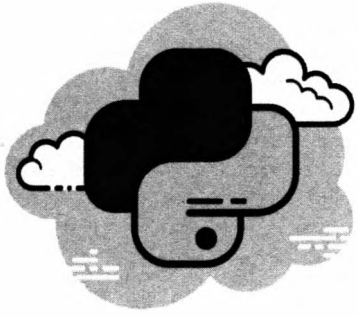
Для работы с двоичными файлами используется тот же механизм open(), что и для текстовых файлов, но с добавлением режима 'rb' для чтения и 'wb' для записи.

Листинг 7.125

```
with open("image.jpg", "rb") as foto:
    # Чтение всего содержимого файла в виде байтов
    data = foto.read()
with open("new_image.jpg", "wb") as foto2:
    # Запись данных в новый файл
    foto2.write(data)
```

В примере выше вместо image.jpg можно указать полный путь до изображения на сайте. Вместо new_image.jpg — указать директорию на компьютере, где следует перезаписать файл. Таким образом происходит сохранение изображения на вашем устройстве. Аналогично этому, но с некоторыми особенностями, происходит сохранение других типов файлов.

Обратите внимание, что для двоичных файлов понятие кодировки не применимо, так как данные хранятся в виде байтов. Структура данных в двоичном файле зависит от конкретного формата файла и может быть достаточно сложной. Для работы со сложными форматами файлов часто используются специализированные библиотеки.



Глава 8.

Структуры данных

8.1. Списки

Списки представляют собой упорядоченные коллекции элементов и позволяют хранить в одном месте взаимосвязанные данные. Они могут включать в себя практически любые типы элементов, например числа, строки, другие списки и даже функции.

Листинг 8.1

```
list_1 = [1, 2, 3, "hello", True] # Список с разными типами данных
list_2 = [[1, 2, 3], [a,b,c],['a','b','c']] # Список, содержащий
другие списки
list_3 = [] # Пустой список
```

Для сбора элементов в список их помещают в квадратные скобки с запятыми между элементами и присваивают уникальное название. Это похоже на переменную, которая ссылается не на один объект, а сразу на несколько. То есть несколько предметов хранятся в одной "коробке".

Аналогично текстовым данным над списками можно проводить определенные операции. Каждый элемент списка тоже имеет свой порядковый номер (индекс), благодаря чему можно подсчитать количество объектов, находящихся в нем.

Листинг 8.2

```
list_1 = [1, 2, 3, "hello", True]
print(len(list_1)) # Результат: 5 (элементов в списке)
```

В примере выше в списке хранятся пять отдельных элементов, каждый из которых разделен запятой. Обратите внимание, что строка "hello" является одним элементом списка, как и значение True.

Несколько разных списков можно сцеплять между собой:

Листинг 8.3

```
list_1 = [1, 2, 3]
list_2 = ["hello", True]
list_3 = list_1 + list_2 # Соединили списки в отдельный список
print(list_3) # Результат: [1, 2, 3, 'hello', True]
```

Элементы списка можно дублировать с помощью операции умножения:

Листинг 8.4

```
list_1 = [1, 2, 3] * 2
print(list_1) # Результат: [1, 2, 3, 1, 2, 3]
```

Используя индексы элементов, можно обращаться к определенному объекту:

Листинг 8.5

```
list_3 = [1, 2, 3, 'hello', True]
print(list_3[3]) # Результат: hello
```

В данном примере мы даем команду вывести на экран третий элемент [3] из списка list_3. Помним, что расчет индексов начинается с нуля, поэтому под третьим индексом стоит строка "hello".

Также мы можем взять срез из списка, для дальнейшей манипуляции с ним:

Листинг 8.6

```
list_3 = [1, 2, 3, 'hello', True]
list_4 = list_3[1:4] # Срез с первого до четвертого индексов
print(list_4) # Результат: [2, 3, 'hello']
```

В примере мы создали новый список, который будет содержать в себе срез элементов из предыдущего списка.

Следующая операция — это проверка наличия элемента в списке. Например, мы можем выяснить, есть ли в нашем списке указанный объект, и, если это так, выполнить соответствующее условие. Поиск вхождения осуществляется с помощью союза **in**.

Листинг 8.7

```
list_4 = [2, 3, 'hello']
if 3 in list_4: # Если тройка в list_4, то
    print("Тройка присутствует в списке")
else:
    print("Значение не обнаружено")
```

Если список состоит полностью из чисел, к нему можно применить операции для нахождения максимального и минимального числа, а также суммирование всех чисел.

Листинг 8.8

```
list_5 = [12, 32, 44, 536, 82, 26]
print(max(list_5)) # Нашли максимальное число
print(min(list_5)) # Нашли минимальное число
print(sum(list_5)) # Нашли сумму всех чисел
```

Для сортировки списка используется функция ***sorted***.

Листинг 8.9

```
list_5 = [12, 32, 44, 536, 82, 26]
print(sorted(list_5)) # Результат: [12, 26, 32, 44, 82, 536]
print(sorted(list_5, reverse=True)) # Результат: [536, 82, 44, 32, 26, 12]
```

Сортировка происходит от меньшего числа к большему. Для обратной сортировки необходимо указать параметр revers=True. В случае если в списке будут находиться не только числа, но и другие объекты, вы получите ошибку. Поэтому, применяя функции min, max, sum, sorted, нужно быть уверенным, что в списках содержатся только числа.

8.2. Методы списков

С методами мы уже знакомились ранее, когда изучали строки. Каждый тип объекта имеет свои методы, применяя которые можно изменять объект.

Объекты в Python бывают **изменяемые** и **неизменяемые**.

> Напомню, что **строки** являются **неизменяемым типом**, поэтому, когда мы применяли к ним методы, они отображали их соответствующим образом, но не изменяли их первоначальный вид, сохраненный в переменных.

Для этого нам требовалось перезаписывать переменные с применением метода:

Листинг 8.10

```
text = "привет, мир!" # Оригинальная строка
print(text.upper()) # ПРИВЕТ, МИР! — отобразили с применением метода
print(text) # привет, мир! — вывели содержимое переменной
text = text.upper() # Перезаписали переменную с применением метода
print(text) # ПРИВЕТ, МИР! — теперь переменная хранит новый вариант
```

> Списки являются **изменяемым типом**, поэтому, применяя к ним методы, мы меняем их содержимое. То есть список нельзя перезаписывать, применяя к нему метод, так как это приведет к потере предыдущих данных.

Давайте разберемся на примере первого **метода append**. С его помощью в конец списка добавляется новый элемент.

Листинг 8.11

```
fruits = ['яблоко', 'груша', 'банан']
fruits.append('апельсин')
print(fruits) # Результат: ['яблоко', 'груша', 'банан', 'апельсин']
```

Первоначальный список фруктов содержит в себе три элемента. Когда мы применяем метод **append**, он пополняется новым объектом, указанным в круглых скобках. Далее, выводя на экран список, мы можем увидеть результат этих действий.

Если мы попробуем перезаписать список, как это происходило со строками, то содержимое списка будет потеряно:

Листинг 8.12

```
fruits = ['яблоко', 'груша', 'банан']
fruits = fruits.append('апельсин')
print(fruits) # Результат: None
```

То есть после применения методов к спискам они сразу изменяются. Перезаписывая список с применением метода, мы затираем данные.

Следующий **метод insert**, в отличие от предыдущего, вставляет элемент не в конец списка, а в любое указанное место. Для этого необходимо в скобках указать индекс, на который следует поместить объект, и сам объект.

Листинг 8.13

```
fruits = ['яблоко', 'груша', 'банан']
fruits.insert(2, 'апельсин')
print(fruits) # Результат: ['яблоко', 'груша', 'апельсин', 'банан']
```

Метод **extend** — расширяет список, добавляя все элементы из итерируемого объекта.

Листинг 8.14

```
fruits = ['яблоко', 'груша', 'банан'] # Фрукты
vegetables = ['томат', 'огурец', 'кабачок'] # Овощи
fruits.extend(vegetables) # Расширяем список fruits списком vegetables
print(fruits) # Результат: ['яблоко', 'груша', 'банан', 'томат',
'огурец', 'кабачок']
```

Метод **remove** — удаляет первое вхождение элемента.

Листинг 8.15

```
fruits = ['яблоко', 'груша', 'банан', 'яблоко']
fruits.remove('яблоко') # Удалили первое встреченное яблоко
(индекс 0)
print(fruits) # Результат: ['груша', 'банан', 'яблоко']
fruits.remove('яблоко') # Еще раз удаляем яблоко
print(fruits) # Результат: ['груша', 'банан']
```

Обратите внимание, если мы попытаемся удалить элемент, которого нет в списке, то получим ошибку. Чтобы избежать этого, сначала можно сделать проверку, есть ли данный объект в списке.

Листинг 8.16

```
fruits = ['яблоко', 'груша', 'банан', 'яблоко']
if 'ананас' in fruits: # Если ананас в списке, то:
    fruits.remove('ананас') # Удаляем ананас
    print('Ананас удален из списка')
else: # Иначе не удаляем
    print('Ананаса нет в списке')
```

Метод **pop** — удаляет указанный элемент из списка по индексу. Если элемент не указан, удаляется последний в списке. В отличие от метода remove необходимо указывать позицию элемента, а не его значение.

Листинг 8.17

```
fruits = ['яблоко', 'груша', 'банан', 'яблоко']
fruits.pop() # Удаляем последний элемент
print(fruits) # Результат: ['яблоко', 'груша', 'банан']
fruits.pop(1) # Удаляем элемент с индексом 1
print(fruits) # Результат: ['яблоко', 'банан']
```

Метод **clear** — удаляет все элементы из списка.

Листинг 8.18

```
fruits = ['яблоко', 'груша', 'банан']
fruits.clear()
print(fruits) # Результат: [] - Пустой список
```

Метод **index** — используется для поиска индекса первого вхождения указанного элемента в списке. Если элемент найден, метод возвращает его индекс. Если элемент не найден, вызывается ошибка.

То есть, чтобы избежать нежелательных ошибок, перед поиском индекса также следует сделать проверку, существует ли объект в списке.

Листинг 8.19

```
fruits = ['яблоко', 'груша', 'банан', 'яблоко']
print(fruits.index('яблоко')) # Результат: 0 - Элемент на
нулевой позиции
```

Кроме того, искать элемент можно в указанном срезе. Дополнительно сделаем проверку, что искомый элемент действительно существует.

Листинг 8.20

```
fruits = ['яблоко', 'груша', 'банан', 'яблоко']
if 'яблоко' in fruits[2:]: # Если яблоко в срезе от 2 до конца списка
    print(fruits.index('яблоко',2)) # Поиск яблока со второй позиции
else:
    print('Ничего не найдено')
```

Метод **count** — подсчитывает количество определенного элемента в списке.

Листинг 8.21

```
fruits = ['яблоко', 'груша', 'банан', 'яблоко']
print(fruits.count('яблоко')) # Результат: 2 — найдено яблок в
списке
```

Метод **sort** — аналогичен функции sorted, сортирует список по возрастанию.

Листинг 8.22

```
numbers = [3, 1, 4, 1, 5, 9]
numbers.sort()
print(numbers) # Результат: [1, 1, 3, 4, 5, 9]
```

Метод имеет параметр *reverse*, благодаря которому список можно сортировать в обратном порядке, от большего к меньшему.

Листинг 8.23

```
numbers = [3, 1, 4, 1, 5, 9]
numbers.sort(reverse=True)
print(numbers) # Результат: [9, 5, 4, 3, 1, 1]
```

Метод **reverse** — разворачивает список.

Листинг 8.24

```
fruits = ['яблоко', 'груша', 'банан']
fruits.reverse()
print(fruits) # Результат: ['банан', 'груша', 'яблоко']
```

Метод **copy** — создает копию списка.

Листинг 8.25

```
fruits = ['яблоко', 'груша', 'банан']
fruits_2 = fruits.copy()
print(fruits_2) # Результат: ['яблоко', 'груша', 'банан']
```

8.2.1. Практические задания

1. У нас есть список с несколькими числами в нем. Необходимо найти максимальное и минимальное значения.

   Список: 3, 7, 2, 9, 1.

2. Существует список из нескольких элементов: 3, 7, 2, 9, 1, 3, 7, 8, 9, 11, 15, 7. Применив определенный метод, посчитайте, сколько раз в нем встречается число 7.

3. Отсортируйте список чисел в обратном порядке: 54, 13, 24, 58, 61, 7, 1, 73, 8.

4. Объедините два списка в один.

   Список 1: стол, стул, диван.

   Список 2: ковер, шторы, торшер.

5. На сайте имеется несколько уровней доступа для разных пользователей: admin, moderator, user, guest. Комментарии могут оставлять все, кроме пользователей со статусом guest. Сделайте проверку, в которой определяется, можно ли пользователю оставлять комментарии.

6. Есть список, включающий ряд элементов: "переменная", "строка", "число", "функция", "аргумент", "оператор". Необходимо взять срез индексов со второго до пятого и вывести его на экран.

7. Дан список чисел: 18, 39, 24, 16, 8, 2. Вставьте число 7 на третью позицию.

8.3. Множества

Множество (set) — это неупорядоченный набор уникальных элементов, то есть такой набор, в котором отсутствуют дубли, а расположение элементов случайно.

Для создания множества необходимо придумать название переменной и в фигурных скобках, через запятую, указать его содержимое.

Листинг 8.33

```
nabor = {13, 17, 54, 21, 13, 17}
print(nabor) # Результат: {17, 13, 21, 54}
```

В примере мы можем увидеть, что множество не содержит дублей, которые были прописаны в первоначальном виде. То есть, в отличие от списков, этот объект самостоятельно занимается отслеживанием и удалением повторяющихся элементов.

> Вместе с тем множество можно создавать при помощи одноименной функции **set**.

Листинг 8.34

```
nabor = set("Пример_множества")
print(nabor) # {'П', 'н', 'о', 'е', 'ж', 'с', 'в', 'и', 'а',
'м', 'р', 'т', '_'}
```

То есть на вход подается определенное значение и с помощью функции set происходит разбивка строки на символы. Далее из этих символов организуется множество, при этом дубли исключаются.

Вместе с тем мы можем сделать множество из списка. В таком случае дубли также будут удалены, а расположение элементов будет случайно:

Листинг 8.35

```
spisok = [15, 4, 12, 9, 4, 9]
nabor = set(spisok)
print(nabor) # {9, 4, 12, 15}
```

Чтобы преобразовать множество снова в список, необходимо провести обратную операцию:

Листинг 8.36

```
spisok = [1, 1, 2, 2, 3, 3, 4, 4] # Первоначальный список
nabor = set(spisok) # Множество {1, 2, 3, 4}
spisok = list(nabor) # Новый список из множества
print(spisok) # Результат: [1, 2, 3, 4]
```

Для создания пустого множества необходимо прописать конструкцию set(). Обратите внимание, что множество нельзя создать при помощи простых фигурных скобок. В таком случае будет создан словарь, с которым мы познакомимся позже.

Листинг 8.37

```
primer = set() # Пустое множество
slovar = {} # Пустой словарь
```

8.4. Методы и операции над множествами

Множество является **неизменяемым типом данных**, поэтому он может содержать неизменяемые типы данных, такие как строки, числа и кортежи. Как в случае со списками и строками, над множествами можно проводить определенные операции и применять методы.

Метод **add** — используется для добавления элемента во множество:

Листинг 8.38

```
nabor = {"один", "два", "три", "четыре"} # Первоначальное множество
nabor.add("пять") # Добавляем элемент
print(nabor) # Результат: {'один', 'два', 'пять', 'четыре', 'три'}
```

Обратите внимание, что если мы добавим во множество объект, который уже существует в нем, то множество не изменится, так как оно может включать в себя только уникальные элементы.

Листинг 8.39

```
nabor = {"один", "два", "три", "четыре"} # Первоначальное
множество
nabor.add("два") # Добавляем элемент
print(nabor) # Результат: {'один', 'два', 'четыре', 'три'}
```

Метод **update** — применяется для добавления сразу нескольких значений:

Листинг 8.40

```
nabor = {"один", "два", "три"} # Первоначальное множество
nabor.update(["четыре", "пять", "шесть"]) # Добавляем список
```

```
элементов
print(nabor) # Результат: {'два', 'три', 'один', 'шесть',
'четыре', 'пять'}
```

Метод **discard** — позволяет удалить выбранный элемент:

Листинг 8.41

```
nabor = {"один", "два", "три"} # Первоначальное множество
nabor.discard("два") # Удаляем элемент
print(nabor) # Результат: {'один', 'три'}
```

Метод **remove** — также удаляет указанный элемент:

Листинг 8.42

```
nabor = {43, 18, 115, 11, 98} # Первоначальное множество
nabor.remove(18) # Удаляем элемент
print(nabor) # Результат: {98, 115, 11, 43}
```

Обратите внимание, что, если во множестве отсутствует элемент, который нужно удалить методом **remove**, это вызовет ошибку. Однако при использовании метода **discard** ничего не произойдет. В этом заключается главное различие между двумя этими методами.

Метод **pop** — удаляет из множества случайный элемент. Если во множестве отсутствует элемент, это вызовет ошибку. Поэтому сначала следует сделать проверку, существует ли объект во множестве.

Листинг 8.43

```
nabor = {43, 18, 115, 11, 98} # Первоначальное множество
nabor.pop() # Удаляем элемент
print(nabor) # Результат: {18, 115, 11, 43}
```

Функция **len**, как и в случае с другими объектами, позволяет пересчитать количество элементов.

Листинг 8.44

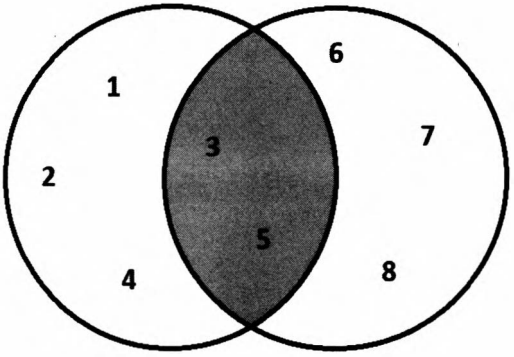
```
nabor = {43, 18, 115, 11, 98}
print(len(nabor)) # Результат: 5 — элементов во множестве
```

Оператор **in** — позволяет определить, находится ли элемент во множестве.

Листинг 8.45

```
nabor = {43, 18, 115, 11, 98}
print(18 in nabor, 21 in nabor) # Результат: True False
# 18 есть во множестве, 21 — нет

if 43 in nabor: # Пример в условии
    print("Число найдено")
```

Нахождение пересечения — позволяет определить, какие значения находятся в двух или нескольких наборах множеств. Для этого используется амперсанд **&**.



Изображение 8.1.

Листинг 8.46

```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
print(nabor_1 & nabor_2) # Результат: {3, 5}
```

Операция **intersection** — также позволяет найти пересечение между двумя множествами.

Листинг 8.47

```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
print(nabor_1.intersection(nabor_2)) # Результат: {3, 5}
```

Обратите внимание, что первый набор после применения метода не был изменен. Внутри множества остались те же значения. Чтобы nabor_1 перезаписал свое содержимое, используется метод **intersection_update**.

Листинг 8.48

```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
nabor_1.intersection_update(nabor_2)
print(nabor_1) # Результат: {3, 5}
```

Операция **объединения множеств** — происходит с помощью вертикальной черты (|). В результате в одно множество помещаются оба набора, за исключением дублей.

Листинг 8.49

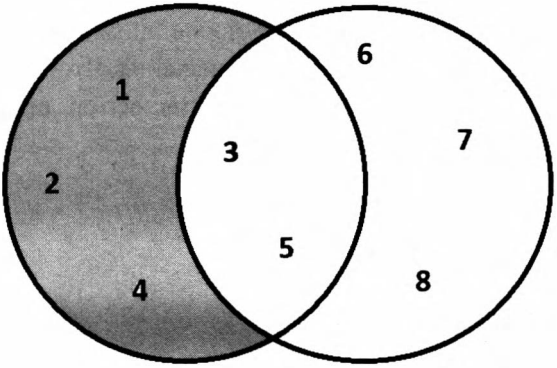
```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
print(nabor_1 | nabor_2) # Результат: {1, 2, 3, 4, 5, 6, 7, 8}
```

Метод **union** — делает аналогичную операцию по объединению.

Листинг 8.50

```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
print(nabor_1.union(nabor_2)) # Результат: {1, 2, 3, 4, 5, 6, 7, 8}

# Для сохранения результата необходимо перезаписать переменную:
nabor_1 = nabor_1.union(nabor_2)
```

Удаление пересекающихся элементов происходит с помощью знака минус (-). Из одного набора мы можем вычесть только те элементы, которые пересекаются со вторым набором.

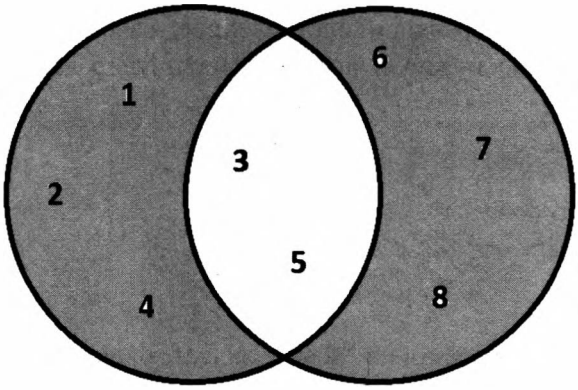


Изображение 8.2.

Листинг 8.51

```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
print(nabor_1 - nabor_2) # Результат: {1, 2, 4}
# 3, 5 - удалены
```

Сложение симметричных разностей — данная операция прописывается с помощью галочки вверх (^). Она объединяет элементы множеств, которые не пересекаются между собой.



Изображение 8.3.

Листинг 8.52

```
nabor_1 = {1,2,3,4,5}
nabor_2 = {3,5,6,7,8}
print(nabor_1 ^ nabor_2) # Результат: {1, 2, 4, 6, 7, 8}
```

8.4.1. Практические задания

1. Дан список: ["ананас", "манго", "банан", "манго", "ананас"], необходимо с помощью множества избавиться от дублей и преобразовать его обратно в список.
2. Дано множество: {"ананас", "манго", "банан"}, необходимо добавить в него элементы "апельсин" и "абрикос", а также удалить "ананас".

8.5. Словари

Словарь (dictionary) в Python — это набор элементов, где каждый элемент представляет собой пару "ключ: значение". Как и списки, словари являются изменяемым типом данных. То есть мы можем на ходу изменять их содержимое.

Для создания словаря используются фигурные скобки, в которые помещается уникальный ключ, хранящий определенное значение. По этому ключу мы можем обращаться к нужному значению, манипулировать им и изменять.

Листинг 8.55

```
dict_1 = {} # Пустой словарь
dict_2 = {"Ivan": "333-333-333", "Andrey": "444-444-444"}
print(dict_2)
```

В примере выше каждое имя в словаре является ключом, которое хранит номер телефона. Обращаясь к словарю и к нужному ключу, мы можем увидеть его содержимое.

Листинг 8.56

```
dict_2 = {"Ivan": "333-333-333", "Andrey": "444-444-444"}
print(dict_2["Andrey"]) # Результат: 444-444-444
```

Это напоминает списки, где по индексу мы находили нужный нам элемент. Однако в словарях вместо индексов используются ключи, которые всегда привязаны к своему значению. Поэтому, независимо от расположения пары в словаре, мы всегда можем обратиться именно к тому элементу, который нам необходим.

Для изменения значения в ключе используется стандартный способ присвоения. Мы указываем название словаря и ключ в квадратных скобках, а затем с помощью присвоения указываем новое значение.

Листинг 8.57

```
dict_2 = {"Ivan": "333-333-333", "Andrey": "444-444-444"} #
Первоначальный словарь
dict_2["Andrey"] = "555-555-555" # Присвоили новый номер
print(dict_2["Andrey"]) # Результат: 555-555-555
```

Изменять содержимое ключа также можно с помощью математических операций:

Листинг 8.58

```
dict_3 = {"Хлеб": 50, "Молоко": 70}
dict_3["Молоко"] += 15 # Добавили 15
print(dict_3["Молоко"]) # Результат: 85
```

Чтобы добавить в словарь новую пару из ключа и значения, прописывается операция, аналогичная присвоению.

Листинг 8.59

```
dict_3 = {"Хлеб": 50, "Молоко": 70}
dict_3["Масло"] = 60 # Добавили новую пару
print(dict_3) # Результат: {'Хлеб': 50, 'Молоко': 70, 'Масло': 60}
```

Для удаления определенной пары используется функция **del**.

Листинг 8.60

```
dict_3 = {'Хлеб': 50, 'Молоко': 70, 'Масло': 60}
del dict_3["Масло"] # Удаляем ключ "Масло" и его значение
print(dict_3) # Результат: {'Хлеб': 50, 'Молоко': 70}
```

Если попытаться удалить ключ, которого нет в словаре, это приведет к ошибке. Поэтому предварительно следует удостовериться, что данный ключ действительно присутствует, например, используя проверку **if key in dict** (если ключ есть в словаре).

Создавая большие словари, их можно записывать блоками для более удобного чтения. Не забывайте, что ключ и его значение разделяются двоеточием (:), а между парами ставится запятая.

Листинг 8.61

```
dict_3 = {'Хлеб': 50,
          'Молоко': 70,
          'Масло': 60
          }
print(dict_3)
```

Вместе с тем словарь можно создавать с помощью соответствующей функции ***dict***. В круглых скобках указываются пары через запятую. А ключ и значение связываются символом присвоения.

Листинг 8.62

```
dict_4 = dict(Хлеб=50, Молоко=70, Масло=60) # Создание словаря
print(dict_4) # Результат: {'Хлеб': 50, 'Молоко': 70, 'Масло': 60}
```

Обратите внимание, что ключи записываются без кавычек. Функция ***dict*** автоматически преобразовывает их к строке. Стоит иметь в виду, что подобным образом словари создаются, только если ключи являются строковыми типами.

Также функция ***dict*** может быть полезна, когда необходимо преобразовать вложенные списки в словарь.

Листинг 8.63

```
# Вложенные списки (списки внутри списка)
spisok = [['Хлеб',15], ['Молоко',16], ['Масло',17]]
dict_5 = dict(spisok) # Преобразуем список в словарь
print(dict_5) # Результат: {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}

# Второй пример
spisok = [[1,15],[2,16],[3,17]]
dict_5 = dict(spisok) # Преобразуем список в словарь
print(dict_5) # Результат: {1: 15, 2: 16, 3: 17}
```

То есть, имея списки, содержащие по два элемента, мы можем сохранить их в виде словаря. Где первое значение будет **неизменным** ключом, а второе — **изменяемым** значением.

Как и с другими типами данных, мы можем подсчитать количество элементов в словаре с помощью функции ***len***. В данном случае в ответе будет выводиться количество пар.

Листинг 8.64

```
dict_5 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(len(dict_5)) # Результат: 3 (пары)
```

8.6. Методы и операции над словарями

Со словарями, как и с другими типами объектов, можно проводить различные операции и применять специальные методы.

Метод **fromkeys** — позволяет создать словарь альтернативным способом. Его особенность заключается в том, что для создания словаря достаточно указать только ключи. По умолчанию все они будут хранить значение None (не определено). Позже эти значения можно изменить по необходимости.

Листинг 8.65

```
dict_6 = dict.fromkeys(["x", "y", "z"]) # Ключи без значений
print(dict_6) # Результат: {'x': None, 'y': None, 'z': None}
```

Вместе с тем можно указать определенное значение, которое продублируется в каждый ключ, и впоследствии заменять эти значения на новые.

Листинг 8.66

```
dict_6 = dict.fromkeys(["x", "y", "z"], 0)
print(dict_6) # Результат: {'x': 0, 'y': 0, 'z': 0}
```

Метод **clear** — очищает словарь, удаляя все элементы.

Листинг 8.67

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
dict_7 = dict_7.clear() # Очищаем словарь и перезаписываем
print(dict_7) # Результат: None
```

Метод **copy** — создает копию словаря.

Листинг 8.68

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
dict_8 = dict_7.copy() # В словарь dict_8 записали копию словаря dict_7
print(dict_8) # Результат: {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
```

Метод **get** — позволяет обратиться к определенному ключу и взять его значение. Если ключ отсутствует, возвращается None или указанное значение.

Листинг 8.69

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(dict_7.get('Хлеб', 'Ничего не найдено')) # Результат: 15
```

В примере выше мы обращаемся к ключу "Хлеб", который содержит значение 15. Если удалить этот ключ из словаря, результатом будет выведена фраза "Ничего не найдено". Таким образом, метод **get** может заменить проверку **if-else**, в которой мы можем проверять, есть ли ключ "Хлеб" в словаре.

Метод **pop** — удаляет элемент по ключу и возвращает его значение. Если ключа нет, возвращает значение по умолчанию или вызывает исключение.

Листинг 8.70

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(dict_7.pop('Молоко')) # Результат: 16
print(dict_7) # Результат: {'Хлеб': 15, 'Масло': 17} Молоко удалено
```

Метод **popitem** — удаляет и возвращает произвольную пару ключ – значение.

Листинг 8.71

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(dict_7.popitem()) # Результат: ('Масло', 17)
print(dict_7) # Результат: {'Хлеб': 15, 'Молоко': 16} Масло удалено
```

Метод **update** — обновляет словарь, добавляя элементы из другого словаря или из любого другого объекта, который поддерживает метод items.

Листинг 8.72

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
dict_7.update({'Творог': 21})
print(dict_7) # Результат: {'Хлеб': 15, 'Молоко': 16, 'Масло': 17, 'Творог': 21}
```

Метод **keys** — выводит список всех ключей без их значений.

Листинг 8.73

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(dict_7.keys()) # Результат: dict_keys(['Хлеб', 'Молоко', 'Масло'])
```

Метод **values** — выводит список всех значений без ключей.

Листинг 8.74

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(dict_7.values()) # Результат: dict_values([15, 16, 17])
```

Метод **items** — возвращает весь набор пар из ключа и значения в виде кортежей.

Листинг 8.75

```
dict_7 = {'Хлеб': 15, 'Молоко': 16, 'Масло': 17}
print(dict_7.items()) # Результат: dict_items([('Хлеб', 15),
('Молоко', 16), ('Масло', 17)])
```

8.7. Пример создания словаря из данных в текстовом виде

Словари позволяют хранить большие объемы данных, где каждый элемент может быть вызван по своему ключу. Для примера давайте разберем ситуацию, когда у нас есть определенный массив данных, который нужно систематизировать.

Для начала полученные данные в виде обычного текста можно сохранить в переменную для дальнейшей работы с ней:

Листинг 8.76

```
# Переменная хранит данные
text = "Иван Васильевич Москва Ленина 19"
```

Затем методом .split можно разбить эту строку на отдельные элементы, благодаря чему мы получим список подстрок:

Листинг 8.77

```
text = "Иван Васильевич Москва Ленина 19"
spisok = text.split()
print(spisok) # ['Иван', 'Васильевич', 'Москва', 'Ленина', '19']
```

Далее нам остается создать словарь и разложить каждый отдельный элемент списка по своим ключам. Мы помним, что у элементов в списке есть индексы, поэтому, привязывая значения к ключам, нужно указать их порядковый номер:

Листинг 8.78

```
data = {} # Создали пустой словарь с данными
# Заполняем словарь
data['name'] = spisok[0] # В ключ name сохранится нулевой элемент
списка
data['lastname'] = spisok[1] # В ключ фамилии сохранится элемент с
индексом 1
data['city'] = spisok[2] # Сохранили третий элемент (индекс 2)
data['street'] = spisok[3] # Сохранили улицу
data['namber'] = spisok[4] # Сохранили номер дома
print(data)
```

Таким образом, мы получили систематизированные данные в словаре и можем вызвать любой элемент по его ключу для дальнейшей работы с ним. В следующих главах, когда мы познакомимся с циклами, данный код можно будет сократить.

8.7.1. Практические задания

1. Создайте список с помощью метода **fromkeys**, который будет содержать три ключа с любым значением. Затем удалите из словаря последнюю пару ключ – значение, а у первых двух пар измените значения на новые. Выведите результат на консоль.
2. Создайте словарь из нескольких пар. Затем проверьте один из ключей на наличие его в словаре. Если ключ есть в словаре, выведите на экран его значение. Если его нет — добавьте ключ в словарь, присвоив ему значение. Выведите результат.

8.8. Кортежи

Кортежи (tuple) являются **неизменяемыми** объектами в Python. Они используются для хранения разнотипных объектов и во многом напоминают списки. Оба этих объекта поддерживают большое количество одинаковых операций.

Главное различие между ними — это **неизменяемость кортежей**. Из-за этого с ними невозможны некоторые операции, в отличие от списков.

Для создания кортежей используются круглые скобки, внутри которых, через запятую, помещаются элементы. Кроме того, кортежи могут не иметь скобок и просто включать в себя ряд элементов через запятую. Такой вариант встречается реже, но стоит иметь его в виду. Общепринято использовать скобки для улучшения читаемости кода и избегания возможной путаницы.

Листинг 8.81

```
x, y, z = 1, 2, 3
kortezh_1 = (x, y, z) # Кортеж, заключенный в скобки
kortezh_2 = x, y, z # Кортеж без скобок
kortezh_3 = x, # Кортеж из одного элемента
kortezh_4 = () # Пустой кортеж
print(kortezh_1)
print(kortezh_2)
print(kortezh_3)
```

Обратите внимание, что в третьем примере (kortezh_3) после элемента стоит запятая. Она указывает на то, что это последовательность, а не просто переменная.

Вместе с тем для создания кортежа существует одноименная функция ***tuple***, которая преобразовывает в кортеж указанную в ней последовательность.

Листинг 8.82

```
kortezh_5 = tuple([1, 2, 3]) # Передали список в функцию для создания кортежа
print(kortezh_5)
```

Над кортежами можно проводить операции, подобно спискам. Так как они содержат последовательность элементов, мы можем подсчитать их количество с помощью функции ***len***.

Листинг 8.83

```
kortezh_6 = (1, 2, 3, 4, 5)
print(len(kortezh_6)) # Результат: 5 — элементов
```

С помощью оператора **in** можно проверить наличие элемента в кортеже:

Листинг 8.84

```
kortezh_6 = (1, 2, 3, 4, 5)
print(3 in kortezh_6) # Результат: True — (тройка есть в кортеже)
```

Кортежи можно сцеплять между собой:

Листинг 8.85

```
x = (31, 42, 53)
y = (64, 75, 86)
z = x + y
print(z) # Результат: (31, 42, 53, 64, 75, 86)
```

Кортежи можно дублировать:

Листинг 8.86

```
x = (31, 42, 53)
print(x * 2) # Результат: (31, 42, 53, 31, 42, 53)
```

Также, если кортежи состоят из чисел, к ним можно применять функции **min, max** и **sum**:

Листинг 8.87

```
x = (31, 42, 53)
print(min(x)) # Результат: 31
print(max(x)) # Результат: 53
print(sum(x)) # Результат: 126
```

Так как кортежи являются последовательностью элементов, мы можем обращаться к одному из них по индексу, а также брать определенные срезы:

Листинг 8.88

```
x = (31, 42, 53, 64, 75, 86)
print(x[2]) # Выводим третий элемент кортежа (индекс 2)
print(x[1:4]) # Выводим срез с первого по четвертый индекс (42,
53, 64)
```

Как уже говорилось выше, кортежи являются неизменяемым типом, поэтому мы не можем изменять их содержимое. При попытке сделать это будет получена ошибка. Из-за этой особенности кортежи имеют очень мало методов по сравнению со списками. В частности, это методы **count** и **index**, которые никак не влияют на содержимое кортежа при их применении.

Метод **count** переводится как "количество" и, соответственно, позволяет узнать количество указанного элемента в кортеже.

Листинг 8.89

```
x = (31, 42, 53, 64, 75, 31)
print(x.count(31)) # Результат: 2 (число 31 встречается два раза)
```

Метод **index** позволяет вызвать элемент под определенным индексом, если таковой имеется. В противном случае будет получена ошибка.

Листинг 8.90

```
x = (31, 42, 53, 64, 75, 86)
print(x.index(64)) # Результат: 3 — узнали индекс элемента
```

Однако, несмотря на то что кортеж является неизменяемым типом, он может содержать в себе изменяемые типы объектов.

Листинг 8.91

```
x = ([31, 42], [53, 64], [75, 86])
x[2].append(97)
print(x) # Результат: ([31, 42], [53, 64], [75, 86, 97])
```

В примере выше изначальный кортеж содержит три списка, каждый из двух чисел, в один из них методом **append** мы добавили новый элемент 97, указав индекс списка в кортеже.

Казалось бы, зачем нужны кортежи, если можно использовать списки, к которым можно применить больше операций? В некоторых ситуациях необходимо, чтобы определенные данные оставались неизменными, так как в списках они могут быть случайно отредактированы в ходе каких-либо операций или вычислений. Кортежи гарантируют неизменность данных.

Вторая причина — оптимизация. Кортеж занимает меньше места в памяти, чем список, поэтому скорость его обработки будет быстрее. В этом можно убедиться, создав однотипные кортеж и список и с помощью служебного метода **sizeof** посмотрев их значения в байтах.

Листинг 8.92

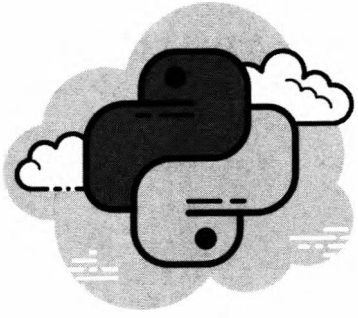
```
x = (31, 42, 53, 64, 75, 86) # Кортеж
y = [31, 42, 53, 64, 75, 86] # Список
print(x.__sizeof__()) # Результат: 72 байта занимает кортеж
print(y.__sizeof__()) # Результат: 88 байт занимает список
```

В данном примере разница может показаться незначительной, но в больших программах, с большим количеством данных, это может сыграть ключевую роль.

Вместе с тем кортеж по необходимости можно преобразовать в список функцией **list**.

Листинг 8.93

```
x = (31, 42, 53, 64, 75, 86) # Кортеж
y = list(x) # Преобразовали в список
print(y) # Результат: [31, 42, 53, 64, 75, 86]
```



Глава 9.

Циклы

9.1. Цикл *while*

Представьте, что вам нужно выполнить одно и то же действие несколько раз подряд. Как это сделать в коде без ручного копирования и вставки одних и тех же строк? Для таких случаев в Python существуют циклы.

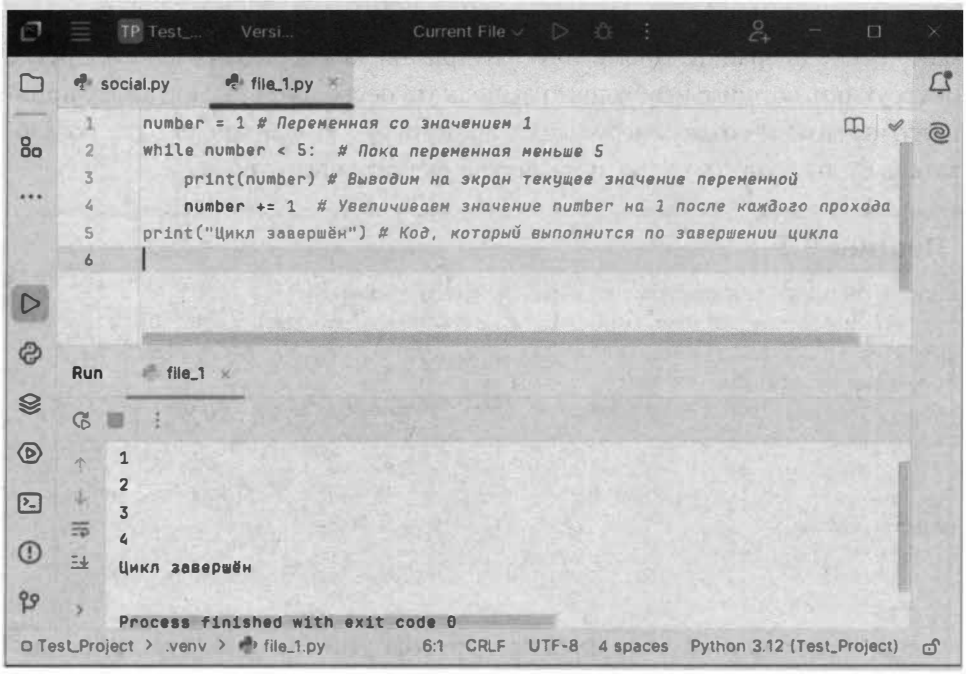
Цикл **while** позволяет программе выполнять блок кода до тех пор, пока выполняется определенное условие. Слово *while* переводится как "пока", и это сразу помогает понять принцип его работы. Программа продолжает выполнять указанный блок кода, **пока** условие истинно (то есть равно True). Как только условие становится ложным (False), программа выходит из цикла.

Структура цикла состоит из условия и блока кода внутри него:

Листинг 9.1

```
while условие: # Условие цикла
    # Блок кода, выполняемый при истинном условии
# Здесь следующий код, который выполняется по завершении цикла
```

То есть в первой строке проверяется условие, аналогично оператору **if**. Затем, если условие верно, выполняется блок внутри условия. После этого интерпретатор снова возвращается к первой строке и опять проверяет условие на истинность. Если оно по-прежнему верно, блок кода внутри цикла выполняется еще раз. И так до тех пор, пока условие не станет ложным. В этом случае, после проверки условия, Пайтон переходит к следующей строке кода, которая следует после блока цикла.



Изображение 9.1.

На изображении во второй строке начинается цикл, в котором стоит условие *while number < 5* (пока переменная меньше пяти). На текущий момент переменная хранит значение 1, и это отвечает условию цикла. Поэтому далее выполняется команда *print*, которая выводит текущее число, а затем к этому числу добавляет единицу (текущее значение переменной становится 2).

Затем интерпретатор снова возвращается ко второй строке с условием и проверяет, является ли двойка меньше пяти. Условие верно, поэтому блок кода внутри повторяется: на экран выводится текущее значение — 2, а затем переменная увеличивается на единицу.

Так происходит несколько итераций, **пока** переменная меньше пяти. Как только переменная достигает значения 5, что не отвечает условию цикла, блок кода пропускается, и Пайтон переходит к следующим инструкциям после этого блока.

Обратите внимание, что внутри цикла должно изменяться условие этого цикла, иначе он будет бесконечным и программа никогда не продвинется дальше.

Помимо увеличения переменной, здесь можно проводить другие математические операции. Кроме того, внутри цикла могут быть любые другие инструкции, которые необходимо выполнять несколько раз. По такому принципу мы можем создать небольшую мини-игру "Угадай число", где пользователь будет вводить число, пока оно не окажется верным.

Листинг 9.2

```
import random # Импортируем модуль рандомизации
secret_number = random.randint(1, 100) # Генерируем число от 1 до 100
guess = int(input("Угадайте число от 1 до 100: ")) # Переменная,
хранящая введенное число
while guess != secret_number: # Пока введенное число не равно
секретному числу
    print("Это не то число, попробуйте еще раз")
    guess = int(input("Угадайте число от 1 до 100: ")) # Предлагаем
ввести число
print(f"Верно! Это число {secret_number}")
```

В примере при запуске цикла проверяется условие: пока введенное число (guess) не равно секретному числу, будет выполняться код внутри цикла. Внутри цикла мы сообщаем пользователю, что введенное число неверно, и предлагаем ввести его еще раз. Как только число будет угадано, обе переменные станут равны друг другу, условие цикла перестанет выполняться и будет выведено сообщение, следующее после цикла.

Цикл **while** также может помочь, когда необходимо пройтись по большому массиву данных и сделать с ним определенные действия. Например, у нас имеется список номеров телефонов, где некоторые из них повторяются несколько раз. Мы можем почистить список от дубликатов с помощью цикла.

Листинг 9.3

```
phone_list = [111, 222, 333, 111, 444, 111] # Список номеров
unique_list = [] # Новый список для уникальных номеров
number = None # Переменная для временного хранения номеров
while phone_list: # Пока список True, то есть не пустой
    number = phone_list.pop(0)  # Извлекаем первый элемент из phone_list
    if number not in unique_list: # Если телефона нет в списке уникальных
        unique_list.append(number)  # Добавляем номер в список unique_list
print(unique_list) # Выводим список без дублей: [111, 222, 333, 444]
```

1. В примере мы создали новый пустой список для хранения уникальных номеров, а также переменную, которая будет временно сохранять каждый из номеров.
2. При запуске цикла происходит проверка условия, пока список номеров не пустой — выполняется цикл.
3. Внутри цикла метод pop берет элемент с индексом 0 (111) и удаляет его из списка phone_list. Удаленный номер сохраняется в переменной **number**.
4. Далее происходит проверка, если телефон, сохраненный в переменной, не в списке unique_list, то методом **append** он туда добавляется.
5. На этом этапе цикл выполнил одну итерацию и возвращается к началу, где снова проверяет, является ли список пустым. На текущий момент в нем на один номер меньше.
6. Теперь телефоном с индексом 0 является 222, и уже он удаляется из списка phone_list и сохраняется в переменной *number*.
7. Затем снова происходит проверка на наличие этого номера в списке unique_list и его добавление туда, если телефона нет в списке.
8. Таким образом, цикл проделывает одни и те же манипуляции над каждым номером, удаляя их из первоначального списка и добавляя в новый, если их там нет.
9. На последней итерации список phone_list остается пустым, поэтому условие цикла перестает выполняться, и интерпретатор переходит к строчке кода после цикла — print(unique_list), которая выводит на экран новый список без дубликатов.

9.1.1. Практические задания

1. Напишите цикл, который будет выводить числа до десяти, через один. То есть числа 1, 3, 5 и т.д.
2. Напишите цикл, который будет выводить числа от 15 до 5 и завершится, когда числовой ряд дойдет до четырех.
3. Есть список, содержащий большое количество строк: ["кот", "пес", "кот", "кит", "кот", "ерш", "кот"], необходимо с помощью цикла while и соответствующих методов удалить из него все строки со значением "кот".

4. Предложите пользователю ввести пароль, и проверяйте правильность ввода до тех пор, пока он не введет правильный.

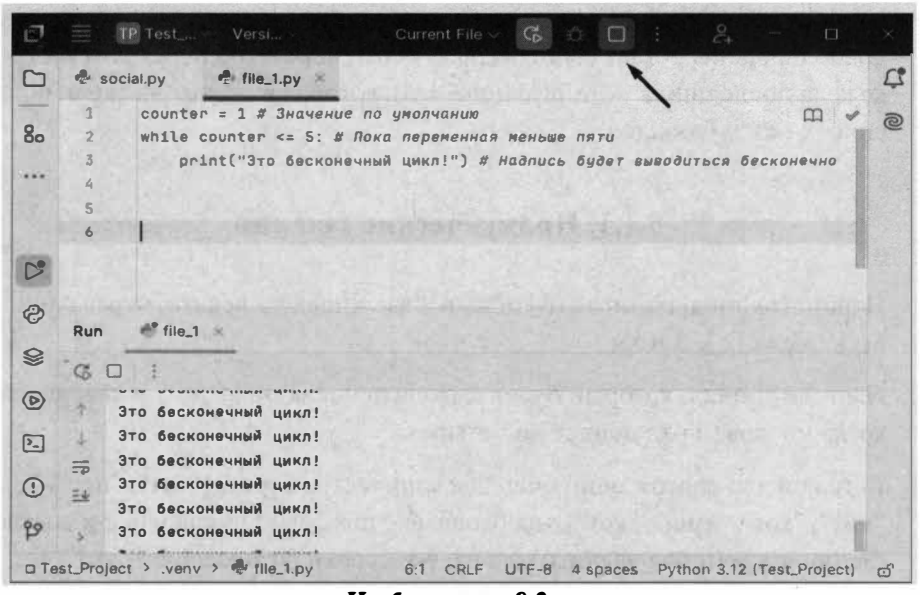
9.2. Инструкции *break*, *continue*, *else*

Если случайно забыть изменить переменную, которая участвует в условии, или некорректно задать само условие, то цикл ***while*** может выполняться бесконечно, то есть никогда не завершится. Это называется бесконечным циклом.

Листинг 9.8

```
counter = 1 # Значение по умолчанию
while counter < 5: # Пока переменная меньше пяти
    print("Это бесконечный цикл!") # Надпись будет выводиться
бесконечно
```

В примере программа выводит надпись бесконечно, так как условие цикла всегда остается верным. Завершить такую программу придется принудительно, например, закрыв ее или нажав соответствующую кнопку в среде разработки. В IDLE для экстренной остановки можно использовать комбинацию Ctrl + C. Поэтому при создании цикла следует тщательно продумать логику и условия выхода из него.



Изображение 9.2.

Однако, бесконечные циклы могут применяться, когда требуется выполнять определенные действия на протяжении всего времени работы программы. Для создания бесконечного цикла обычно используются константные выражения, которые всегда остаются неизменными, например *while 1==1* (пока единица равна единице).

Из предыдущих разделов мы знаем, что все выражения можно привести к булевым значениям (True / False). То есть **1==1** — это верное сравнение (True). Поэтому условие для бесконечного цикла можно записать как while True — в таком виде вы чаще всего будете встречать бесконечный цикл.

Листинг 9.9

```
number = 1
while True:
    print(number)
    number += 1 # Бесконечное увеличение числа и вывод его на экран
print("Код после цикла")
```

При экстренном торможении цикла работа программы прекращается. То есть, если после цикла у вас прописан какой-то код, он не будет выполнен. Поэтому для контроля циклов используются управляющие операторы **break** и **continue**.

Оператор **break** — используется для выхода из цикла до того, как условие станет ложным.

Листинг 9.10

```
number = 1
while True:
    print(number)
    if number == 5:
        break # Выходим из цикла, когда переменная равна 5
    number += 1 # Бесконечное увеличение числа
print("Код, после цикла")
```

Подобная операция может использоваться, например, в программе, принимающей какие-то данные:

Листинг 9.11

```
while True:
    text = input() # Ввод данных от пользователя
    if text == "завершить":
        break # Выходим из цикла, когда введено слово "завершить"
    print(text)
```

В данном примере программа бесконечно принимает любые данные от пользователя, пока он не напишет слово "завершить". В этом случае сработает команда *break*, цикл тут же будет завершен, а весь нижеследующий код этого цикла проигнорирован. Далее интерпретатор перейдет к коду, который следует после цикла.

Оператор **continue** — пропускает текущую итерацию, но продолжает выполнять следующие.

Листинг 9.11

```
number = 1
while number < 5: # Пока число меньше 5
    if number == 3:
        number += 1
        continue # Переходим к следующей итерации, пропуская print(number)
    print(number) # Выводим число на экран
    number += 1 # Увеличиваем число
# Результат: пропущена третья итерация
# 1
# 2
# 4
```

В примере выше цикл выводит на экран содержимое переменной, а затем увеличивает ее на 1. Когда переменная становится равна трем, переменная также увеличивается на единицу, но после этого команда *continue* отправляет интерпретатор к началу цикла, пропуская нижеследующий код. Поэтому на экран не выводится текущее значение. И далее цикл работает в штатном режиме.

Оператор **else** — работает с циклом *while* так же, как и в связке с оператором if. Он запускает свои инструкции, если условие цикла не выполняется.

Листинг 9.12

```
number = 1
while number < 5: # Пока число меньше 5
    print(number) # Выводим число на экран
    number += 1 # Увеличиваем число
else:
    print("Цикл завершен")
```

В примере на экран выводится значение переменной, которая увеличивается при каждой итерации. По достижении значения 5 цикл прекращается, так как условие становится ложным (5 < 5 == False). И далее срабатывает инструкция ***else***, которая выполняет свои действия, если предыдущее условие ложно.

Стоит иметь в виду, что, если цикл был принудительно завершен командой *break*, это также пропускает операцию *else*, и Пайтон переходит к коду после всей конструкции **while-else**.

Листинг 9.13

```
while number < 5: # Пока число меньше 5
    print(number) # Выводим число на экран
    if number == 4:
        break # Завершили цикл
    number += 1 # Увеличиваем число
else:
    print("Цикл завершен")
print("Программа завершена") # Код после цикла
```

В примере при достижении числа 4 срабатывает команда *break*, тут же завершающая весь цикл. Условие в строке *while* больше не проверяется, а значит, также неизвестны противоположные условия для оператора ***else***. Интерпретатор переходит к строке *print* ("Программа завершена").

9.3. Функция *range*

Функция **range** позволяет создавать последовательности чисел. Она часто используется с циклом for, о котором речь пойдет в следующем разделе. Полезность функции заключается в том, что она избавляет от необходимости вручную писать массивы чисел. Range имеет три параметра: start, stop и step.

1. В параметре *stop* указывается конечное значение, до которого будет сделан расчет чисел. По умолчанию расчет начинается с нуля.

Листинг 9.14

```
spisok = list(range(5))
print(spisok) # Результат: [0, 1, 2, 3, 4]
```

2. В примере с помощью функции range мы указали интерпретатору, что нужно ранжировать ряд чисел **до** пяти (не включительно). Затем, для удобства отображения, мы преобразовали этот ряд чисел в список функцией ***list*** и сохранили результат в переменной *spisok*.

   Также мы можем указать стартовый параметр, от которого будет производиться расчет:

Листинг 9.15

```
spisok = list(range(15, 20))
print(spisok) # Результат: [15, 16, 17, 18, 19]
```

3. Третий параметр *step* — устанавливает шаг между ранжируемыми числами, благодаря чему мы можем выводить числа через один, через два и т.д.

Листинг 9.16

```
spisok = list(range(15, 25, 2)) # Берем каждое второе число
print(spisok) # Результат: [15, 17, 19, 21, 23]
```

В примере мы указали, что нужно брать каждое второе число, начиная с 15. То есть все нечетные. Так как последнее число не включается в ранжирование, аналогично срезам, расчет был закончен на 23.

Для обратного отсчета параметр шага нужно установить на -1, а стартовое значение должно быть больше конечного, иначе сформированный список будет пустым.

Листинг 9.17

```
spisok = list(range(20, 15, -1))
print(spisok) # Результат: [20, 19, 18, 17, 16]
```

На практике функция ***range*** используется для прохода по определенному количеству шагов или для создания счетчиков. Для примера рассмотрим скрипт, который перебирает числа от 1 до 10 и умножает каждое на 9. Таким образом, мы можем получить таблицу умножения числа 9.

Листинг 9.18

```
for i in range(1, 11):
    print(f"9 x {i} = {9 * i}")
# Результат:
# 9 x 1 = 9
# 9 x 2 = 18
# 9 x 3 = 27
# 9 x 4 = 36
# 9 x 5 = 45
# 9 x 6 = 54
# 9 x 7 = 63
# 9 x 8 = 72
# 9 x 9 = 81
# 9 x 10 = 90
```

В переменную ***i*** поочередно подставляются числа от 1 до 11, а во второй строке каждое из них умножается на 9, и далее выводится результат.

Затем цикл возвращается к началу, и в переменную подставляется следующее число из числового ряда. Цикл завершается, когда Пайтон переберет все числа в указанном диапазоне.

9.4. Цикл *for*

Цикл **for** позволяет выполнять определенные действия несколько раз подряд, перебирая каждый элемент в последовательности, такой как список, строка или диапазон чисел. Он используется, когда заранее известно, сколько раз необходимо повторить код или когда нужно пройтись по элементам в коллекции данных.

Когда мы пишем цикл **for**, мы указываем интерпретатору, по какой последовательности он должен "пройтись" и что делать с каждым элементом в этой последовательности. В предыдущем разделе, на примере таблицы умножения, мы рассмотрели, как цикл **for** работает вместе с функцией ***range*** и производит одни и те же действия для каждого элемента последовательности.

Листинг 9.19

```
for i in range(5):
    print(i)
```

Функция ***range*** создает последовательность чисел от 0 до 5 (не включительно), а цикл **for** проходится по этой последовательности, подставляя в переменную *i* каждое из чисел по очереди. То есть при первой итерации в переменной сохраняется значение 0, команда *print* выводит его на экран. Затем цикл возвращается в начало, в переменную сохраняется следующее значение — 1, и так продолжается, пока цикл не пройдет всю последовательность.

Таким же образом мы можем последовательно пройтись по каждому элементу строки и вывести каждый символ по отдельности.

Листинг 9.20

```
for i in "яблоко":
    print(i)
```

В примере каждая буква будет выводиться на новой строке. Мы помним, что символы в строках имеют свой индекс, по которым, собственно, и про-

ходится цикл **for**, начиная с нулевого (первой буквы) и заканчивая последним. По индексу рассчитываются кортежи и списки, элементы которых также можно перебрать по очереди.

Листинг 9.21

```
spisok = ["Вишня", "Арбуз", "Апельсин"]
for i in spisok: # Проходимся по элементам списка
    print(i)
# Результат:
# Вишня
# Арбуз
# Апельсин
```

С помощью оператора **break**, как и в цикле *while*, мы можем завершить перебор, когда сработает определенное условие.

Листинг 9.22

```
for i in "яблоко":
    if i == "л": # Если переменная равна "л"
        break
    print(i)
print(f"Текущее значение: {i}")
```

Оператор **continue** позволяет пропустить одну итерацию, в результате чего один символ будет пропущен:

Листинг 9.23

```
for i in "яблоко":
    if i == "л": # Символ "л" будет пропущен
        continue
    print(i)
```

При обходе пар в словаре с помощью цикла **for** в переменную автоматически добавляются ключи:

Листинг 9.24

```
slovar = {"Андрей":30, "Иван":26, "Демид":20}
for i in slovar:
```

```
    print(i)
# Результат:
# Андрей
# Иван
# Демид
```

Если мы хотим отображать значение ключей, необходимо добавить переменную, в которую будут заноситься эти значения, а также разбить пары на отдельные кортежи методом ***items***, чтобы цикл мог последовательно работать с каждым.

Листинг 9.25

```
slovar = {"Андрей":30, "Иван":26, "Демид":20}
for i, j in slovar.items():
    print(f"{i} {j}")
# Результат:
# Андрей: 30
# Иван: 26
# Демид: 20
```

Мы также можем создавать циклы внутри других циклов. Такие циклы называются **вложенными**. В предыдущем разделе приводился пример вывода таблицы умножения на 9.

Листинг 9.26

```
for i in range(1, 11):
    print(f"9 x {i} = {9 * i}")
# Результат:
# 9 x 1 = 9
# 9 x 2 = 18
# 9 x 3 = 27
# 9 x 4 = 36
# 9 x 5 = 45
# 9 x 6 = 54
# 9 x 7 = 63
# 9 x 8 = 72
# 9 x 9 = 81
# 9 x 10 = 90
```

Немного отредактировав данный цикл, можно выводить аналогичные таблички для каждого числа от 2 до 9. То есть, скопировав вышеуказанный

код и изменив в нем множитель, мы получим восемь табличек для каждого числа.

Листинг 9.27

```
for i in range(1, 11):
    print(f"2 x {i} = {2 * i}")
for i in range(1, 11):
    print(f"3 x {i} = {3 * i}")
for i in range(1, 11):
    print(f"4 x {i} = {4 * i}")
# И так далее до девяти
```

Но все эти восемь циклов по своей структуре повторяют друг друга, поэтому их можно сделать элементами одного общего цикла, который внутри себя имеет вложенный цикл для создания отдельных таблиц. Где после завершения цикла для одной строки/столбца заканчивается одна итерация внешнего цикла, и он запускает вложенный цикл для создания следующей строки/столбца с другим множителем.

Листинг 9.28

```
for i in range(1, 11):
    for j in range(2, 10):
        print(f"{j} x {i} = {i * j}", end="  |  ")
    print()
# Результат:
# 2 x 1 = 2  |  3 x 1 = 3  |  4 x 1 = 4  | ...
# 2 x 2 = 4  |  3 x 2 = 6  |  4 x 2 = 8  | ...
# 2 x 3 = 6  |  3 x 3 = 9  |  4 x 3 = 12  | ...
# 2 x 4 = 8  |  3 x 4 = 12  |  4 x 4 = 16  | ...
# 2 x 5 = 10  |  3 x 5 = 15  |  4 x 5 = 20  | ...
# 2 x 6 = 12  |  3 x 6 = 18  |  4 x 6 = 24  | ...
# 2 x 7 = 14  |  3 x 7 = 21  |  4 x 7 = 28  | ...
# 2 x 8 = 16  |  3 x 8 = 24  |  4 x 8 = 32  | ...
# 2 x 9 = 18  |  3 x 9 = 27  |  4 x 9 = 36  | ...
# 2 x 10 = 20  |  3 x 10 = 30  |  4 x 10 = 40  | ...
```

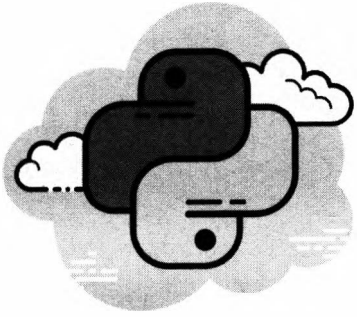
- В первой строке у нас располагается внешний цикл, где переменная *i* принимает значения от 1 до 10, определяя число, которое будет множителем.

- Во второй строке запускается вложенный цикл, где в переменную *j* поочередно сохраняются числа от 2 до 9. Это числа, по которым будут строиться таблицы умножения.
- Внутри вложенного цикла выводится операция, в которой число *j* умножается на множитель *i* (2 x 1). Параметр *end*= в функции **print** по умолчанию содержит экранированный символ \n, переносящий следующий текст на новую строку. Но мы заменили его на вертикальную черту, поэтому строка не переносится, а следующие данные выводятся тут же, после указанного разделителя.
- На текущем моменте внутренняя итерация завершена, вложенный цикл начинается сначала, где переменная *j* принимает значение 3 (3 x 1). Далее цикл повторяется до девяти включительно, продолжая строку. После чего обход всех чисел вложенного цикла завершается, и интерпретатор переходит к четвертой строке print(), которая является частью внешнего цикла.
- Строка print() по умолчанию содержит экранированный символ \n, благодаря чему остальные данные переносятся на новую строку. Здесь завершается итерация внешнего цикла, и переменная *i* принимает следующее значение — 2.
- Снова запускается вложенный цикл, где переменная *j*, поочередно принимая значения от 2 до 9, умножается на 2, формируя вторую строку общей таблицы (2 x 2 | 3 x 2 | 4 x 2 | и т.д.).

9.4.1. Практические задания

1. С помощью цикла ***for*** и функции ***range*** выведите на экран только двузначные числа.
2. Предложите пользователю ввести два числа, проверьте, чтобы первое число было меньше второго, а затем выведите на экран диапазон чисел между этими числами.
3. Есть список, содержащий большое количество строк: ["кот", "пес", "кот", "кит", "пес", "ерш", "кит"], необходимо с помощью цикла for и методов списков удалить из него все дубликаты.

4. С помощью вложенных циклов создайте таблицу из пяти столбцов и четырех строк. В каждую из "ячеек" поместите переменную со значением "x".



Глава 10.

Функции

10.1. Понятие функции

Функция — это блок кода, который выполняет определенные действия.

Ее можно сравнить с инструментом, который предназначен для решения одной или нескольких задач. Например, у нас есть шуруповерт, который мы используем каждый раз, когда нам требуется что-то закрутить, а в остальное время он лежит и бездействует.

Чтобы создать функцию, необходимо указать ключевое слово **def** (сокращение от *definition* — определение), затем присваивается уникальное название, по которому мы будем ее вызывать, а также проставляются скобочки и двоеточие.

В скобочках могут быть прописаны аргументы, о которых речь пойдет в следующем разделе. А двоеточие указывает, что функция включает в себя блок кода, состоящий из определенных инструкций. То есть при вызове функции будут выполняться все инструкции, которые в ней прописаны.

Листинг 10.1

```
def primer():
    print("Задача этой функции")
    print("Выводить текст на экран")
```

Если запустить скрипт с этой функцией, то ничего не произойдет. Так как здесь мы только определили функцию, сообщив интерпретатору, что у нас есть "шуруповерт", задача которого выводить две строки на экран. Сейчас он лежит без дела.

Чтобы воспользоваться нашим инструментом, необходимо обратиться к нему в коде, прописав его название:

Листинг 10.2

```
def primer(): # Определили функцию
    print("Задача этой функции")
    print("Выводить текст на экран")

print("Здесь начинается программа")
primer() # Вызвали функцию
print("Программа завершена")
# Результат:
# Здесь начинается программа
# Задача этой функции
# Выводить текст на экран
# Программа завершена
```

Таким образом, в любом месте нашей программы мы можем вызывать эту функцию любое количество раз. Это значительно сокращает объем кода, так как нам не нужно постоянно прописывать одни и те же инструкции.

Обратите внимание, что функции всегда прописываются со скобочками, даже если они не содержат аргументов. Когда интерпретатор видит имя функции, за которым следуют скобки, он понимает, что это не просто ссылка на функцию, а команда на ее выполнение. Кроме того, это избавляет от путаницы при чтении кода, давая понять, где функция, а где просто переменная.

Также следует иметь в виду, что блок функции должен быть создан выше, чем ее вызов. Помним, что Python проходится по коду сверху вниз. И для начала он должен узнать о существовании функции, прежде чем начнет работать с ней. Поэтому, если вначале кода вы вызовете функцию, а тело функции находится где-то ниже, интерпретатор сообщит вам, что ему не известно данное название объекта.

10.2. Аргументы функции

Функции становятся еще полезнее, когда они принимают **аргументы** — входные данные, которые мы передаем при вызове функции.

Например, создадим функцию, которая приветствует пользователя по имени:

Листинг 10.3

```
def greeting(name):
    print(f"Привет, {name}!")

greeting("Иван") # Вызвали функцию и передали аргумент
# Результат: Привет, Иван!
```

В скобочках нашей функции прописан один параметр (name), название которого можно придумать любое. Затем в коде, вызывая функцию, в скобках мы передали аргумент "Иван". Это имя автоматически подставляется в параметр name. По ходу работы программы функцию можно вызывать много раз, постоянно передавая разные имена пользователей в качестве аргумента. Таким образом, мы получили универсальный инструмент, который в зависимости от аргумента будет выдавать разный результат.

Аргументов у функции может быть любое количество. Например, мы можем передавать два аргумента, а внутри функции манипулировать ими:

Листинг 10.4

```
def summa(x, y): # Функция принимает два числа
    return x + y # и складывает их

number_1 = int(input("Введите первое число: "))
number_2 = int(input("Введите второе число: "))
print (f"Сумма чисел: {summa(number_1, number_2)}")
```

В последней строке мы вызвали функцию, передав ей две переменные с числами, которые подставились в параметры x и y соответственно. Внутри функции они сложены между собой, а затем результат возвращается командой *return*. Оператор **return** позволяет функции возвращать значение, которое потом можно использовать в коде. Если не использовать return, функция по умолчанию вернет значение None.

Ранее вы могли заметить, что результат работы функции также выводился на экран с помощью команды *print*. Иногда может возникнуть вопрос: в чем разница между *print* и *return*?

Листинг 10.5

```
def summa(x, y):
    print (x + y)
# В чем разница?
def summa(x, y):
    return (x + y)
```

Когда в функции используется *print*, результат просто выводится на экран. Мы не можем сохранить результат этих вычислений, потому как по умолчанию возвращается значение None.

Листинг 10.6

```
x,y = 1, 2
def summa(x, y):
    print (x + y)
z = summa(x, y) # Сохраняем в переменной результат вычислений
print(z) # Смотрим содержимое переменной: None
```

На экран в отдельной строке будет выведен ответ, но в переменной ничего не сохранится. То есть результатом работы функции мы не сможем воспользоваться и применить его где-то еще. Чтобы сохранить результат, нужно возвращать его оператором **return**.

Листинг 10.7

```
x,y = 1, 2
def summa(x, y):
    return (x + y)
z = summa(x, y) # Сохраняем в переменной результат вычислений
print(z) # Смотрим содержимое переменной: 3
```

Иногда в функции можно задать значение параметров по умолчанию. Это значит, что если при вызове функции аргумент не передан, то используется значение по умолчанию. Для примера создадим функцию, которая приветствует пользователя по имени, а если имя не указано, то будет отображаться значение "гость".

Листинг 10.8

```
def greeting(name="гость"):
    print(f"Привет, {name}!")

greeting("Иван") # Вызвали функцию, передав аргумент
greeting() # Второй вызов без аргумента
# Результат:
# Привет, Иван!
# Привет, гость!
```

То есть, когда мы определяем функцию и указываем в ней параметры, мы создаем как бы "пустые ячейки" для данных, которые будут заполнены во время вызова функции передачей аргумента. Но эти ячейки могут содержать значения по умолчанию.

Стоит иметь в виду, что аргументы "живут" только внутри функции. То есть они используются только внутри нее. После того как функция завершит свою работу, эти аргументы перестают существовать. И когда мы обращаемся к функции еще раз, она уже не помнит переданных ранее аргументов, поэтому если есть значение по умолчанию, оно будет использовано.

Если значения по умолчанию нет, но функция имеет какой-то параметр, при вызове ее без аргумента будет получена ошибка, сообщающая об этом. Также будет получена ошибка, если передать в функцию больше аргументов, чем она способна получать по умолчанию.

Листинг 10.9

```
def summa(x, y):
    return (x + y)
```

В примере выше мы создали функцию, которая по умолчанию принимает два аргумента. Поэтому, передавая ей меньше или больше аргументов, получим соответствующую ошибку.

10.3. Область видимости переменных

Область видимости определяет, где переменная видна и доступна для использования в коде.

Она помогает "ограничить" доступ к переменным в зависимости от того, где и как они были объявлены.

Представьте, что переменные — это небольшие коробки с информацией, и каждая из них находится в каком-то помещении. Если вы в одном помещении, то видите только те коробки (переменные), которые находятся с вами в комнате, но не те, что находятся в другом помещении. Так же в Python: каждая переменная может быть видна только в определенном месте программы

Всего существует четыре основных области видимости. Их часто называют правилом LEGB, где каждая буква обозначает область:

1. Local — локальная область.
2. Enclosing — область охватывающей функции.
3. Global — глобальная область.
4. Built-in — встроенная область.

10.3.1. Локальная область (Local)

Локальная область — это область внутри функции. Переменные, объявленные внутри функции, называются **локальными** переменными и видны только в этой функции. Они создаются в момент вызова функции и исчезают после завершения ее работы.

Листинг 10.10

```
def primer(): # Создали функцию
    x = 5 # Создали локальную переменную "x"
    print(x) # Отобразили переменную

primer() # Вызвали функцию — результат: 5
print(x)  # Ошибка! Обращение к переменной невозможно
```

Если мы попытаемся вызвать переменную "x" вне функции, то получим ошибку, так как интерпретатор не знает о существовании этой переменной. Он узнает о ней только во время работы с функцией, где эта локальная переменная объявляется. По завершении работы с функцией информация о переменной "забывается".

10.3.2. Охватывающая область (Enclosing)

Иногда у нас есть функция, созданная внутри другой функции. В этом случае вложенная функция может "видеть" переменные своей охватывающей функции (внешней функции). Переменные из охватывающей функции называются **переменными охватывающей области** (Enclosing).

Листинг 10.11

```
def внешняя_функция():
    переменная_внешней = "Переменная внешней функции"
    def вложенная_функция():
        print(переменная_внешней) # Отобразили внешнюю переменную
    вложенная_функция() # Внешняя функция вызвала вложенную функцию

внешняя_функция() # Вызвали внешнюю функцию, которая вызовет
вложенную, та, в свою очередь, отобразит переменную
```

В примере выше вложенная функция "видит" переменную внешней функции, объявленную в охватывающей функции. Это работает благодаря правилу LEGB: если переменная не найдена в локальной области (L) внутри функции, Python ищет ее в охватывающей области (E). Если и там нет переменной, он ищет ее в глобальной области (G) и далее во встроенной области (B).

10.3.3. Глобальная область (Global)

Глобальные переменные объявляются вне функций и доступны для всего модуля (файла) программы. Это те переменные, которые мы создавали ранее, до знакомства с темой функций. Переменные, объявленные на уровне модуля, можно использовать внутри функций, если они не пересекаются с локальными переменными.

То есть, если у нас в файле создана переменная x = 5, а внутри функции мы создадим переменную x = 7, это будут две разные переменные, не влияю-

щие друг на друга. Если внутри функции будут проводиться манипуляции с переменной (например, x + 10), это не будет влиять на состояние глобальной переменной. Она по-прежнему будет хранить значение 5.

Листинг 10.12

```
x = 5 # Глобальная переменная
def primer():
    x = 7 # Локальная переменная
    return x + 10 # Прибавили 10 к локальной переменной

print(x) # При обращении к переменной отображается глобальная
переменная (5)
print(primer()) # При обращении к функции отображается локальная
переменная(17)
```

Однако, если нам потребуется изменить глобальную переменную внутри функции, необходимо использовать ключевое слово *global*.

Листинг 10.13

```
x = 5 # Глобальная переменная
def primer():
    global x # Указываем, что работаем с глобальной переменной
    x = x + 10 # Это больше не локальная переменная

print(x) # Значение переменной до запуска функции (5)
primer() # Запустили функцию
print(x) # Значение переменной после запуска функции (15)
```

Ключевое слово *global* позволяет интерпретатору понять, что мы хотим изменить переменную, объявленную на уровне модуля, а не создавать новую локальную переменную.

10.3.4. Встроенная область (Built-in)

Встроенные функции и **переменные** — это те, что уже есть в Python по умолчанию, такие как *print*, *len*, *str* и т.д. Эти функции и переменные находятся в так называемой встроенной области.

Когда мы используем их, они берутся именно из этой встроенной области. Например, при вызове функции print() Пайтон сначала проверяет в порядке очереди, есть ли такое имя в локальной (L), охватывающей (E) и глобальной областях (G). Если функции с таким именем не находится, то Python обращается к встроенной области (B).

То есть, если мы создадим свою функцию ***print***, то интерпретатор сначала увидит и воспользуется ей, но не увидит встроенную функцию. В итоге мы потеряем такой инструмент, как вывод данных на экран. То же самое касается и других встроенных функций. Поэтому, как уже писалось в одной из первых глав, при выборе имени для переменных и функций не стоит называть их именами встроенных функций.

Пример работы LEGB:

social.py file_1.py

```
глобальная_переменная = "Это глобальная переменная"

def внешняя_функция():
    переменная_внешней = "Это переменная внешней функции"

    def вложенная_функция():
        переменная_вложенной = "Это локальная переменная вложенной функции"

        print(переменная_вложенной)  # Локальная область
        print(переменная_внешней)  # Охватывающая область
        print(глобальная_переменная)  # Глобальная область

    вложенная_функция() # Внешняя функция вызывает вложенную

внешняя_функция() # Вызываем внешнюю функцию
# Результат:
# Это локальная переменная вложенной функции
# Это переменная внешней функции
# Это глобальная переменная
```

Изображение 10.1.

На изображении **вложенная_функция** находит сначала свою локальную переменную **переменная_вложенной**, затем **переменная_внешней** берется из охватывающей функции, так как у себя она не имеет такой переменной. И наконец, **глобальная_переменная** берется из глобальной области, так как во вложенной функции, и далее во внешней, эта переменная не найдена.

Таким образом, области видимости помогают организовать переменные, их использование и защищают от случайного переопределения значений.

10.4. Сопоставление и порядок аргументов

Когда мы пишем функцию, нам часто нужно передавать ей данные (аргументы), которые она будет обрабатывать. Эти данные прописываются при вызове функции и подразделяются на **позиционные** и **именованные** аргументы.

Позиционные аргументы — значения передаются по позиции, то есть в том порядке, в каком они указаны при объявлении функции.

Именованные аргументы — значения передаются по имени аргумента, то есть когда мы явно указываем, какому параметру соответствует значение.

Позиционные аргументы передаются в том порядке, в каком они указаны в функции. То есть первое переданное значение соответствует первому параметру, второе — второму и так далее.

Листинг 10.14

```
def greeting(name, age):
    print(f"Привет, {name}! Тебе {age} лет.")

greeting("Иван", 30) # Передали два аргумента в name, age
```

В примере первый аргумент "Иван" передается в значение *name*, а второй 30 — в значение *age*. Порядок важен! Если мы поменяем значения местами, результат изменится.

Листинг 10.15

```
greeting(30, "Иван") # Возраст и имя перепутаны
# Результат: Привет, 30! Тебе Иван лет.
```

В результате получится что-то странное. Это пример того, что позиционные аргументы требуют внимательного соблюдения порядка.

С именованными аргументами порядок не так важен. Вместо того чтобы полагаться на позицию, мы явно указываем, какому параметру соответствует значение.

Листинг 10.16

```
greeting(name="Иван", age=30)
```

В таком виде порядок больше не имеет значения. Мы можем передать аргументы в любом порядке, главное указать имя каждого параметра.

Листинг 10.17

```
greeting(age=30, name="Иван") # Поменяли местами, результат тот же
# Привет, Иван! Тебе 30 лет.
```

Использование именованных аргументов делает код более читаемым и понятным. Например, если у функции много параметров, именованные аргументы помогают избежать ошибок. Также они полезны, когда нужно указать только некоторые из аргументов, а остальные оставить с их значениями по умолчанию.

Листинг 10.18

```
def greeting(name, age=25): # age имеет параметр по умолчанию
    print(f"Привет, {name}! Тебе {age} лет.")
```

Мы можем смешивать позиционные и именованные аргументы, но с некоторыми ограничениями. Позиционные аргументы всегда должны идти **перед** именованными. То есть мы не можем сначала передать именованный аргумент, а затем позиционный — это вызовет ошибку.

Листинг 10.19

```
greeting(age=30, "Иван") # Вызовет ошибку
greeting("Иван", age=30) # Правильный вариант
```

Пример использования разных типов аргументов:

Листинг 10.20

```
def user_info(name, age, city="Москва"):
    print(f"{name} - {age} лет, из города {city}.")

user_info("Иван", 30)           # Иван - 30 лет, из города Москвы.
user_info("Демид", 25, "Сочи") # Демид - 25 лет, из города Сочи.
user_info(age=22, name="Олег") # Олег - 22 лет, из города Москвы.
```

В первом вызове мы используем только позиционные аргументы.

Во втором — все аргументы позиционные.

В третьем вызове мы применяем именованные аргументы, чтобы указать возраст и имя, а город остается по умолчанию.

10.5. Специальные аргументы *args и **kwargs

В функцию можно передавать произвольное количество аргументов. За это отвечают специальные аргументы *args и **kwargs. Они позволяют передавать в функцию любое количество позиционных и именованных аргументов соответственно.

***args —** позволяет функции принимать произвольное количество **позиционных аргументов**. То есть таких, которые идут в функцию просто списком, без имени. Обычно это удобно, когда заранее неизвестно, какое количество аргументов будет передано.

Когда мы используем *args в определении функции, Python собирает все дополнительные переданные позиционные аргументы в один кортеж.

Листинг 10.21

```
def summa(*args): # Функция примет любое количество чисел
    total = 0 # Локальная переменная со значением по умолчанию
    for num in args: # Цикл переберет все полученные числа
        total += num # К переменной будет добавлено каждое из чисел
    return total # Результат будет возвращен

# Вызываем функцию и передаем ей произвольное количество чисел
print(summa(5, 14, 18, 21)) # Результат: 58
print(summa(11, 13, 99)) # 123
print(summa(4, 81)) # 85
```

В таком виде мы создали простой калькулятор, который складывает все полученные числа.

Название **args** — это просто общепринятое имя. Вместо него можно использовать любое слово, главное поставить перед ним звездочку *, которая сообщит интерпретатору, что мы хотим собрать все позиционные аргументы в один кортеж.

Листинг 10.22

```
def print_numbers(*numbers): # Вместо args используем другое название
    for num in numbers:
        print(num)

print_numbers(15, 28, 172)
```

****kwargs — позволяет функции принимать произвольное количество именованных аргументов.** То есть таких, которые идут в формате ключ=значение.

Когда мы используем **kwargs в определении функции, Пайтон собирает все переданные именованные аргументы в словарь, где ключи — это имена аргументов, а значения — их переданные значения.

Листинг 10.23

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_info(name="Иван", age=30, city="Москва")
# Результат:
# name: Иван
# age: 30
# city: Москва
```

Как и в случае с *args, здесь название **kwargs** является просто общепринятым, но часто его используют для ясности, где **kwargs** расшифровывается как *keyword arguments* — именованные аргументы.

Мы можем использовать одновременно и *args, и **kwargs в одной функции. В этом случае позиционные аргументы соберутся в **args**, а именованные — в **kwargs**.

Листинг 10.24

```
def info(name, age, *args, **kwargs):
    print(f"Имя: {name}")
    print(f"Возраст: {age}")
    print("Дополнительные аргументы (args):")
    for arg in args:
        print(arg)

    print("Дополнительные именованные аргументы (kwargs):")
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Передаем в функцию большое количество информации
info("Иван", 30, "Профессия - повар", "Хобби - футбол",
Город="Москва", Статус="Холост")
```

Важно помнить, что при использовании *args и **kwargs они должны идти в определенном порядке.

1. Сначала прописываются обычные позиционные аргументы.
2. Далее *args для произвольного количества позиционных аргументов.
3. Затем обычные именованные аргументы.
4. В конце — **kwargs для произвольного количества именованных аргументов.

Такой порядок нужен для того, чтобы интерпретатор мог корректно распознать, какие аргументы к каким параметрам относятся.

10.6. Функция *enumerate*

Функция **enumerate** является полезным инструментом, помогающим при работе с последовательностями, такими как списки или строки. В частности, когда нам нужно получить доступ не только к элементам, но и к их индексам.

Допустим, у нас есть список имен и нам нужно вывести каждое имя вместе с его порядковым номером. Мы можем сделать это с помощью цикла for, отслеживая номер элемента вручную, но это будет не самым удобным и простым способом. Функция ***enumerate*** упрощает задачу, она автоматически создает для нас пары из индекса и значения, которые мы можем легко использовать в цикле.

Листинг 10.25

```
names = ["Иван", "Демид", "Андрей", "Константин"]
for index, name in enumerate(names):
    print(index, name)
# Результат:
# 0 Иван
# 1 Демид
# 2 Андрей
# 3 Константин
```

Теперь каждый элемент выведен с его индексом, и нам нет необходимости отслеживать этот индекс вручную. Цикл подставляет в переменные *index* и *name* пары индекса и имени из списка *names*. Эти пары были созданы функцией ***enumerate***. Далее первая пара выводится на экран, и цикл начинается сначала, пока не закончатся все пары из списка.

Иногда бывает полезно, чтобы нумерация начиналась не с 0, а с другого значения, например с 1. Это можно сделать, используя параметр *start*.

Листинг 10.26

```
names = ["Иван", "Демид", "Андрей", "Константин"]
for index, name in enumerate(names, 1): # 1 - второй параметр start
    print(index, name)
```

Добавив в функцию второй параметр, мы указываем, с какого числа следует начинать расчет. В таком виде функция часто используется, чтобы отслеживать, какой элемент на каком месте находится, а также для более привычного отображения списков.

10.7. Функция *isinstance*

Функции **isinstance** позволяет проверить, относится ли объект к определенному типу данных. Например, когда нужно узнать, является ли переменная числом, строкой, списком или каким-то другим типом. После проверки функция возвращает True, если объект принадлежит к указанному типу, и False — если нет.

Листинг 10.27

```
x = 10
print(isinstance(x, int)) # Результат: True
```

В функцию мы передаем объект, который нужно проверить, а также тип объекта, с которым происходит сверка. То есть мы проверяем, является ли число 10 целым числом (int). Сравнение верное, поэтому результат True.

Данная функция помогает написать более гибкий и безопасный код. Иногда есть необходимость, чтобы определенный фрагмент кода выполнялся только с конкретными типами данных. Например, если у нас есть функция, принимающая только числа, то с помощью isinstance() можно проверить тип и избежать ошибок, если передан неподходящий аргумент (например, строка).

Листинг 10.28

```
def number(x):
    if isinstance(x, int): # Если переменная является числом
        return x * 2 # Умножаем его на 2
    else:
        print("Ошибка: аргумент должен быть числом.")

print(number(10))        # Результат: 20
print(number("десять")) # Ошибка: аргумент должен быть числом.
```

Когда нужно проверить сразу нескольких типов, например, является ли аргумент целым числом или числом с плавающей точкой (float), мы можем передать кортеж с несколькими типами.

Листинг 10.29

```
x = 10.5
print(isinstance(x, (int, float))) # Результат: True
```

Здесь ***isinstance*** вернет True, если *x* является либо целым числом (int) либо числом с плавающей точкой (float).

В некоторых случаях можно встретить проверку типа через функцию ***type***:

Листинг 10.30

```
x = 10
print(type(x) == int) # Результат: True
```

Хотя это работает аналогично, функция ***isinstance*** является более подходящим вариантом, так как проверяет не только точный тип, но и учитывает классы-наследники. А также может проверять одновременно несколько типов.

10.8. Функции *all* и *any*

Из раздела **5.3. Логический тип Bool** мы узнали, что Python преобразует любые типы данных в булево значение (True или False). Напомню, что пустые списки, строки, кортежи, словари, число 0 и None преобразовываются в значение False. И, соответственно, непустые структуры преобразуются в True.

Функция **all** (с английского — "все") позволяет проверить **все** элементы коллекции (списка, кортежа, множества и т. д.) на предмет того, являются ли они истинными или ложными. То есть, имея список с различными данными, функция пройдется по каждому элементу в нем и определит, является ли каждый из них заполненным (True) или пустым (False).

Если **все** элементы списка будут иметь значение True, общий результат функции также будет True. Если хотя бы один элемент будет в значении False — общий ответ также будет False.

Листинг 10.31

```
data_1 = [1, 2.4, "три", "апельсин", True]
print(all(data_1)) # Результат: True

data_2 = [0, 2.4, "", "апельсин", True]
print(all(data_2)) # Результат: False
```

В первом примере все элементы преобразовываются в True, поэтому общий результат для списка аналогичный.

Во втором примере первый элемент является 0, а третий — пустой строкой. То есть не все элементы списка преобразовываются в значение True. В итоге функция возвращает False.

Функция **any** (с английского — "какой-нибудь") проверяет, является ли хотя бы **какой-нибудь** элемент в коллекции истинным. То есть функция вернет True, если всего один элемент из списка примет истинное значение. Ответ False будет только в случае, когда все элементы могут быть преобразованы в ложное значение.

Листинг 10.32

```
data_3 = [0, None, "", "апельсин", False]
print(any(data_3)) # Результат: True

data_4 = [0, None, "", 0, False]
print(any(data_4)) # Результат: False
```

Как понятно из примеров выше, функции ***all*** и ***any*** могут быть полезны, когда нужно сразу проверить большое количество элементов.

Например, при фильтрации данных можно отбирать только те элементы, которые подпадают под определенные условия.

Листинг 10.33

```
passwords = ["gthtzsd12ws", "fgg67df4d", "dfb4"]
print(all(len(password) >= 8 for password in passwords))
# Результат: False
```

В примере выше происходит проверка, являются ли все строки в списке длиннее восьми символов. Одна из них слишком короткая, поэтому результат False.

10.9. Функция *lambda*

Функция **lambda**, также известная как анонимная функция, это функция без имени, которую мы можем создать для выполнения небольшого действия. Она может принимать любое количество аргументов, но выполнять только одно выражение.

Часто такие функции нужны, когда мы хотим использовать их один раз и не определять их в коде через **def**.

Листинг 10.34

```
summa = lambda x, y, z: x + y + z
print(summa(5,7,9)) # Результат: 21
```

lambda — это ключевое слово, которое сообщает интерпретатору, что мы собираемся создать лямбда-функцию. Сразу после нее идут аргументы, которая она может принимать. А после двоеточия — действие, которое она выполняет.

Так как функция не может иметь имени, мы сохраняем ее в переменную summa для удобства работы с ней. Данную функцию можно записать стандартным способом:

Листинг 10.35

```
def summa(x, y, z):
    return x + y + z

print(summa(5,7,9)) # Результат: 21
```

Как видим, благодаря записи lambda-функции в одну строку можно сделать код более лаконичным.

Однако, если ваша функция становится сложной или производит много различных действий, лучше использовать обычную функцию с ***def***, так как это сделает код более понятным. Также, если вам нужно использовать функцию в нескольких местах программы, лучше определить ее явно с помощью ***def*** и дать ей понятное имя.

10.10. Документирование функций

Документирование функций — важная часть написания качественного кода. Представьте, что ваш код читают коллеги или даже вы сами спустя год. Документация поможет быстро понять, что делает функция, какие аргументы принимает и что возвращает.

Когда речь идет о документировании функций, подразумевается описание работы функции, добавляемое прямо в ее определение. В Python для этого используют строку документации или, как ее еще называют, **docstring**.

Docstring пишут сразу после объявления функции. Она заключается в тройные кавычки (`"""..."""`) и содержит описание того, что делает функция. Данная строка может включать сведения об аргументах, возвращаемых значениях и даже о том, какие исключения может вызвать функция.

Листинг 10.36

```
def greet(name):
    """Функция приветствует пользователя по имени."""
    print(f"Привет, {name}!")
```

В примере ***docstring*** состоит всего из одной строки, так как функция достаточно простая и не требует дополнительных пояснений.

Когда функция становится более сложной и требует описания аргументов или возвращаемых значений, ***docstring*** лучше оформить в несколько строк. Обычно это делается в следующем формате:

1. Краткое описание функции (что она делает).

2. Аргументы, которые принимает функция (с пояснением, что они представляют и в каком формате передаются).

3. Возвращаемое значение, если функция что-то возвращает.

4. Исключения, которые могут быть вызваны в результате выполнения функции (необязательно, но полезно).

```
def factorial(n):
    """
    Вычисляет факториал заданного числа.

    Факториал числа n — это произведение всех целых чисел от 1 до n.

    Args:
        n (int): Целое число, факториал которого нужно вычислить. Должно быть неотрицательным.

    Returns:
        int: Факториал числа n.

    Raises:
        ValueError: Если n отрицательное.
    """
    if n < 0:
        raise ValueError("Факториал определён только для неотрицательных чисел.")
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

```

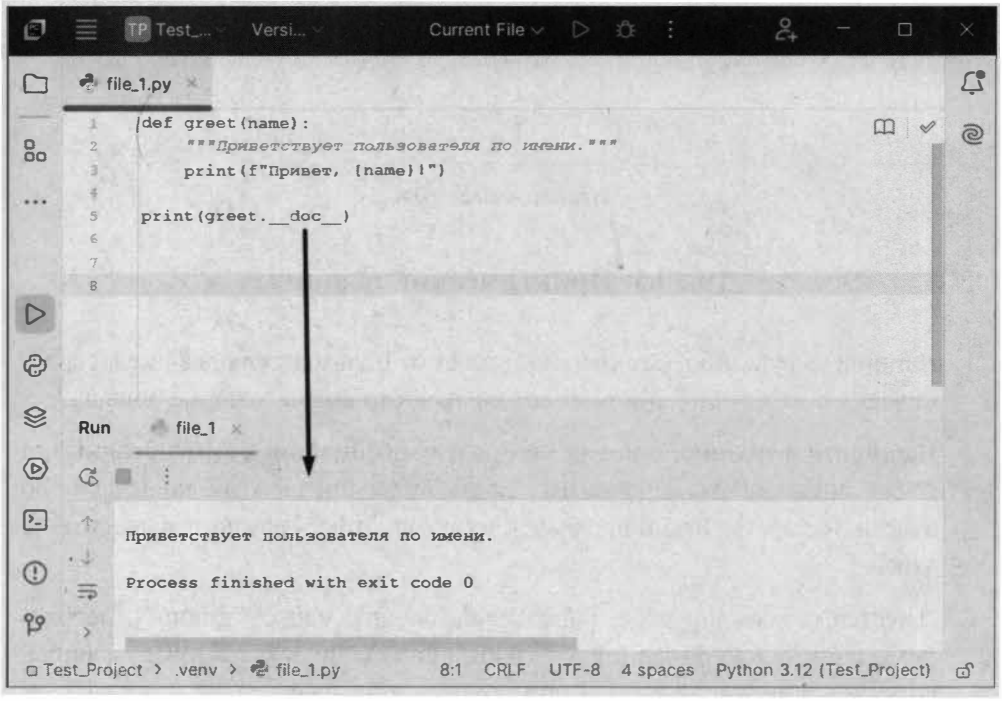
Изображение 10.4.

На изображении мы видим:

- **Краткое описание** — первое предложение сразу объясняет, что делает функция: "Вычисляет факториал заданного числа".
- **Подробное описание** — во втором абзаце поясняется, что такое факториал, чтобы было понятно, как именно работает функция.

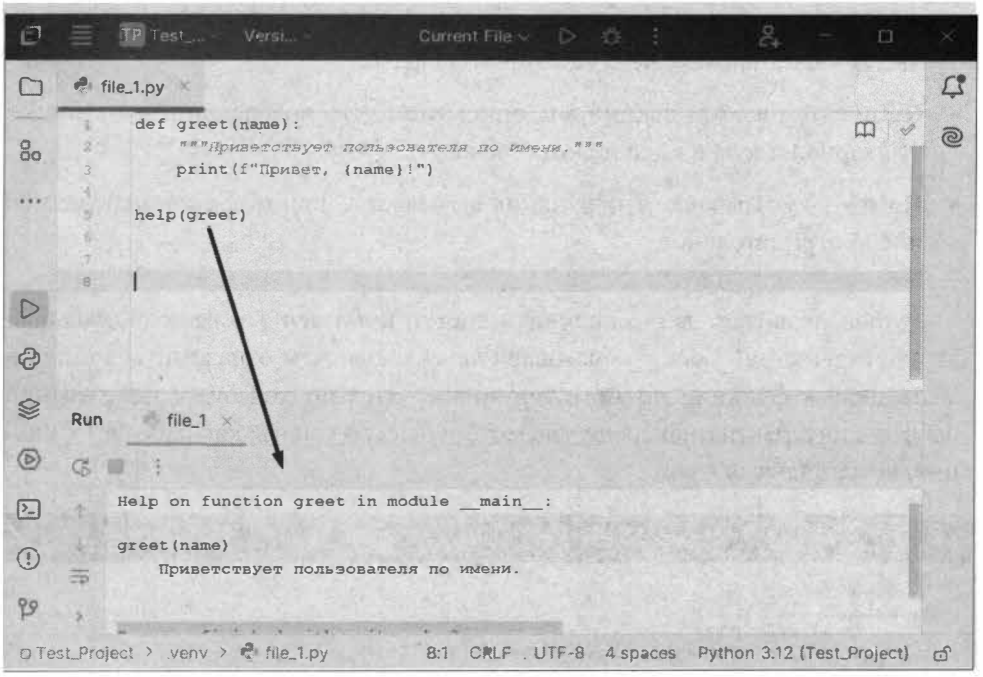
- **Args** — секция Args описывает аргумент **n**. Мы поясняем, что это целое число и оно должно быть неотрицательным.
- **Returns** — в этой секции мы описываем, что возвращает функция — факториал числа в виде целого числа.
- **Raises** — указываем, что функция вызывает ValueError, если переданное число отрицательное.

Python позволяет легко получить доступ к ***docstring*** через специальный атрибут функции __doc__, благодаря чему мы можем ознакомиться с документацией к функции прямо в терминале. Этот подход будет полезен при работе в интерактивной среде, где можно быстро узнать, как работает функция, не заглядывая в код.



Изображение 10.5.

Кроме __doc__, можно использовать функцию help() для получения информации о функции.



Изображение 10.6.

10.10.1. Практические задания

1. Напишите функцию, которая принимает от пользователя два числа, сравнивает их между собой и выводит на экран то число, которое больше.
2. Напишите функцию, которая принимает позиционные, именованные, а также специальные аргументы *args. Затем предложите пользователю ввести эти аргументы и передайте их в функцию. Результат выведите на экран.
3. Имеется список фруктов: ['апельсин', 'банан', 'манго', 'вишня'], необходимо с помощью цикла **for** и функции enumerate создать нумерованный список, начинающийся с 1 и заканчивающийся на 4.
4. Имеется список чисел: [1, 2.5, 3.1, 4, 5.0]. С помощью функций **all** и **isinstance** необходимо выяснить, являются ли все они целыми числами.

Глава 11.

Итераторы, генераторы, рекурсия

11.1. Генераторы и итераторы

Генераторы — это простой вариант создания итераторов в Пайтон.

Итератор — это объект, позволяющий поочередно перебирать элементы коллекции (списков или кортежей) по одному за раз.

Генераторы позволяют создавать такие итераторы с помощью более простой и понятной конструкции, чем стандартное использование методов и классов.

Обычная функция выполняет свои действия и возвращает результат с помощью оператора **return**. После этого функция завершает свое выполнение. Генераторы используют оператор **yield**. Когда функция встречает *yield*, она "приостанавливает" свое выполнение и возвращает текущее значение. В следующий раз, когда мы вызываем генератор, он продолжает выполнение с того места, где остановился, сохраняя свое состояние.

Листинг 11.1

```
def generator(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

Данная функция является генератором, который будет возвращать числа от 1 до *n*. Теперь, чтобы воспользоваться им, мы можем сделать так:

Листинг 11.2

```
gen = generator(5)

for num in gen:
    print(num)
# Результат:
# 1
# 2
# 3
# 4
# 5
```

Переменная *gen* выступает в роли итератора, аналогично тому, как это делает *range*. Это позволяет функции возвращать последовательные значения по одному за раз, вместо того чтобы возвращать все сразу (как это делает обычная функция с возвращением значения через *return*).

То есть генератор не хранит все значения в памяти. Вместо этого он генерирует значения по требованию, экономя память при работе с большими объемами данных. Кроме того, создание генераторов требует меньше кода и позволяет избежать сложной логики, связанной с управлением состоянием. Это также называется "ленивым вычислением".

11.2. Ленивые вычисления

Ленивые вычисления (или "отложенные вычисления") — это стиль программирования, при котором вычисления выполняются только в тот момент, когда они становятся необходимыми. То есть мы не вычисляем какое-либо значение заранее, а откладываем выполнение операций до тех пор, пока не потребуется результат.

Генераторные выражения по своей сути похожи на списковые, но они используют круглые скобки вместо квадратных и позволяют создавать генераторы на лету.

Листинг 11.3

```
# Пример генераторного выражения
squares = (x*x for x in range(10))
```

В данном примере **squares** — это генератор, который будет возвращать квадраты чисел от 0 до 9 по мере необходимости. Чтобы увидеть, как это работает на практике, мы можем использовать цикл **for**:

Листинг 11.4

```
for square in squares:
    print(square)
```

В итоге будут последовательно вычислены квадраты всех чисел. Однако не стоит забывать, что генератор будет работать только один раз. После того как мы пройдем по нему, он будет исчерпан. Если мы снова попытаемся его использовать, он не даст никаких результатов:

Листинг 11.5

```
# Пример генераторного выражения
squares = (x*x for x in range(10))

for square in squares:
    print(square)

# Повторное использование генератора в другом месте кода
print("Повторное использование:")
for square in squares:
    print(square)
# Пусто, так как генератор уже исчерпан
```

То есть последовательность уже пройдена от 0 до 9. Таким образом, генераторы и ленивые вычисления помогают работать с данными эффективно и позволяют разрабатывать более производительные программы.

11.3. Функции *map* и *filter*

Функция **map()** является встроенной функцией Python, она позволяет применять заданную функцию ко всем элементам итерируемого объекта (например, списка, кортежа или любого другого объекта, поддерживающего итерацию) и возвращает итератор с результатами.

Листинг 11.6

```
map(function, iterable, ...)
```

В данном примере *function* — это функция, которую мы хотим применить ко всем элементам. А **iterable** — это объект, элементы которого нужно обработать. Функция может принимать несколько таких объектов, но об этом немного позже.

Принцип работы ***map*** довольно прост: она берет каждый элемент переданного итерируемого объекта и поочередно применяет к ним указанную функцию. Возвращается объект-итератор, который можно преобразовать в список или другой тип коллекции, например, с помощью функции list().

Листинг 11.7

```
def square(x): # Вычисляет квадрат числа
    return x * x

# Список чисел
spisok = [1, 2, 3, 4, 5]
# Сохраняем в переменную результат работы функции map
squared_numbers = map(square, spisok)
print(list(squared_numbers)) # Результат: [1, 4, 9, 16, 25]
```

В примере функция square была применена к каждому элементу списка, а затем функция ***map*** вернула итератор, который был преобразован в новый список функцией ***list***.

Иногда, чтобы сделать код более компактным, с ***map*** используют lambda-функции:

Листинг 11.8

```
spisok = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x * x, spisok)
print(list(squared_numbers)) # Результат: [1, 4, 9, 16, 25]
```

Функция ***map*** также может работать с несколькими итерируемыми объектами одновременно, если переданная функция принимает несколько аргументов. В этом случае функция будет применяться к соответствующим элементам всех переданных итерируемых объектов.

Листинг 11.9

```
spisok_1 = [1, 2, 3]
spisok_2 = [4, 5, 6]
sums = map(lambda x, y: x + y, spisok_1, spisok_2)
print(list(sums)) # Результат: [5, 7, 9]
```

В примере мы последовательно сложили первые, вторые и третьи элементы каждого списка между собой (1 + 4, 2 + 5, 3 + 6).

Функция filter() также является встроенной функцией в Пайтон и используется для фильтрации итерируемых объектов. Она принимает два аргумента — функцию и итерируемый объект, а затем возвращает итератор, который содержит только те элементы, для которых функция имеет значение True.

Например, у нас есть список чисел и нам необходимо извлечь из него только четные числа. Вместо того, чтобы писать цикл и создавать промежуточный список, мы можем использовать ***filter*** для достижения той же цели с меньшими усилиями и более читаемым кодом.

При вызове функции ***filter*** она применяет заданную в ней функцию ко всем элементам итерируемого объекта. Если заданная функция возвращает True, элемент сохраняется в итоговом наборе, если False — изымается.

Листинг 11.10

```
# Функция для определения четности числа
def parity(num):
    return num % 2 == 0

spisok = [1, 2, 3, 4, 5]
even_numbers = filter(parity, spisok)
print(list(even_numbers)) # Результат: [2, 4]
```

В данном примере каждое число из списка передается в функцию ***parity***, где в результате сравнения возвращается ответ True или False. Затем все истинные результаты записываются в новый список.

Тот же пример с lambda-функцией:

Листинг 11.11

```
spisok = [1, 2, 3, 4, 5]
even_numbers = filter(lambda num: num % 2 == 0, spisok)
print(list(even_numbers)) # Результат: [2, 4]
```

Если передать None вместо функции, ***filter*** удалит все элементы, которые оцениваются как False (например, 0, None, ""):

Листинг 11.12

```
spisok = [0, 1, False, 2, '', 3, 'Hello', None, True]
true_values = filter(None, spisok)
print(list(true_values)) # Результат: [1, 2, 3, 'Hello', True]
```

11.4. Функция *zip*

Функция **zip()** также является встроенной функцией в Пайтон, она позволяет объединять элементы из несколько итерируемых объектов в "упаковки" — кортежи. При вызове функции она берет первый элемент из каждого переданного итерируемого объекта и объединяет их в один кортеж. Затем эта операция повторяется для вторых элементов и так далее.

Если итерируемые объекты имеют разные длины, ***zip*** завершает свою работу, когда достигает конца самого короткого из них.

Листинг 11.13

```
# Имеем два списка
names = ["Иван", "Демид", "Андрей"]
scores = [85, 92, 78]

# Применяем функцию zip и сохраняем в переменной
unite = zip(names, scores)
print(list(unite))
# Результат:
# [('Иван', 85), ('Демид', 92), ('Андрей', 78)]
```

Теперь у нас есть список кортежей, где каждый содержит имя и соответствующий результат. Как видно из примера, функция удобна для объединения связанных данных из разных списков.

Если у нас есть матрица, представляющая собой список списков, с помощью ***zip*** можно легко провести транспонирование.

Листинг 11.14

```
# Матрица — список со списками
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

transposed = list(zip(*matrix))
print(transposed)
# Результат:
# [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

То есть мы сгруппировали данные, собрав каждый столбец в отдельный кортеж. Кроме того, мы можем использовать ***zip*** в циклах для одновременной обработки нескольких последовательностей.

Листинг 11.15

```
names = ["Иван", "Демид", "Андрей"]
scores = [85, 92, 78]

for name, score in zip(names, scores):
    print(f"{name}: {score}")
# Результат:
# Иван: 85
# Демид: 92
# Андрей: 78
```

Мы также можем использовать функцию ***zip*** для создания словарей:

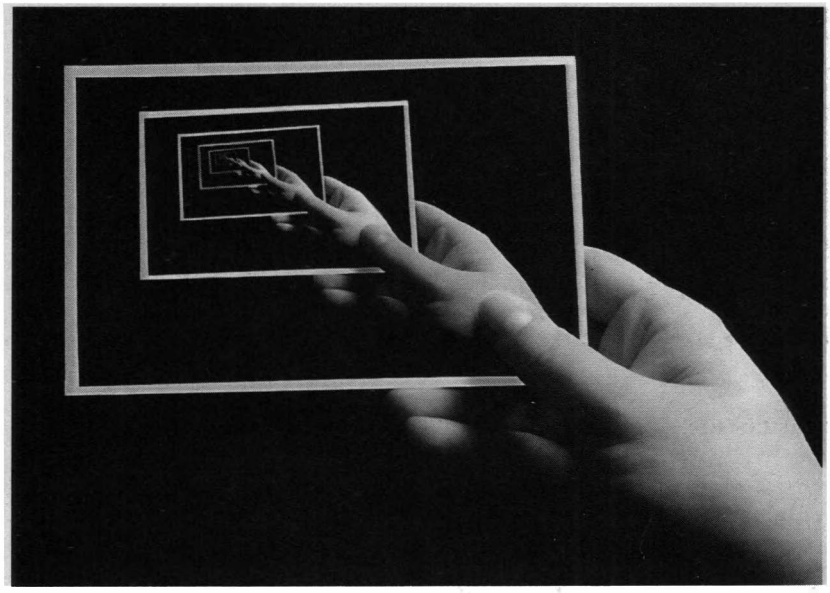
Листинг 11.16

```
names = ["Иван", "Демид", "Андрей"]
scores = [85, 92, 78]

slovar = dict(zip(names, scores))
print(slovar)
# Результат:
# {'Иван': 85, 'Демид': 92, 'Андрей': 78}
```

11.5. Рекурсивные функции

Рекурсивные функции — это один из интересных аспектов программирования, который может поначалу показаться сложным, но, поняв его, вы сможете решить многие задачи более элегантно.

Рекурсия — это метод, при котором функция вызывает сама себя. Подобная функция называется ***рекурсивной***. Главная идея заключается в том, чтобы разбить большую задачу на более мелкие подзадачи того же типа. В результате рекурсивные функции могут решать задачи, которые требуют повторяющихся действий, пока не будет достигнуто условие завершения.



Изображение 11.1.

Представьте себе традиционную матрешку. Когда мы открываем ее, внутри оказывается еще одна, меньшая, а в ней — еще одна. С каждым разом матрешка становится меньше, пока не дойдет до самой маленькой. В рекурсивных функциях происходит нечто похожее: функция вызывает саму себя с новыми, более простыми параметрами, пока не достигнет так называемого базового случая (или условия выхода), где рекурсия заканчивается.

Любая рекурсивная функция имеет две составляющие:

1. **Базовый случай** — условие, при котором функция завершает вызовы и начинает возвращать значения. Это важный момент, потому что без базового случая функция будет вызывать сама себя бесконечно (хотя в Python есть ограничение на количество рекурсий, чтобы избежать бесконечного самоповтора).
2. **Рекурсивный вызов** — часть, где функция вызывает сама себя, но с измененными параметрами, которые приближают нас к базовому случаю.

Одним из самых распространенных примеров рекурсивной функции является вычисление факториала числа. Напомню, что факториал числа — это произведение всех чисел от 1 до самого числа. Обозначается как n!.

Например:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Этот расчет можно записать как: n! = n * (n - 1)!. То есть факториал числа 5 можно представить как:

```
5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
```

и так далее, пока не дойдем до 1!, которая равна 1.

Листинг 11.17

```
def factorial(n):
    if n == 1: # Базовый случай: если n равно 1, вернем 1
        return 1
    else:
        return n * factorial(n - 1) # Рекурсивный случай
        # Здесь функция вызывает себя с аргументом n - 1
        # Результат умножается на n и возвращается

print(factorial(5)) # Вызвали функцию и передали число
# Результат: 120
```

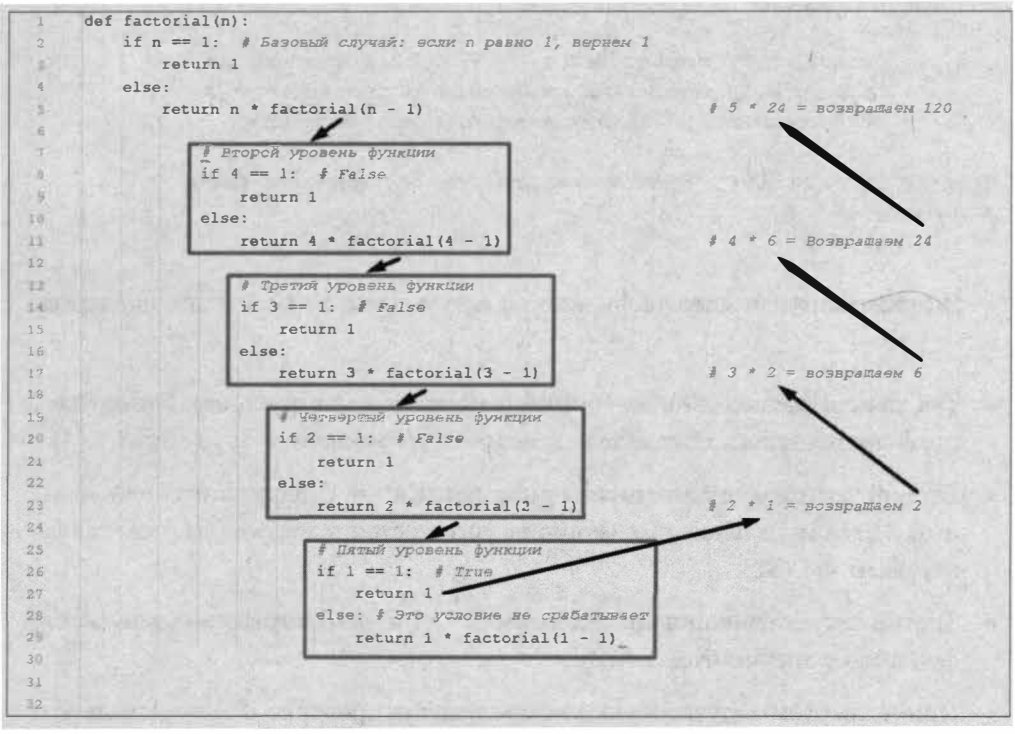
В примере после запуска функции с аргументом 5 происходит проверка if 5 == 1.

- Так как сравнение ложное, Пайтон переходит к инструкции в блоке **else**, где функция запускается еще раз, но уже с аргументом n-1, то есть 5-1 (4).
- Второй запуск функции снова проверяет if 4 == 1, результат снова ложный. Поэтому в блоке else функция запускается в третий раз, передавая аргумент 4-1 (3).
- Третий запуск функции сравнивает if 3 == 1 и, соответственно, запускает функцию с аргументом 3-1 (2).
- При четвертом запуске функции происходит проверка if 2 == 1 и далее очередной перезапуск с аргументом 2-1 (1).

- На пятой итерации сравнение становится верным 1 == 1. Поэтому условие if возвращает единицу на четвертый уровень функции. Где в блоке else она перемножается на 2 (return n * результат функции = 1). Ответ 2 возвращается на третий уровень функции.
- На третьем уровне в блоке else: return 3 * 2 (ответ: 6). Этот результат возвращается на второй уровень.
- На втором уровне в блоке else: 4 * 6 (ответ: 24). Результат возвращается на первый уровень.
- На первом уровне в блоке else: 5 * 24 (ответ: 120). Итоговый ответ возвращается на экран.

Если вы смотрели фильм "Начало" Кристофера Нолана, то можете провести аналогию с концепцией сна внутри сна. Внутри которого еще один сон.

Если визуально отобразить работу функции, то она может выглядеть так:



Изображение 11.2.

Рекурсия позволяет решать задачи, которые, естественно, выражаются через повторение. Например, задачи с ветвящимися структурами (деревья, графы) легко описываются рекурсией.

Кроме того, рекурсивный подход может упростить код и сделать его более понятным, особенно для действий, требующих многократного деления задачи на подзадачи.

К недостаткам можно отнести то, что рекурсивные функции могут быть медленнее из-за создания новых вызовов функций внутри себя и необходимости хранения промежуточных значений в памяти. Также, как упоминалось выше, у Пайтона есть ограничение на количество повторений (стек). Поэтому может возникнуть ситуация, когда стек вызовов переполнится и рекурсия не завершится вовремя.

Если вы понимаете, что задача требует глубокой рекурсии, лучше рассмотреть возможность решения с помощью цикла. Например, факториал можно легко вычислить таким образом:

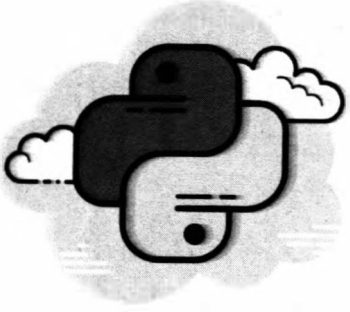
Листинг 11.18

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial(5))
```

Рекурсия может упростить код, сделать его более понятным и кратким, но важно понимать, что не для каждой задачи она подходит. Если задача слишком сложна и требует большого количества вызовов функции, лучше искать альтернативу в виде цикла или других подходов.

Для полного понимания работы рекурсивной функции рекомендуется перечитать этот раздел еще раз.

* Когда вы перечитаете раздел, вы снова увидите рекомендацию "перечитать еще раз". Это еще один пример рекурсии, где вы будете перечитывать раздел до тех пор, пока не случится базовый случай — к вам придет понимание работы рекурсивной функции.



Глава 12.

Исключения и их обработка

12.1. Понятие исключений

Исключение (exception) — это особая ситуация, которая может возникнуть во время выполнения программы, если что-то пошло не так.

Представьте, что скрипт пытается выполнить действие, которое невозможно. Например, разделить число на ноль или открыть файл, которого не существует. В такие моменты Python вызывает исключение, чтобы сообщить: "Хьюстон, у нас проблемы!"

Исключения важны, так как если их не обрабатывать, программа завершится с ошибкой. Это неудобно, особенно если ваш код должен работать в реальном проекте, например в веб-приложении или мобильной игре. Обработка исключений помогает сделать программу более устойчивой в случае непредвиденных ситуаций. Это также позволит показать пользователю понятное сообщение вместо краша программы. А разработчику сообщит, что именно пошло не так.

Когда в коде возникает ошибка, Python останавливает выполнение программы и показывает так называемый **трейсбек** (traceback) — сообщение об ошибке.

Листинг 12.1

```
print(10 / 0)
# Результат:
# ZeroDivisionError: division by zero
```

В Пайтон есть множество встроенных исключений. Самые распространенные следующие:

- **ZeroDivisionError** — попытка деления на ноль.
- **ValueError** — передача функции значения неправильного типа.
- **TypeError** — операция с несовместимыми типами.
- **IndexError** — попытка обратиться к элементу списка по несуществующему индексу.
- **KeyError** — попытка получить доступ к несуществующему ключу в словаре.
- **FileNotFoundError** — попытка открыть файл, которого нет.

Листинг 12.2

```
spisok = [1, 2, 3]
print(spisok[5]) # IndexError: элемента с таким индексом нет.
```

12.2. Блок *try-except*

Если исключения не обрабатывать, программа завершится с ошибкой. Чтобы этого избежать, нужно использовать обработку исключений.

Python предоставляет специальную конструкцию **try-except**, чтобы отлавливать и обрабатывать исключения.

Листинг 12.3

```
try:
    x = int(input("Введите число: "))
    print(10 / x)
except ZeroDivisionError:
    print("На ноль делить нельзя!")
except ValueError:
    print("Вы ввели не число!")
```

В данном примере в блоке **try** Пайтон пытается выполнить определенные инструкции. Он принимает от пользователя данные и использует их в

вычислениях. Если был введен 0, будет получена ошибка *ZeroDivisionError*, но вместо нее сработает первый блок **except**, который перехватывает ошибку и отображает соответствующее сообщение. То есть программа не вылетела, а продолжит работать, сообщив о проблеме.

Аналогичным образом сработает второй блок **except**, если вместо числа будет введено что-то другое, например слово. Если данные введены правильно, блоки **except** пропускаются. Выполняется только блок **try**.

Пример обработки нескольких ошибок вместе:

Листинг 12.4

```
try:
    x = int(input("Введите число: "))
    print(10 / x)
except (ValueError, ZeroDivisionError):
    print("Ошибка: Неправильный ввод или деление на ноль!")
```

Блоки **else** и **finally** делают обработку ошибок более структурированной. Они помогают разделять неудачные и успешные сценарии (**else**), а также гарантировать выполнение важного кода в любых обстоятельствах (**finally**).

Листинг 12.5

```
try:
    x = int(input("Введите число: "))
    result= 10 / x
except (ValueError, ZeroDivisionError):
    print("Ошибка: Неправильный ввод или деление на ноль!")
else:
    print(f"Результат: {result}")
```

Блок **else** отвечает за вывод результата только в том случае, если ошибки не возникли. Это позволяет отделить обработку ошибок от основного кода.

Блок **finally** используется для выполнения кода, который должен срабатывать всегда, независимо от того, произошла ошибка или нет. Даже если в блоке except был выход из программы.

Блок **finally** полезен, когда в любом случае необходимо выполнить завершающие действия. Например, закрыть файл, завершить соединение с базой данных, освободить ресурсы.

Листинг 12.6

```
try:
    file = open("data.txt", "r")
    data = file.read()
    print("Содержимое файла:", data)
except FileNotFoundError:
    print("Ошибка: файл не найден!")
finally:
    print("Закрываем файл.")
    file.close()
```

В примере выше мы открываем файл и читаем из него данные. Независимо от того, произошла ошибка или нет, файл нужно закрыть.

Конструкция **try-except-else-finally** может показаться сложной, но при правильном подходе она делает код чище и понятнее, так как мы можем предусмотреть все возможные случаи.

Листинг 12.7

```
try:
    x = int(input("Введите число: "))
    result = 10 / x
except ValueError:
    print("Ошибка: введено не число!")
except ZeroDivisionError:
    print("Ошибка: деление на ноль невозможно!")
else:
    print(f"Результат: {result}") # Если деление произошло
finally:
    print("Программа завершила выполнение.") # Выполняем в любом случае
```

12.4. Создание собственных исключений

Мы уже знакомы со стандартными исключениями, такими как ValueError, TypeError, ZeroDivisionError и другие. Они полезны при обработке стандартных ошибок, но иногда этих исключений недостаточно.

Представим такую ситуацию, где нам нужно разработать банковское приложение. В нем может быть особая ошибка, например "Недостаточно средств". Здесь стандартные исключения нам не подойдут, поэтому нужно создать свое собственное исключение.

Преимущество своих исключений в том, что мы можем присвоить им имя, которое точно описывает проблему. Когда возникнет ошибка, мы легко определим, где и почему это произошло. Для создания собственных исключений нужно наследоваться от класса Exception (или его дочернего класса). О классах мы поговорим в следующем разделе, а здесь рассмотрим типичный пример.

Листинг 12.8

```
class InsufficientFundsError(Exception):
    """Ошибка: недостаточно средств на счете"""
    pass
```

В примере мы создали класс с именем *InsufficientFundsError*. Этот класс наследуется от Exception. Внутри класса ничего нет, кроме строки-документации. Она нужна для описания возникшего исключения. Весь остальной функционал существует в классе Exception, он обработает ошибку.

Листинг 12.9

```
class InsufficientFundsError(Exception):
    """Ошибка: недостаточно средств на счете"""
    pass

def withdraw_money(balance, amount):
    """Функция для вывода средств
    Принимает два аргумента:
    balance - баланс на счете
```

```
    amount - сумма для вывода"""
    if amount > balance:
        # Если сумма вывода больше баланса, срабатывает исключение
        raise InsufficientFundsError(f"Недостаточно средств: баланс
{balance}, требуется {amount}")
    balance -= amount # Из баланса вычитается снятая сумма
    return balance # Возвращаем текущий баланс

try:
    current_balance = 100 # Текущий баланс
    withdrawal_amount = 200 # Сумма для снятия
    # Передаем в функцию два аргумента
    # и сохраняем результат в переменную new_balance
    new_balance = withdraw_money(current_balance, withdrawal_amount)
    print(f"Снятие успешно! Новый баланс: {new_balance}")
except InsufficientFundsError as e:
    print(e) # Выводим сообщение в случае исключения
```

Оператор **raise** в функции ***withdraw_money*** создает исключение указанного типа. Выполнение текущей функции прерывается, и Python начинает искать ближайший блок **try-except**, который может обработать это исключение.

В блоке except созданный нами класс ошибки сохраняется в переменную "e" с помощью ключевого слова **as**. А затем мы выводим на экран содержимое этой переменной print(e).

Таким образом, для каждого отдельного случая можно создать свое исключение. Когда одно или несколько из них произойдет, мы будем точно знать, в каком месте кода и по какой причине это произошло.

В больших проектах можно создавать иерархию исключений. Например:

Листинг 12.10

```
class BankError(Exception):
    """Базовый класс для всех ошибок банка"""
    pass

class InsufficientFundsError(BankError):
    """Ошибка: недостаточно средств"""
    pass

class AccountNotFoundError(BankError):
```

```
    """Ошибка: счет не найден"""
    pass
```

В нашем приложении может быть много разных ошибок, но все они относятся к одной категории. Вы можете перехватывать их сразу, обрабатывая общий класс (BankError), или отдельно обрабатывать каждую.

Листинг 12.11

```
try:
    raise AccountNotFoundError("Счет с номером 12345 не найден.")
except BankError as e:
    print(f"Произошла ошибка в банке: {e}")
# Результат:
# Произошла ошибка в банке: Счет с номером 12345 не найден.
```

12.4.1. Практические задания

1. Напишите функцию, которая делит два числа, введенных пользователем. Обработайте исключения при вводе ноля или нечислового значения. Выведите соответствующее сообщение.
2. Напишите функцию, которая принимает строку и пытается преобразовать ее в число. Если строка не является числом, выведите сообщение об ошибке.
3. Напишите программу, которая запрашивает у пользователя имя файла для чтения. Если файл не найден, выведите сообщение об ошибке.

Глава 13.

Классы и объекты

13.1. Введение в ООП

Объектно-ориентированное программирование (ООП) — это способ организации кода, при котором данные и функции объединяются в единые сущности, называемые объектами. Объекты, в свою очередь, создаются на основе классов — шаблонов или чертежей, описывающих, какими будут данные объекта (атрибуты) и поведение объекта (методы).

К примеру, у нас есть типовой чертеж дома (класс), а объекты — это построенные дома по данному чертежу или шаблону. Каждый дом может быть уникальным (различаться цветом стен, мебелью), но все они построены по одной схеме.

Преимущество ООП в том, что код становится более структурированным. Создавая классы, мы можем писать шаблоны для реализации множества похожих объектов. Это позволяет создавать программы, которые более интуитивно понятны, так как они отражают структуру реального мира. Кроме того, такой подход облегчает совместную разработку программ, так как разные разработчики могут работать над разными классами и объектами.

13.2. Создание классов и объектов

Давайте разберем, как создать класс (чертеж) на примере создания предметов в компьютерной игре. Все предметы определенного класса имеют схожие свойства. Например, все предметы, входящие в класс мечей, могут иметь общие атрибуты, такие как название модели, урон, стоимость, описание.

Листинг 13.1

```
class Sword(): # Создали класс мечей
    def __init__ (self, name, damage, cost, description):
        # Эти строки создают атрибуты объекта
        # и связывают их с переданными значениями
        self.name = name
        self.damage = damage
        self.cost = cost
        self.description = description
```

С помощью ключевого слова **class** мы сообщаем интерпретатору, что создаем шаблон для объектов с определенным названием. Обратите внимание, что имя класса пишется с заглавной буквы. Это не строгое правило, а общепринятое соглашение среди программистов.

Далее пишется функция инициализации, которая включает в себя все аргументы для объектов данного класса (общие свойства мечей).

__init__ является специальным методом (конструктором), вызываемым при создании нового объекта. Он пишется с двумя нижними подчеркиваниями в начале и в конце.

self является ссылкой на создаваемый объект, позволяющей обращаться к его атрибутам и методам внутри класса.

Аргументы, передаваемые в __init__(), используются для инициализации атрибутов объекта. То есть каждый атрибут ссылается на соответствующий аргумент функции. При создании объектов (моделей мечей) каждый из них будет иметь свои характеристики (атрибуты) на основе общего шаблона.

Когда мы изучали тему функций, мы выяснили, что они могут иметь разные параметры в скобках и при их вызове можно передавать определенные значения, которые подставляются в качестве аргументов. Теперь, при создании объектов класса, функции таким же образом имеют атрибуты с опреде-

ленными значениями, которые будут подставляться по созданному шаблону класса.

То есть на основе общего шаблона мы можем создать множество разных моделей мечей, которые будут иметь одинаковый пул параметров (имя, урон, стоимость, описание), но с разными характеристиками. Один меч может иметь урон 80, второй 100, третий — 120 и т.д.

Листинг 13.2

```
# Объекты класса Sword
sword_1 = Sword("Ржавый меч", 20, 25, "Старый, ржавый меч")
sword_2 = Sword("Одноручный", 80, 75, "Хорошо ложится в руку")
sword_3 = Sword("Полуторный", 100, 150, "Тяжеловат для одной руки")
sword_4 = Sword("Двуручный", 120, 250, "Увесистая штука")
```

В примере выше мы создали четыре разных предмета (объекта), каждый из которых входит в класс мечей. При создании мы можем дать им любое уникальное имя переменной (например, sword_1), далее мы сообщаем Пайтону, что объект входит в класс мечей (= Sword), и в скобках прописываем параметры для каждого мяча по шаблону класса.

Где первый параметр ("Ржавый меч") подставляется в атрибут *name*, второй (20) — в атрибут урона, третий (25) — в атрибут стоимости и последний — в атрибут описания предмета.

Теперь мы можем обращаться к определенным атрибутам и работать с ними с помощью точечной нотации:

Листинг 13.3

```
# Выводим описание второго меча
print(sword_2.description) # Результат: хорошо ложится в руку

# При покупке двух предметов считаем их стоимость
print(sword_3.cost + sword_4.cost) # Результат: 400 (150 + 250)
```

13.3. Применение методов

Итак, мы разобрались, что класс используется как чертеж для создания объектов с определенным набором атрибутов.

Методы — это по сути функции, которые непосредственно связаны с этим чертежом и позволяют задавать определенные действия.

Например, каждый из мечей может быть экипирован на персонажа, снят с персонажа, может быть заточен или отображен в интерфейсе. Для каждой подобной возможности можно прописать метод — небольшую функцию, которая будет отвечать за это действие.

Листинг 13.4

```
class Sword:
    def __init__(self, name, damage, cost, description, equipped=False):
        self.name = name
        self.damage = damage
        self.cost = cost
        self.description = description
        self.equipped = equipped
```

В примере мы добавили дополнительный атрибут *equipped*, который отвечает за состояние предмета — экипирован он или нет. По умолчанию все созданные предметы будут иметь статус False, то есть не надеты на персонажа. Теперь создадим метод для экипировки:

Листинг 13.5

```
def equip(self):
    if not self.equipped: # Если статус предмета не True
        self.equipped = True # Переключаем на True
        print(f"{self.name} экипирован!") # Оповещение
    else:
        print(f"{self.name} уже экипирован.")

# Обращаемся к методу equip для четвертого меча
sword_4.equip() # Результат: Двуручный экипирован!
```

Для снятия предмета создаем аналогичную функцию:

Листинг 13.6

```
def unequip(self):
    if self.equipped: # Если статус предмета True,
        self.equipped = False # переключаем на False
        print(f"{self.name} снят.") # Оповещение
    else:
        print(f"{self.name} уже снят.")

# Обращаемся к методу unequip для четвертого меча
sword_4.unequip() # Результат: Двуручный снят.
```

Теперь давайте сделаем возможность заточки мечей:

Листинг 13.7

```
def upgrade_damage(self, additional_damage):
    # Плюсуем к старому значению новое
    self.damage += additional_damage
    print(f"Урон {self.name} увеличен на {additional_damage}. Новый
урон: {self.damage}") # Оповещение

# Затачиваем Ржавый меч
sword_1.upgrade_damage(5)
# Урон Ржавый меч увеличен на 5. Новый урон: 25
```

Здесь мы с помощью созданного метода ***upgrade_damage*** передали значение 5 в параметр *additional_damage*, которое прибавилось к значению по умолчанию 20, и получили измененные характеристики у предмета. Сам предмет *sword_1* подставляется в параметр *self*, поэтому интерпретатор знает старый показатель урона этого меча.

Для отображения предмета в инвентаре или на полке магазина мы также можем создать метод, который будет выводить на экран его характеристики.

Листинг 13.8

```
def display_info(self):
    print(f"Название: {self.name}")
    print(f"Урон: {self.damage}")
    print(f"Стоимость: {self.cost}")
    print(f"Описание: {self.description}")
```

```
# Отображаем характеристики предмета
excalibur.display_info()
# Результат:
# Название: Экскалибур
# Урон: 150
# Стоимость: 1000
# Описание: Легендарный меч короля Артура
```

В итоге мы создали класс для предметов, который группирует их по определенным характеристикам и позволяет применять к ним общие шаблонные методы. Полный код выглядит следующим образом:

Листинг 13.9

```
class Sword:
    def __init__(self, name, damage, cost, description,
equipped=False):
        self.name = name
        self.damage = damage
        self.cost = cost
        self.description = description
        self.equipped = equipped

    def equip(self):
        if not self.equipped: # Если статус предмета не True,
            self.equipped = True # переключаем на True
            print(f"{self.name} экипирован!") # Оповещение
        else:
            print(f"{self.name} уже экипирован.")

    def unequip(self):
        if self.equipped: # Если статус предмета True,
            self.equipped = False # переключаем на False
            print(f"{self.name} снят.") # Оповещение
        else:
            print(f"{self.name} уже снят.")

    def upgrade_damage(self, additional_damage):
        # Плюсуем к старому значению новое
        self.damage += additional_damage
        print(f"Урон {self.name} увеличен на {additional_damage}.
Новый урон: {self.damage}")

    def display_info(self):
        print(f"Название: {self.name}")
        print(f"Урон: {self.damage}")
```

```
        print(f"Стоимость: {self.cost}")
        print(f"Описание: {self.description}")

# Объекты
sword_1 = Sword("Ржавый меч", 20, 25, "Старый, ржавый меч")
sword_2 = Sword("Одноручный", 80, 75, "Хорошо ложится в руку")
sword_3 = Sword("Полуторный", 100, 150, "Тяжеловат для одной руки")
sword_4 = Sword("Двуручный", 120, 250, "Увесистая штука")
excalibur = Sword("Экскалибур", 150, 1000, "Легендарный меч короля
Артура")
```

Методы, применяемые с классами, бывают трех типов:

1. **Обычные методы** (instance methods) — они работают с объектом класса и его данными. Это те методы, которые мы разобрали выше.
2. **Методы класса** (@classmethod) работают с самим классом, а не с его объектами. Они принимают в качестве первого аргумента *cls*, который является ссылкой на текущий класс. Аналогично тому, как *self* ссылается на экземпляр (объект) класса.

   Чтобы сделать метод методом класса, используется декоратор @classmethod.

Листинг 13.10

```
class Characters: # Класс персонажей
    # Атрибут класса
    group = "Спутники"

    @classmethod
    def show_group(cls):
        print("Это метод класса")
        print(f"Данные персонажи относятся к группе: {cls.group}")

Characters.show_group()
# Это метод класса
# Данные персонажи относятся к группе: Спутники
```

Здесь ***show_group*** — это метод класса. Вместо *self* он использует *cls*, чтобы обратиться к атрибуту класса ***group***. Аргумент *cls* внутри метода ссылается на сам класс, позволяя обращаться к его атрибутам и методам. Методы класса не могут напрямую обращаться к атрибутам экземпляров (предметов), так как они не связаны с конкретным объектом.

3. **Статические методы** (static methods) — это функции внутри класса, которые не зависят ни от объекта (self), ни от самого класса (cls). Они используются для логически связанных операций, но не изменяют состояние объекта или класса.

Чтобы создать статический метод, используется декоратор @staticmethod.

Листинг 13.11

```
class Characters:
    @staticmethod
    def joining():
        print("Персонаж следует за героем")

Characters.joining() # Персонаж следует за героем
```

Рассмотрим общий пример, с использованием всех типов вместе. Допустим, мы разрабатываем класс для управления книгами в библиотеке:

Листинг 13.12

```
class Library:
    total_books = 0 # Атрибут класса

    def __init__(self, title):
        self.title = title # Атрибут объекта
        Library.total_books += 1

    @classmethod
    def show_total_books(cls):
        print(f"Всего книг в библиотеке: {cls.total_books}")

    @staticmethod
    def library_rules():
        print("Правила библиотеки: с книгами обращаться аккуратно!")

# Создаем объекты — книги в библиотеке
book1 = Library("Гарри Поттер")
book2 = Library("Властелин колец")
book3 = Library("Песнь льда и пламени")

# Вызов метода класса
Library.show_total_books() # Всего книг в библиотеке: 3
# Вызов статического метода
Library.library_rules() # Правила библиотеки: с книгами обращаться
аккуратно!
```

В строке *Library.total_books* при создании нового предмета атрибут класса увеличивается на единицу, благодаря чему происходит автоматический подсчет всех созданных объектов. То есть независимо от того, сколько книг будет создано, мы всегда сможем узнать общее количество, вызвав метод класса *Library.show_total_books*.

13.4. Инкапсуляция

Инкапсуляция — это механизм, который позволяет объединить данные (атрибуты) и методы для их обработки в единое целое — объект. При этом внутренняя реализация объекта скрывается, предоставляя доступ к данным только через специально определенные интерфейсы (методы).

Для примера: вы купили какую-то технику и вас интересует, какие задачи она может выполнять, но вам не интересно, какие именно вычислительные процессы происходят внутри устройства. Вы нажимаете кнопки, получаете результат, и этого достаточно. В программировании этот принцип и называется инкапсуляцией.

Рассмотрим на примере утюга, который работает по определенному сценарию:

1. После нажатия кнопки происходит подача тока на нагревательный элемент.
2. Загорается лампочка, оповещающая о нагреве.
3. По достижении определенной температуры подача тока прекращается, а лампочка потухает.

То есть пользователю доступно только одно действие — нажатие кнопки, а все остальные события происходят без его участия, и ему не доступны прямые манипуляции с ними. Он не может самостоятельно по отдельности включать/выключать лампочку и нагревательный элемент. Эти возможности от него скрыты (инкапсулированы).

В коде это может выглядеть так:

Листинг 13.13

```
class Flatiron:
    def turn_on(self):
        print("Кнопка включена")
        self.power_supply_on()
        self.lamp_on()
        self.checking_temperature()
        self.turn_off()

    def power_supply_on(self):
        print("Подача тока")

    def lamp_on(self):
        print("Загорелась лампочка")

    def checking_temperature(self):
        print("Проверка температуры")

    def turn_off(self):
        print("Выключение")

# После нажатия кнопки выполняются события класса Flatiron
user_action = Flatiron()
user_action.turn_on()
```

В примере выше после нажатия кнопки пользователем происходит ряд запланированных событий, которые теоретически можно вызвать по отдельности, но у пользователя только одна кнопка, и он не может этого сделать.

Листинг 13.14

```
# Вызов действий по отдельности
user_action.lamp_on()
user_action.checking_temperature()
```

Если пользователь вскроет корпус утюга и подаст ток на лампочку или проверяющий элемент — эти функции сработают по отдельности. Чтобы этого избежать, функции инкапсулируются. Таким образом, даже при взаимодействии с элементами напрямую, они не будут работать по отдельности. Благодаря чему устройство будет безопасно для нерадивого пользователя.

Инкапсуляция в коде определяется одинарным и двойным нижним подчеркиванием:

Листинг 13.15

```
class Flatiron:
    def turn_on(self):
        print("Кнопка включена")
        self.__power_supply_on()
        self.__lamp_on()
        self.__checking_temperature()
        self.__turn_off()

    def __power_supply_on(self):
        print("Подача тока")

    def __lamp_on(self):
        print("Загорелась лампочка")

    def __checking_temperature(self):
        print("Проверка температуры")

    def __turn_off(self):
        print("Выключение")

# После нажатия кнопки выполняются события класса Flatiron
user_action = Flatiron()

# Вызов действий по отдельности приведет к ошибке выполнения
user_action.__lamp_on()
user_action.__checking_temperature()
```

Помимо безопасности пользователя, это также гарантирует защиту от случайного изменения класса программистом, если в результате каких-то вычислений один из параметров может быть изменен.

Однако стоит иметь в виду, что подобная защита в Python является условной, так как все равно остается доступ через служебные функции. Инкапсуляция несет рекомендательный характер, сообщая другим программистам, работающим над кодом, что функции, помеченные двойным подчеркиванием, должны оставаться неизменными (приватными).

Вместе с тем может возникнуть ситуация, когда пользователь решил выключить утюг, не дожидаясь нагрева (например, опаздывая на работу). Он

отключает утюг той же кнопкой. В таком случае происходит прерывание событий, когда программа завершается не по своему сценарию — последней **командой выключения** (turn_off), а в любой промежуточный момент из-за отсутствия тока.

Такой метод прерывания можно считать условно допустимым, то есть такая возможность есть, но лучше всего ей не пользоваться. Для этого предусмотрен "средний" уровень приватности, который помечается одним подчеркиванием. Тогда к методу можно получить доступ извне. Но подчеркивание все же намекает, что изменения также нежелательны.

Листинг 13.16

```
class Flatiron:
    def turn_on(self):
        print("Кнопка включена")
        self.__power_supply_on()
        self.__lamp_on()
        self.__checking_temperature()
        self._turn_off() # Одно подчеркивание

  # Здесь промежуточные функции
    def _turn_off(self): # Одно подчеркивание
        print("Выключение")

user_action = Flatiron()
user_action._turn_off() # Одно подчеркивание
```

В итоге функция _turn_off доступна для работы в отрыве от остального сценария, но разработчик будет иметь в виду, что изменение этой функции критично.

13.5. Геттеры и сеттеры

В программировании широко распространены три уровня доступа: *public*, *protected* и *private*.

public — свободный доступ;

protected — защищенный (одно подчеркивание);

private — приватный (два подчеркивания).

Бывают случаи, когда пользователю нужно дать доступ к приватным данным, но с ограниченными возможностями. Например, у него есть банковская ячейка. Он может узнать, что в ней лежит (получить значение), и положить туда что-то новое (изменить значение). Но доступ к аккаунту строго контролируется, в него можно попасть только с определенным ключом доступа (паролем), и класть в него можно только определенные вещи (только рубли, или только евро, или только доллары).

Ключи и правила доступа называются **геттерами** и **сеттерами**.

Геттер — это метод, который позволяет получить значение атрибута. Он как бы "открывает ящик" и показывает, что внутри.

Сеттер — позволяет изменить значение атрибута. "Закрывает ящик" с новым содержимым.

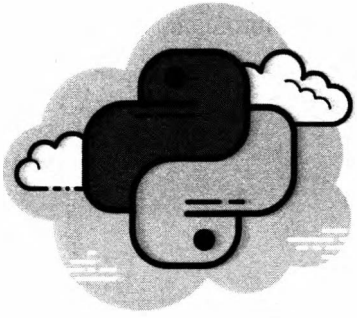
Листинг 13.17

```
class Bank_account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance # Приватный атрибут

    # Геттер для получения баланса
    def get_balance(self):
        return self.__balance

    # Сеттер для изменения баланса
    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Сумма должна быть положительной!")

account = Bank_account("Иван", 100) # Аккаунт Ивана со счетом 100
print(account.get_balance()) # Проверили баланс: 100
account.set_balance(250) # Изменили баланс
print(account.get_balance()) # Результат: 250
```

В этом примере у пользователя есть доступ к проверке баланса и изменению его через соответствующие методы, по правилам банка. Но у него нет прямого доступа к балансу (account.__balance). При попытке обратиться к нему будет получена ошибка.

13.5.1. Практические задания

1. Создайте класс для игровых персонажей, который будет иметь следующие свойства: имя персонажа, возраст и его профессия. Реализуйте метод ***get_info***, который будет возвращать строку с информацией о персонажах. Затем создайте несколько героев, и выведите информацию о них на экран.
2. Создайте класс для прямоугольника, у него должно быть два свойства: ширина и высота. Реализуйте методы, которые будут вычислять его площадь и периметр. Затем создайте два объекта-прямоугольника с разными параметрами и выведите их площадь и периметр на экран.



Глава 14.

Декораторы

14.1. Вложенные функции и замыкания

Вложенные функции — это функции, которые определены внутри другой функции. Они работают так же, как обычные функции, но находятся в локальной области видимости функции-оболочки.

Листинг 14.1

```
def obolochka(): # Определили внешнюю функцию
    print("Работает внешняя функция")
    # Здесь могут быть какие-то инструкции, например x + y

    def inner_func(): # Определили вложенную функцию
        print("Работает вложенная функция")
        # Здесь еще какие-то инструкции

    inner_func() # Внешняя вызывает вложенную функцию

# Вызвали функцию-оболочку
obolochka()

# Результат:
# Работает внешняя функция
# Работает вложенная функция
```

Вложенные функции помогают логически структурировать код. Если какие-то инструкции используются только внутри одной функции, удобно изолировать их в виде вложенной функции. Вместе с тем это применяется при инкапсуляции. Вложенные функции не видны за пределами своей внешней функции, что предотвращает их случайный вызов из другой части программы. То есть, если мы напрямую вызовем функцию inner_func(), то получим ошибку.

Кроме того, одной из ключевых причин использования вложенных функций является создание замыканий.

Замыкание — это функция, которая помнит значения переменных из своей внешней области видимости (оболочки), даже после того, как внешняя функция завершила свою работу.

Листинг 14.2

```
def obolochka(msg): # Внешняя функция создает замыкание
    def inner_func():
        # Вложенная функция, захватывает значение msg
        print(f"Сообщение: {msg}")
    return inner_func # Возвращаем вложенную функцию

# Сохраняем замыкание в переменную
data = obolochka("Привет, как дела?")
# Здесь внешняя функция завершается, но замыкание продолжает
существовать
# Значение "Привет, как дела?" сохраняется в замыкании
data() # Вызываем сохраненное замыкание

# Результат:
# Сообщение: Привет, как дела?
```

Когда внешняя функция *obolochka* завершает свою работу, локальные переменные (msg) удаляются из ее стека. Однако вложенная функция *inner_func* захватывает значение переменной *msg* и создает замыкание. Это означает, что даже после того, как *obolochka* завершила работу, значение *msg* остается доступным для *inner_func*. Переменные, которые вложенная функция "запоминает", называются **связанными переменными**.

Замыкания используются для сохранения состояния, они позволяют сохранять данные между вызовами функций. Благодаря этому можно создавать функции с разным поведением, в зависимости от входных данных. Кроме того, это позволяет реализовывать декораторы, о которых речь пойдет немного позже.

Чтобы замыкание работало корректно, следует придерживаться двух правил:

1. Внешняя функция должна возвращать вложенную функцию.

2. Вложенная функция должна использовать переменные из внешней функции.

У каждой функции в Пайтон есть специальное свойство __closure__. Оно возвращает информацию о связанных переменных. Это позволяет проверить, есть ли у функции замыкание.

Листинг 14.3

```
def obolochka(msg):
    def inner_func():
        print(f"Сообщение: {msg}")
    return inner_func

data = obolochka("Замыкание!")
print(data.__closure__) # Проверка замыкания

# Результат:
# (<cell at 0x000001F32F1237F0: str object at 0x000001F32EDAE550>,)
```

Это показывает, что *inner_func* имеет связанный объект (в данном случае строку *msg — str object*).

Для примера рассмотрим простую функцию, которая возвращает другую функцию, добавляющую заданное число к входному аргументу.

Листинг 14.4

```
def obolochka(x):
    def inner_func(y):
        return y + x
    return inner_func

data = obolochka(25) # Передали 25 во внешнюю функцию
print(data(5)) # 30
```

Сначала мы вызвали функцию *obolochka* и передали значение 25. Внутри функции создается новая функция *inner_func*, которая "запоминает" значение x, равное 25. Возвращенная функция *inner_func* присваивается переменной ***data***.

При отображении переменной (print(data)) та хранит функцию *inner_func* с сохраненным значением 25. Передавая ей новое значение 5, выполняется вычисление, результат которого выводится на экран.

Если передать новое значение (print(data(10))), оно будет сложено с сохраненным числом 25, а ответом будет 35. То есть внешняя функция забывает числа после завершения работы, а внутренняя запоминает (кэширует) первое состояние.

Листинг 14.5

```
print(data(5)) # 30
print(data(10)) # 35
```

14.2. Понятие декораторов

Декоратор — это функция, которая оборачивает другую функцию и добавляет к ней дополнительные возможности, не изменяя при этом исходный код этой функции.

Например, у нас есть подарок, который выполняет определенные функции. Мы хотим завернуть его в красивую упаковку (декорировать). Эту самую упаковку можно сравнить с декоратором в программировании.

Упаковка выполняет новые функции (защищает подарок от повреждения и придает красивый внешний вид), но не влияет на функциональность самого подарка. Декоратор (упаковка) принимает другую функцию (подарок) в качестве аргумента и возвращает новую функцию, которая расширяет поведение исходной функции.

Пример простой функции:

Листинг 14.6

```
def podarok():
    print("Шуруповерт закручивает саморезы")
```

Теперь обернем эту функцию в другую функцию, которая будет записывать в лог, что делает вызванная функция:

Листинг 14.7

```
def upakovka(func): # Декоратор, принимающий функцию в качестве
аргумента
    def logger(): # Внутренняя функция, создающая логи
        # Вывод имени декорируемой функции
        print(f"Вызвана функция {func.__name__}")
        func() # Вызов декорируемой функции (подарка)
        print(f"Функция {func.__name__} завершила выполнение")
    # Возвращаем новую функцию, которая будет вызвана вместо исходной
    return logger

decorator = upakovka(podarok) # Сохраняем декорированную функцию в
переменной
decorator() # Вызываем декорированную функцию

# Результат:
# Вызвана функция podarok
# Шуруповерт закручивает саморезы
# Функция podarok завершила выполнение
```

Функция **podarok** представляет собой основную функциональность, которую мы хотим "упаковать" в дополнительные действия. В данном случае это симуляция работы шуруповерта.

Функция **upakovka** является декоратором. Она принимает функцию **podarok** в качестве аргумента и возвращает новую функцию **logger**, которая выполняет дополнительные действия до и после вызова исходной функции. В данном случае декоратор добавляет логирование о начале и конце выполнения задачи.

Внутренняя функция **logger** отвечает за создание логов. Она выводит сообщение о начале выполнения, затем вызывает исходную функцию и, наконец, выводит сообщение об окончании выполнения.

В строке *decorator = upakovka(podarok)* мы применяем декоратор *upakovka* к функции *podarok*. Результатом является новая функция **decorator**, которая при вызове будет выполнять логирование и вызывать исходную функцию.

Такой подход полезен потому, что декораторы позволяют разделить основную логику функции от дополнительной функциональности, такой как логирование, профилирование, кэширование и т.д. Кроме того, один и тот же декоратор можно применять ко множеству разных функций. Например, с помощью вышеописанного декоратора мы можем отслеживать работу других функций *decorator = upakovka(другая_функция)*.

Листинг 14.8

```
def другая_функция():
    print("Работает другая_функция")
decorator = upakovka(другая_функция)
decorator()
```

В Python есть более удобный синтаксис для применения декораторов: @upakovka. Он эквивалентен коду *decorator = upakovka(podarok)*.

Листинг 14.9

```
def upakovka(func): # Создали декоратор
    def logger():
        print(f"Вызвана функция {func.__name__}")
        func()
        print(f"Функция {func.__name__} завершила выполнение")
    return logger

@upakovka # Применяем декоратор к функции
def podarok():
    print("Работает функция шуруповерт")

podarok() # Вызвали функцию с декоратором
```

Теперь, когда мы вызываем podarok(), Пайтон автоматически применяет декоратор @upakovka.

Мы можем применять сразу несколько декораторов к одной функции. Они будут выполняться в порядке сверху вниз:

Листинг 14.10

```
@timer
@upakovka
def podarok():
    print("Работает функция шуруповерт")

podarok() # Вызвали функцию с двумя декораторами
```

14.3. Декораторы с аргументами

Декораторы с аргументами расширяют возможности стандартных декораторов, позволяя передавать дополнительные параметры для настройки их поведения. Это делает декораторы более гибкими и адаптируемыми к различным задачам.

Листинг 14.11

```
def log_args(func):
    def logger(*args, **kwargs):
        print(f"Вызвана функция {func.__name__} с аргументами {args}
и именованными аргументами {kwargs}")
        return func(*args, **kwargs)
    return logger

@log_args
def summa(a, b):
    return a + b

result = summa(5, 7)
print(f"Результат: {result}")

# Вызвана функция summa с аргументами (5, 7) и именованными
аргументами {}
# Результат: 12
```

В примере выше внешняя функция **log_args** принимает функцию (summa) в качестве аргумента и возвращает другую функцию **logger**. Внутренняя функция **logger** выводит информацию о вызове функции, включая ее имя и переданные аргументы. Затем возвращает результат исходной функции.

14.4. Декораторы классов

Декораторы классов — это специальные функции, которые принимают на вход класс, изменяют или дополняют его поведение и возвращают модифицированный класс. Декораторы классов позволяют добавлять функциональность ко всему классу сразу, а не только к отдельным методам (функциям).

Листинг 14.12

```
def decorator_for_class(cls): # Декоратор для классов
    # Модифицируем класс
    cls.new_attribute = "Это новый атрибут!"
    return cls

@decorator_for_class
class Character: # Класс персонажей с декоратором
    pass

# Обращаемся к новому атрибуту класса Character
print(Character.new_attribute) # "Это новый атрибут!"
```

В примере мы создали декоратор **decorator_for_class** для применения его к одному или нескольким классам. Он добавляет новый атрибут, которого по умолчанию в созданных классах нет. Это удобно, когда мы работаем с большой программой и, чтобы добавить в нее новый функционал, создаем декоратор, оборачивающий классы, вместо того чтобы редактировать каждый класс по отдельности. Повторяя при этом один и тот же код в каждом классе.

Далее мы видим пустой класс Character, который по умолчанию ничего не делает, и, если обратиться к нему, ничего не произойдет. Но при добавлении к нему декоратора @decorator_for_class класс получает новые свойства.

Давайте рассмотрим более сложный пример. У нас есть несколько классов для персонажей: "Спутники", "Торговцы", "Ремесленники", каждый из них может обладать схожими или разными атрибутами.

Листинг 14.13

```
class Companions(): # Спутники
    def __init__(self, name, strength, agility):
        self.name = name
        self.strength = strength
        self.agility = agility

class Merchants(): # Торговцы
    def __init__(self, name, inventory, money):
        self.name = name
        self.inventory = inventory
        self.money = money

class Artisans(): # Ремесленники
    def __init__(self, name, inventory, money, skill):
        self.name = name
        self.inventory = inventory
        self.money = money
        self.skill = skill

# Создаем персонажей разных классов
Ulric = Companions("Ульрик", 20, 30)
Lukan = Merchants("Лукан", 55, 900)
Alvor = Artisans("Алвор", 15, 60, "Кузнец")
```

В какой-то момент разработки нам понадобится добавить новый атрибут ко всем или нескольким классам. Первое действие, которое напрашивается, — это дописать в каждый класс и каждому персонажу новый атрибут. Но что если у нас десятки классов и сотни персонажей? С помощью одного декоратора мы можем добавить новый атрибут ко всем классам сразу или только к нескольким необходимым.

Листинг 14.14

```
# Декоратор для добавления атрибута intellect
def add_intellect(cls):
    # Словарь со значениями интеллекта по умолчанию для разных
персонажей
    default_intellect = {
        "Ульрик": 15,
        "Лукан": 20,
        "Алвор": 10,
```

```
    }
    # Добавляем новый атрибут intellect к каждому экземпляру класса
    original_init = cls.__init__ # Сохраняем оригинальный __init__

    def new_init(self, name, *args, **kwargs):
        # Вызываем оригинальный __init__
        original_init(self, name, *args, **kwargs)
        # Устанавливаем интеллект в зависимости от имени
        # По умолчанию 10, если нет в словаре
        self.intellect = default_intellect.get(name, 10)

    cls.__init__ = new_init # Переназначаем __init__
    return cls
```

Если не использовать словарь, всем персонажам можно раздать одинаковое значение по умолчанию.

Далее нам остается только применить декоратор к нужным классам:

Листинг 14.15

```
# Применяем декоратор к классам
@add_intellect
class Companions(): # Спутники
    # Здесь остальной код класса

@add_intellect
class Merchants(): # Торговцы
    # Здесь остальной код класса

@add_intellect
class Artisans(): # Ремесленники
    # Здесь остальной код класса

# Проверяем атрибуты персонажей
print(f"Интеллект Ульрика: {Ulric.intellect}") # 15
print(f"Интеллект Лукана: {Lukan.intellect}") # 20
print(f"Интеллект Алвора: {Alvor.intellect}") # 10

# Изменяем значение интеллекта на новое
Ulric.intellect += 15
# И снова проверяем
print(f"Интеллект Ульрика: {Ulric.intellect}") # 30
```

14.5. Декоратор *@property*

Декоратор @property — это специальный декоратор, который превращает метод класса в атрибут. То есть мы можем вызвать метод, не используя круглые скобки, а так, будто обращаемся к обычному свойству объекта.

Это может понадобиться при инкапсуляции. Декоратор @property позволяет скрыть внутреннюю логику и работать с данными через интерфейс как с обычным атрибутом. Можно управлять доступом к атрибуту (например, проверять условия при его чтении или изменении).

Когда мы используем @property, создаются три связанных метода:

1. **Геттер** — отвечает за получение значений свойства.
2. **Сеттер** — отвечает за изменение значений свойства.
3. **Делетер** — отвечает за удаление значений свойства.

Пример создания геттера:

Листинг 14.16

```
class Circle:
    def __init__(self, radius):
        self._radius = radius # Приватный атрибут (начинается с "_")

    @property
    def radius(self):
        #Геттер возвращает значение радиуса
        return self._radius

# Передали значение
circle = Circle(12)
# Посмотрели значение
print(circle.radius) # Результат: 12
```

В примере у нас есть приватный атрибут *_radius*. Метод *radius* помечен как @property, и теперь мы можем обращаться к нему как к атрибуту, через точечную нотацию *circle.radius*.

Теперь давайте сделаем так, чтобы радиус можно было изменять. Добавим сеттер:

Листинг 14.17

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        #Геттер возвращает значение радиуса
        return self._radius

    @radius.setter
    def radius(self, value):
        # Сеттер устанавливает новое значение, но только положительное
        if value > 0: # Если введенное значение больше 0
            self._radius = value
        else:
            raise ValueError("Радиус должен быть положительным числом")

circle = Circle(12)
circle.radius = 15 # Устанавливаем новое значение радиуса
print(circle.radius) # Результат: 15
# circle.radius = -5 # Ошибка при вводе отрицательных значений
```

Метод @radius.setter позволяет задавать новое значение радиуса. Мы также добавили проверку, чтобы убедиться, что введено положительное значение. Если условие не выполнено, выводится ошибка (raise).

В некоторых случаях нужно полностью удалить значение свойства. Добавим делетер:

Листинг 14.18

```
    @radius.deleter
    def radius(self):
        # Делетер удаляет значение радиуса
        print("Радиус удален")
        del self._radius

circle = Circle(15) # Добавили значение
del circle.radius # Удалили значение
```

Декоратор @property позволяет добавить дополнительную логику для проверки значений перед их установкой. Это также полезно при инкапсуляции, мы можем скрыть внутреннюю реализацию свойства от пользователя.

В качестве еще одного примера рассмотрим расчет площади круга на лету, при получении новых значений:

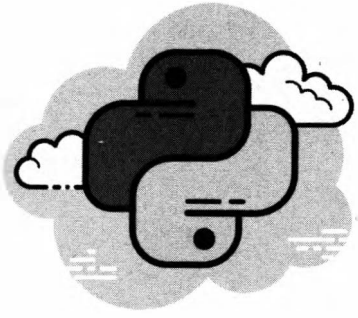
Листинг 14.19

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        # Возвращает радиус
        return self._radius

    @radius.setter
    def radius(self, value):
        # Устанавливает радиус
        if value > 0:
            self._radius = value
        else:
            raise ValueError("Радиус должен быть положительным числом")

    @property
    def area(self):
        # Вычисляет площадь круга
        return 3.14159 * self._radius ** 2

# Передали значение
circle = Circle(7)
# Получили рассчитанную площадь
print(circle.area) # 153.93791

circle.radius = 12 # Передали новое значение
print(circle.area) # 452.38896
```

Как видим, *area* становится свойством, которое всегда актуально, так как зависит от текущего значения *radius*.

14.5.1. Практические задания

1. Создайте декоратор, который ведет лог вызовов функции. Лог должен содержать имя функции, переданные аргументы и возвращаемое значение.
2. Создайте декоратор, который проверяет, имеет ли пользователь права доступа к выполнению функции. Если прав недостаточно, функция не запускается, а выводится сообщение: "Доступ запрещен".



Глава 15.

Специальные методы

15.1. Что такое специальные методы

Специальные методы, также называемые **магическими методами**, это методы в Python, которые имеют двойное подчеркивание в начале и конце своего имени (например, __init__, __str__). Они вызываются автоматически в определенных ситуациях и позволяют переопределить стандартное поведение объектов.

Магические методы позволяют создавать собственные классы, которые ведут себя так же, как встроенные типы данных. Это полезно для кастомизации поведения объектов. Мы можем сделать, чтобы наши объекты поддерживали операции сложения, сравнения, итерации и многое другое.

Ранее мы уже сталкивались с некоторыми из них.

Например, метод __init__ вызывается автоматически при создании объекта и отвечает за инициализацию. Он, как правило, принимает какие-то аргументы и сохраняет их в экземпляр класса.

Листинг 15.1

```
class Chracter():
    def __init__(self, name, skill):
        self.name = name
        self.skill = skill

# Создали один экземпляр Ulrik класса Chracter
Ulrik = Chracter("Ульрик", "Лучник")
```

В примере имя "Ульрик" и навык "Лучник" были присвоены конкретному объекту Ulrik. Когда мы создадим другого персонажа, метод __init__ присвоит ему свои атрибуты (имя и навык). В Пайтоне представлено довольно большое количество специальных методов, которые выполняют определенные функции. С большинством из них разберемся в соответствующих разделах этой главы.

15.2. Методы для арифметических операций

Арифметические специальные методы, как понятно из названия, отвечают за вычисления. Эти методы позволяют переопределить или определить поведение операторов, таких как +, -, *, /, для наших собственных классов.

Например, мы можем использовать + для сложения чисел. Однако, если нам понадобиться использовать его для объединения объектов класса, мы можем сделать это с помощью специального метода.

Листинг 15.2

```
class Coordinates:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, another):
        return Coordinates(self.x + another.x, self.y +
another.y)

    def __sub__(self, other):
        return Coordinates(self.x - other.x, self.y - other.y)

    def __repr__(self):
        return f"Новые координаты:({self.x}, {self.y})"

# Создаем две точки
point1 = Coordinates(10, 10)
point2 = Coordinates(5, 5)
```

```
# Складываем и вычитаем два объекта
print(point1 + point2) # Новые координаты: (15, 15)
print(point1 - point2) # Новые координаты: (5, 5)
```

Метод **__add__** определяет поведение оператора +. В нашем примере мы указали, что при сложении двух объектов их координаты должны складываться.

Метод **__sub__** делает противоположную операцию, вычитая координаты второй точки из координат первой.

Метод **__repr__** нужен для того, чтобы вывод объектов был понятным.

Без него мы бы увидели что-то вроде <__main__.Coordinates object at 0x00000159007FD970> (координата объекта хранится в такой-то ячейке памяти). В таком виде это не очень понятно и неудобно для работы с объектами.

Список специальных методов для арифметических операций:

__add__ Сложение (+)

__sub__ Вычитание (-)

__mul__ Умножение (*)

__truediv__ Деление (/)

__floordiv__ Целочисленное деление (//)

__mod__ Остаток от деления (%)

__pow__ Возведение в степень (**)

__matmul__ Матричное умножение (@)

__iadd__ Добавление с присваиванием (+=)

__isub__ Вычитание с присваиванием (-=)

15.3. Методы для создания и инициализации объектов

Методы инициализации позволяют контролировать процесс создания и инициализации объектов, чтобы они подходили под задачи нашей программы.

Когда мы вызываем класс, Пайтон создает новый объект этого класса. Затем его нужно подготовить к использованию, заполнить начальными данными, задать начальное состояние.

Метод **__new__** отвечает за создание нового экземпляра класса. Он вызывается первым и возвращает объект, который затем передается методу __init__.

Обычно нет необходимости переопределять этот метод, если мы не работаем с метаклассами или сложной логикой создания объектов.

Листинг 15.3

```
class Character:
    def __new__(cls, *args, **kwargs):
        print("Создаем объект")
        return super().__new__(cls)

    def __init__(self, name):
        print("Инициализируем объект")
        self.name = name

Ulrik = Character("Ульрик")
# Результат:
# Создаем объект
# Инициализируем объект
```

Когда мы пишем Ulrik = Character("Ульрик"), Пайтон вызывает метод __new__, который создает пустой объект. Затем этот объект передается в метод __init__, который наполняет его данными.

Метод **__init__** отвечает за инициализацию объекта, то есть за задание начальных значений атрибутов.

В типовых ситуациях нам нужно переопределять только __init__.

А __new__ переопределяют, если требуется тонкая настройка на этапе создания объекта (например, для реализации определенных шаблонов проектирования).

Метод **super()** — это встроенная функция в Python, которая позволяет вызывать методы из родительского (или базового) класса. Функция используется для обеспечения доступа к методам родителя без необходимости явно указывать его имя.

В примере super().__new__(cls) вызывает метод __new__ из базового класса **object** (все классы в Python неявно наследуются от **object**). Это возвращает новое, "сырое" (еще не инициализированное) место в памяти для будущего экземпляра класса. Таким образом, функция super() используется для вызова стандартного механизма создания объекта, который заложен в **object**.

15.4. Методы для представления объектов

Метод **__str__** определяет, как объект будет представлен в виде строки. Его задача создавать удобочитаемую строку, которая описывает объект.

Листинг 15.4

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def __str__(self):
        return f"Имя: {self.name}, Возраст: {self.age}"

# Создаем объект
Vakula = Person("Вакула", 25)
print(Vakula) # Результат: Имя: Вакула, Возраст: 25
```

Метод **__repr__** отвечает за "официальное" строковое представление объекта, предназначенное для разработчиков.

Это более формальный вид, который представляет информацию для разработки и отладки, ранее он уже был поверхностно описан.

Листинг 15.5

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Имя: {self.name}, Возраст: {self.age}"

# Создаем объект
Oxana = Person("Оксана", 20)
print(repr(Oxana)) # Результат: Имя: Оксана, Возраст: 20
```

Различия между __str__ и __repr__ в том, что первый определяется для пользователя, а второй для разработчика. Если мы определим только __repr__, но не определим __str__, то __repr__ будет использоваться вместо __str__.

Метод **__call__** позволяет объекту класса вести себя как функция. Если объект класса вызывается как функция с круглыми скобками, например obj(), Пайтон автоматически вызывает метод __call__.

Это полезно для реализации объектов-функций, а также для создания сложной логики при вызове объектов.

Листинг 15.6

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, x):
        return x * self.factor

# Создаем объект с фактором 3
triple = Multiplier(3)

# Используем объект как функцию
print(triple(10)) # Результат: 30 (10 x 3)
```

15.5. Методы для сравнения

Данные методы позволяют создавать объекты, которые могут сравниваться друг с другом с помощью таких операторов, как ==, <, >, !=, <=, >=.

Когда мы работаем с классами, интерпретатор не знает, как именно сравнивать экземпляры этих классов. Например, вы создаете класс для работы с координатами точки. Если попробовать сравнить две точки с помощью ==, Пайтон выдаст ошибку, потому что ему неясно, как определить, равны ли две точки.

Специальные методы сравнения решают эту проблему. Мы описываем логику сравнения объектов на основе их данных или других критериев. Это делает объекты класса более гибкими и удобными в использовании.

Основные специальные методы для сравнения:

__eq__(self, other) — проверяет равенство объектов (==).

__ne__(self, other) — проверяет неравенство (!=).

__lt__(self, other) — проверяет, меньше ли текущий объект (<).

__le__(self, other) — проверяет, меньше или равен текущий объект (<=).

__gt__(self, other) — проверяет, больше ли текущий объект (>).

__ge__(self, other) — проверяет, больше или равен текущий объект (>=).

Каждый из представленных методов принимает два аргумента:

self — текущий объект;

other — объект, с которым производится сравнение.

Методы возвращают True, если условие выполняется, или False, если нет.

Для примера создадим класс, который будет вычислять площади разных прямоугольников и сравнивать их между собой:

Листинг 15.7

```
class Rectangles:
    def __init__(self, width, height):
        # Создаем прямоугольник с длиной и шириной
        self.width = width
        self.height = height

    def area(self):
        # Вычисляем площадь прямоугольника
        return self.width * self.height

    def __eq__(self, other):
        # Сравниваем на равенство площадей
        return self.area() == other.area()

    def __lt__(self, other):
        # Проверяем, меньше ли текущий другого по площади
        return self.area() < other.area()

    def __le__(self, other):
        # Проверяем, меньше ли текущий или равен другому по площади
        return self.area() <= other.area()

# Создаем два прямоугольника
rect1 = Rectangles(5, 6) # Площадь 30
rect2 = Rectangles(3, 4) # Площадь 12

# Сравниваем их
print(rect1 == rect2) # False (площади не равны)
print(rect1 > rect2) # True (площадь rect1 больше)
print(rect1 <= rect2) # False (rect1 не меньше и не равен)
```

Методы сравнения могут понадобиться, если мы захотим отсортировать объекты класса. Это можно сделать с помощью функции sorted().

15.6. Методы для итерирования

Итерирование — это процесс последовательного перебора элементов в коллекции (итераций), этой темы мы уже касались ранее. Когда мы создаем цикл **for**, мы на самом деле выполняем итерирование.

Листинг 15.8

```
# Последовательно проходим по элементам списка
for item in [1, 2, 3]:
    print(item)
```

Но как интерпретатор "узнает", каким образом итерироваться по объекту? Это делается с помощью специальных методов итерирования. Чтобы объект мог быть использован в цикле for или функциях, таких как list() или sum(), он должен реализовать два метода.

Метод **__iter__(self)** возвращает итератор — объект, который знает, как обрабатывать итерирование. Итератор может быть самим объектом, если он реализует следующий метод.

Метод **__next__(self)** возвращает следующий элемент в последовательности. Когда элементы заканчиваются, он должен выбросить исключение StopIteration, чтобы Пайтон понял, что итерирование завершено.

Итак, итерируемый объект — это объект, у которого есть метод __iter__. Например: список, строка, словарь.

Итератор — это объект, у которого есть методы __iter__ и __next__.

Итератор "запоминает", где он остановился, и может выдать следующий элемент.

Когда мы создали цикл в примере выше, Пайтон вызывает метод __iter__ у списка, получая итератор. Затем вызывается метод __next__ у итератора, чтобы получить следующий элемент. Далее повторяется вызов метода __next__, пока не возникнет исключение *StopIteration*.

Для примера создадим класс, представляющий диапазон чисел (аналог встроенной функции ***range***):

Листинг 15.9

```
class Ranking:
    def __init__(self, start, end):
        self.start = start # Стартовая позиция
        self.end = end # Конечная позиция
        self.current = start # Текущая позиция для итерации

    def __iter__(self):
        # Метод возвращает сам объект как итератор
        return self

    def __next__(self):
        # Метод возвращает следующий элемент или выбрасывает
StopIteration
        if self.current < self.end: # Если текущая меньше конечной
            result = self.current # Записываем текущую позицию
            self.current += 1 # Прибавляем 1 к текущей позиции
            return result # Возвращаем текущую позицию
        else: # Иначе останавливаемся
            raise StopIteration

my_range = Ranking(1, 5)
for num in my_range:
    print(num)
# Результат: 1, 2, 3, 4
```

На этапе, когда текущая позиция получает значение 5, она становится не меньше конечной позиции в строке *if self.current < self.end (False)*, поэтому число 5 не выводится. Пайтон переходит в блок **else**, где создается исключение *StopIteration*.

Итераторы полезны, когда нужно работать с большим объемом данных. Они позволяют не загружать в память все сразу, а генерировать элементы "по запросу".

15.7. Методы для доступа к элементам

Методы для доступа к элементам это те самые методы, которые стоят за квадратными скобками [ ], используемыми для индексации, срезов, а также проверки на принадлежность элемента.

Доступ к элементам дает возможность извлекать или изменять данные в объекте, используя знакомый нам синтаксис:

spisok[0] — доступ к элементам списка по их индексу.

spisok[1:4] — доступ к срезу элементов в списке.

slovar['key'] — доступ к элементу словаря по его ключу.

'a' in spisok — проверка наличия элемента.

Чтобы Пайтон понимал, что нужно делать, когда мы используем такие конструкции, он вызывает специальные методы, которые можно настроить в своих классах.

Метод **__getitem__(self, key)** — отвечает за получение элемента. Он вызывается, когда используется синтаксис obj[key].

Листинг 15.10

```
class Spisok:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        return self.data[index]

# Используем
my_list = Spisok([10, 20, 30])
print(my_list[1]) # Результат: 20 (элемент с индексом 1 в списке)
```

Метод **__setitem__(self, key, value)** — вызывается, когда нужно изменить элемент: obj[key] = value.

Листинг 15.11

```
class Spisok:
    def __init__(self, data):
        self.data = data

    def __setitem__(self, index, value):
        self.data[index] = value

my_list = Spisok([10, 20, 30])
my_list[1] = 50 # Изменяем элемент с индексом 1
print(my_list.data) # Результат: [10, 50, 30]
```

Метод **__delitem__(self, key)** — используется для удаления элемента: del obj[key].

Листинг 15.12

```
class Spisok:
    def __init__(self, data):
        self.data = data

    def __delitem__(self, index):
        del self.data[index]

my_list = Spisok([10, 20, 30])
del my_list[1] # Удаляем элемент с индексом 1
print(my_list.data) # Результат: [10, 30]
```

Метод **__contains__(self, item)** — вызывается при использовании оператора **in**, чтобы проверить, содержится ли элемент в объекте.

Листинг 15.13

```
class Spisok:
    def __init__(self, data):
```

```
        self.data = data

    def __contains__(self, item):
        return item in self.data

my_list = Spisok([10, 20, 30])
print(20 in my_list) # True (элемент есть в списке)
print(50 in my_list) # False (элемента нет)
```

Метод **__missing__(self, key)** — используется для работы с отсутствующими элементами в словарях. Он вызывается, если ключ отсутствует.

Листинг 15.14

```
class Slovar(dict):
    def __missing__(self, key):
        return f"Ключ {key} отсутствует"

# Используем
my_dict = Slovar({'A': 1, 'B': 2})
print(my_dict['C']) # "Ключ C отсутствует"
```

Когда мы создаем свой класс и добавляем в него методы __getitem__, __setitem__ и другие, мы можем создавать объекты, которые будут вести себя как списки, словари или другие коллекции. Это удобно для создания собственных структур данных.

15.8. Методы для управления контекстом

Методы управления контекстом (контекстные менеджеры) используются для создания объектов, которые могут безопасно управлять ресурсами, например, файлами, соединениями с базами данных или сетевыми подключениями.

Управление контекстом позволяет гарантировать, что ресурс будет корректно освобожден или закрыт, даже если внутри блока кода произойдет ошибка. Это достигается с помощью конструкции **with**.

Листинг 15.15

```
with open('file.txt', 'r') as f:
    data = f.read()
# После выхода из блока файл автоматически закрывается
```

Из предыдущих разделов мы помним, что без контекстного менеджера файлы нужно закрывать вручную:

Листинг 15.16

```
f = open('file.txt', 'r')
try:
    data = f.read()
finally:
    f.close()
```

Если мы забудем закрыть файл или ресурс, это может привести к утечкам памяти, блокировкам или другим проблемам. Управление контекстом решает эту задачу автоматически.

Метод **__enter__(self)** — вызывается в начале блока **with**. Он выполняет инициализацию или подготовительные действия. Например, открывая файл или устанавливая соединение.

Листинг 15.17

```
def __enter__(self):
    print("Ресурс инициализирован")
    return self
```

Метод **__exit__(self, exc_type, exc_value, traceback)** — вызывается при выходе из блока **with**. Он выполняет завершающие действия, такие как закрытие файла или освобождение ресурса.

Метод принимает три параметра:

1. *exc_type* — тип исключения, если оно произошло;
2. *exc_value* — само исключение;
3. *traceback* — объект трассировки исключения.

Если исключение было обработано внутри __exit__, метод должен вернуть True, чтобы оно не распространялось.

Листинг 15.18

```
def __exit__(self, exc_type, exc_value, traceback):
    print("Ресурс освобожден")
    if exc_type:
        print(f"Исключение: {exc_value}")
    return True # Предотвращаем распространение исключения
```

Для примера создадим простой класс, который симулирует открытие и закрытие ресурса:

Листинг 15.19

```
class Resource:
    def __enter__(self):
        print("Ресурс открыт")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Ресурс закрыт")
        if exc_type:
            print(f"Обработано исключение: {exc_value}")
            return True # Предотвращаем распространение исключения

# Используем
with Resource() as r:
    print("Внутри блока with")
    raise ValueError("Произошла ошибка") # Исключение будет
обработано

# Результат:
# Ресурс открыт
# Внутри блока with
# Ресурс закрыт
# Обработано исключение: Произошла ошибка
```

Таким образом, благодаря методам __enter__ и __exit__ достигается автоматизация. Нам нет необходимости вручную закрывать или освобождать ресурсы. Исключения внутри блока не нарушают правильного освобождения ресурсов. Кроме того, код становится чище и проще для понимания.

15.9. Методы для преобразования типов

Данные методы позволяют контролировать, как объект нашего класса преобразуется в стандартные типы данных, такие как строки, числа, булевы значения и другие.

Метод **__int__(self)** — определяет, как объект преобразуется в целое число с помощью функции int().

Листинг 15.20

```
class Item:
    def __init__(self, price):
        self.price = price

    def __int__(self):
        return int(self.price)

item = Item(19.99)
print(int(item)) # Результат: 19 (преобразовали к целому числу)
```

Метод **__float__(self)** — позволяет преобразовать объект в число с плавающей точкой с помощью float().

Листинг 15.21

```
class Item:
    def __init__(self, price):
        self.price = price

    def __float__(self):
```

```
        return float(self.price)

item = Item(19)
print(float(item)) # Результат: 19.0
```

Метод **__bool__(self)** — определяет логическое значение объекта (истина или ложь) при вызове bool() или использовании объекта в логических выражениях.

Листинг 15.22

```
class Box:
    def __init__(self, items):
        self.items = items

    def __bool__(self):
        return bool(self.items) # True, если список не пуст

box = Box([])
print(bool(box)) # Результат: False
box.items.append("Игрушка") # Добавили предмет
print(bool(box)) # Результат: True
```

Метод **__bytes__(self)** — позволяет преобразовать объект в байтовое представление с помощью bytes().

Листинг 15.23

```
class Data:
    def __init__(self, content):
        self.content = content

    def __bytes__(self):
        return self.content.encode('utf-8')

data = Data("Привет")
print(bytes(data))
# Результат: b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
```

Методы **__str__(self)** и **__repr__(self)** преобразуют данные в тип "строка".

Они были рассмотрены в главе 13.

Рассмотрим пример класса с несколькими методами преобразования:

Листинг 15.24

```
class Product:
    def __init__(self, name, price, stock):
        self.name = name
        self.price = price
        self.stock = stock

    def __str__(self):
        return f"{self.name}: ${self.price} ({self.stock} в наличии)"

    def __repr__(self):
        return f"Product(name='{self.name}', price={self.price},
stock={self.stock})"

    def __int__(self):
        return int(self.stock)

    def __float__(self):
        return float(self.price)

    def __bool__(self):
        return self.stock > 0

# Тестируем
book_1 = Product("Книга", 14.99, 10)
print(book_1) # Книга: $14.99 (10 в наличии)
print(repr(book_1)) # Product(name='Книга', price=14.99, stock=10)
print(int(book_1)) # 10
print(float(book_1)) # 14.99
print(bool(book_1)) # True
```

15.10. Другие специальные методы

В Python есть группа вспомогательных методов, которые дают классам дополнительные возможности. Они позволяют управлять поведени-

ем объектов в нестандартных ситуациях, таких как использование в хеш-таблицах, проверки на наличие элементов или динамическая работа с атрибутами.

Метод **__hash__** — возвращает числовое значение (хеш) объекта. Это значение используется для работы с хеш-таблицами, такими как словари (dict) и множества (set).

Для примера у нас есть большая библиотека с большим количеством книг. Чтобы не перебирать каждую полку в поисках нужной книги, у каждой из них есть уникальный номер.

Листинг 15.25

```
class Books:
    def __init__(self, name):
        self.name = name

    def __hash__(self):
        return hash((self.name))

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name
        return False

book_1 = Books("Властелин колец") # Создали две книги
book_2 = Books("Гарри Поттер")
spisok = [book_1, book_2] # Добавили книги в список
print(book_1 in spisok) # True (книга есть в списке)
print(hash(book_1)) # Хеш-значения объектов: 6582370375084028807
print(hash(book_2)) # 278107612259741418
```

Метод **__getattr__** вызывается, когда Python не может найти запрашиваемый атрибут у объекта. С его помощью мы можем динамически обрабатывать такие обращения.

Листинг 15.26

```
class Dynamic_attr:
    def __init__(self):
        self.data = {"x": 42, "y": 99}

    def __getattr__(self, name):
        return self.data.get(name, f"Атрибут  {name} не найден")

obj = Dynamic_attr()
print(obj.x) # 42 (существует в словаре)
print(obj.z) # Атрибут z не найден(атрибута нет)
```

Метод **__setattr__** вызывается при установке атрибута у объекта. С его помощью контролируется процесс присвоения значений. Метод полезен, когда необходимо проверить или изменить значения перед их сохранением.

Листинг 15.27

```
class Character:
    def __init__(self, name, age):
        self.name = name # Это вызовет __setattr__
        self.age = age

    def __setattr__(self, name, value):
        if name == "age" and value < 0:
            raise ValueError("Возраст не может быть отрицательным")
        super().__setattr__(name, value) # Сохраняем значение

Vakula = Character("Вакула", 25)
print(Vakula.age) # 25
# Vakula.age = -5 # Ошибка: Возраст не может быть отрицательным
```

Метод **__delattr__** вызывается, когда мы удаляем атрибут объекта с помощью **del**.

Листинг 15.28

```
class Character:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __delattr__(self, name):
        print(f"Атрибут {name} удален")
        super().__delattr__(name) # Удаляем атрибут

Vakula = Character("Вакула", 25)
del Vakula.age # Атрибут age удален
```

Обратите внимание, что мы не можем создавать свои специальные методы. Они строго определены языком, их поведение связано с базовыми функциями и операциями, такими как арифметические, итерирование, сравнение и другие.

Однако, если нам нужно реализовать определенное поведение, мы всегда можем создать обычный метод или вспомогательные функции, а также использовать декораторы.

Если вы хотите, чтобы объект выполнял "особую" операцию, которую стандартные методы не охватывают, просто создайте метод с говорящим названием:

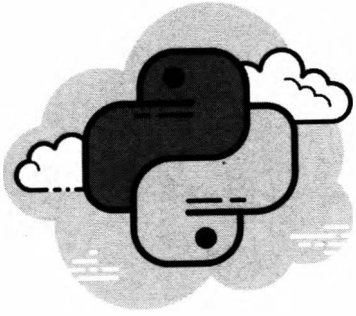
Листинг 15.29

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def scale(self, ratio):
        # Масштабируем вектор на заданный коэффициент
        self.x *= ratio
        self.y *= ratio
        return self

vec = Vector(2, 3)
vec.scale(2) # Масштабируем вектор на 2
print(vec.x, vec.y) # 4, 6
```

Таким образом мы добавили "особую" функциональность без необходимости придумывать новый магический метод.

15.10.1. Практические задания

1. Создайте класс, который подсчитывает количество созданных объектов, а также выводит это количество при каждом новом созданном объекте.
2. Создайте класс, который представляет собой коробку, содержащую несколько элементов. Реализуйте методы: для доступа к элементам по индексу, для изменения элементов по индексу, для получения количества элементов в коробке.



Глава 16.

Наследование и полиморфизм

16.1. Наследование

Наследование — это одна из ключевых концепций объектно-ориентированного программирования, она полезна, если мы хотим создавать гибкие и масштабируемые программы. Благодаря наследованию одни классы могут наследовать свойства и методы других классов.

Например, в нашем проекте имеются два класса персонажей: "Торговцы" и "Ремесленники":

Листинг 16.1

```
class Merchants:
    def can_trade(self):
        print("Я могу торговать")

class Artisans:
    def can_create(self):
        print("Я могу создавать")

# Создаем персонажей каждого класса
Lukan = Merchants()
Alvor = Artisans()

# Обращаемся к их методам
Lukan.can_trade() # Я могу торговать
Alvor.can_create() # Я могу создавать

# Обращаемся к методу чужого класса
Lukan.can_create() # Вызовет ошибку
```

Это два отдельных класса, не связанные друг с другом. Поэтому при попытке обратиться к методу чужого класса будет вызвана ошибка с соответствующим описанием.

Однако персонажи разных классов могут иметь общие свойства или методы. Например, каждый из них имеет навык владения мечом. Чтобы реализовать этот навык, каждому классу нужно прописать соответствующий атрибут.

Листинг 16.2

```
class Merchants:
    def can_trade(self):
        print("Я могу торговать")

    def sword_skill(self):
        print("Я могу сражаться")

class Artisans:
    def can_create(self):
        print("Я могу создавать")

    def sword_skill(self):
        print("Я могу сражаться")
```

Как видим, метод ***sword_skill*** у обоих классов идентичен, один и тот же код повторяется два раза. Представьте, что у нас есть десятки разных классов (Торговцы, Ремесленники, Барды, Спутники и т.д.), все они умеют владеть мечом. Кроме того, у них также могут быть десятки других одинаковых навыков, таких как владение щитом, езда верхом и т.д.

То есть один и тот же код с одними и теми же навыками может повторяться очень много раз. Это лишнее время при написании, лишний объем кода и неудобство при чтении и разборе чужого скрипта. А теперь представим, что навык владения мечом нужно изменить, добавив дополнительные возможности. Нам предстоит пройтись по всем этим классам и каждый отредактировать отдельно.

В таких случаях нам и понадобится наследование, где все общие навыки персонажей помещаются в один общий родительский класс. А затем каждый из подклассов наследует этот навык у родителя. Чтобы один класс наследовал свойства другого (родительского) класса, этот класс прописывается в скобках подкласса.

Листинг 16.3

```
class Character: # Родительский класс
    def sword_skill(self):
        print("Я могу сражаться")

class Merchants(Character): # Наследуем навык меча от родителя
    def can_trade(self):
        print("Я могу торговать")

class Artisans(Character): # Наследуем навык меча от родителя
    def can_create(self):
        print("Я могу создавать")

# Создали персонажей
Lukan = Merchants()
Alvor = Artisans()

# Обращаемся к методу владения мечом
Lukan.sword_skill() # Я могу сражаться
Alvor.sword_skill() # Я могу сражаться
```

Теперь, при попытке изменить общий навык для всех персонажей, нам потребуется отредактировать его в родительском классе. И все изменения отобразятся на всех подклассах, которые его наследуют.

У каждого подкласса могут быть свои подклассы. Например, торговцы могут подразделяться на торговцев оружием, лошадьми, едой, алхимическими ингредиентами. Все эти подклассы могут иметь свои уникальные свойства и методы, а также наследовать их от родителя. Кроме того, если родительский класс также имеет своего родителя, то и эти свойства будут переданы подклассу подкласса. Такое наследование называется **многоуровневым**.

Листинг 16.4

```
class Character:
    def sword_skill(self):
        print("Я могу сражаться")

class Merchants(Character): # Наследуем навык меча от Character
    def can_trade(self):
```

```
        print("Я могу торговать")

class Alchemist(Merchants): # Наследуем навык торговли и меча от
Merchants
    def sell_alchemy(self):
        print("Продаю алхимические ингредиенты")

# Персонаж класса алхимиков
Lukan = Alchemist()

# Обращаемся к родительским методам
Lukan.sword_skill() # Я могу сражаться
Lukan.can_trade() # Я могу торговать
Lukan.sell_alchemy() # Продаю алхимические ингредиенты
```

Вместе с тем один класс может наследоваться сразу от нескольких, не связанных между собой классов. Это называется **множественным наследованием**.

В таком случае в скобках класса прописываются через запятую несколько классов, от которых будут унаследованы свойства и методы.

Листинг 16.5

```
class Monsters:
    def can_bite(self):
        print("Я могу кусаться")

class Flyer:
    def fly(self):
        print("Я могу летать")

class Dragon(Monsters, Flyer): # Наследуем от двух отдельных классов
    def breath_fire(self):
        print("Дышу огнем")

# Персонаж класса драконов
Paarthurnax = Dragon()

# Наследует методы двух родителей
Paarthurnax.can_bite() # Я могу кусаться
Paarthurnax.fly() # Я могу летать
Paarthurnax.breath_fire() # Дышу огнем
```

Гибридное наследование — сочетает в себе несколько типов наследования. Это может быть комбинация многоуровневого и множественного наследования. Полезно, когда нужно объединить функциональность из разных классов, если объект должен обладать их общими свойствами.

16.2. Наследование от встроенных типов

Из предыдущих разделов мы помним, что встроенные типы данных в Python — это базовые кирпичики, которые мы используем для работы с данными. Эти типы предоставляют основные методы и операции для взаимодействия с ними (например, методы для списков, методы для строк и т.д.).

int (целые числа)

float (вещественные числа)

str (строки)

list (списки)

dict (словари)

set (множества)

Иногда встроенных возможностей этих типов недостаточно для решения конкретных задач. Например, нам может понадобиться, чтобы список автоматически сортировался при добавлении элемента. Или чтобы словарь игнорировал регистр при поиске ключей. Наследование от встроенных типов позволяет использовать их функциональность как основу, добавляя или изменяя ее под свои задачи.

Чтобы наследовать от встроенных типов, нужно создать свой класс, который станет дочерним от встроенного типа.

Листинг 16.6

```
class Spisok(list): # Наследуем возможности встроенного типа list
    pass
```

Теперь наш класс обладает всеми методами и поведением обычного списка, и мы можем добавлять к нему свои методы или переопределять существующие. Для примера создадим класс, который добавляет элемент в список только в том случае, если его там еще нет.

Листинг 16.7

```
class Unique_list(list): # Наследуем от стандартного списка
    def append(self, item): # Метод для добавления элемента
        if item not in self: # Если элемента нет в текущем списке
            super().append(item) # Используем стандартный метод append
        else: # Иначе выводим сообщение
            print(f"Элемент {item} уже есть в списке")

# Создали экземпляр класса, представляющий собой список
spisok = Unique_list(['апельсин', 'арбуз', 'дыня'])
spisok.append('банан') # Добавляем новый элемент
print(spisok) # Вывели список на экран
spisok.append('дыня') # Пробуем добавить существующий элемент
# Результат: Элемент "дыня" уже есть в списке
```

Таким образом, мы создали собственный метод для добавления элементов в список, который наследует стандартный функционал метода для списков (list.append()), но добавляет функцию проверки элементов на уникальность.

Вы можете заметить, что работу метода добавления мы не прописывали, а с помощью функции ***super*** обратились к родительскому методу, откуда и была взята возможность пополнения списка.

Рассмотрим еще один пример, в котором словарь будет игнорировать регистр букв в ключах. Мы помним, что похожие слова или переменные в разных регистрах являются отдельными объектами (например, slovar, Slovar или SLOVAR), но иногда это может быть неудобно при работе с некоторыми данными.

Листинг 16.8

```
class Slovar(dict): # Наследуем от стандартного словаря
    def __setitem__(self, key, value):
        # Сохраняем ключи в нижнем регистре методом lower
        super().__setitem__(key.lower(), value)

    def __getitem__(self, key):
```

```
        # Ищем ключи в нижнем регистре
        return super().__getitem__(key.lower())

    def __contains__(self, key):
        # Проверяем наличие ключей без учета регистра
        return super().__contains__(key.lower())

# Создаем объект класса Slovar
data = Slovar()
# Присваиваем ключу Name значение "Иван"
data["Name"] = "Иван"
# Проверяем значение по ключу в другом регистре
print(data["name"]) # Иван
# Снова обращаемся к ключу в другом регистре
print("NAME" in data) # True
```

В данном примере наш словарь также наследует методы стандартного словаря, и при обращении к прописанным в нем методам они функцией ***super*** возвращают методы из базового типа. Таким образом, мы можем расширять базовый функционал методов, добавляя необходимые нам возможности.

16.3. Делегирование

Делегирование — это процесс, при котором один объект передает выполнение некоторых своих операций другому объекту. Представьте, что вы начальник отдела, и вместо того, чтобы выполнять все задачи самому, распределяете их между своими сотрудниками.

Каждый из них отвечает за свою часть работы, а вы организуете этот процесс. Это и есть делегирование, только в программировании. Вместо того чтобы наследовать все свойства и методы базового класса, наш класс может "сотрудничать" с другими объектами.

Допустим, у нас есть класс, который отвечает за управление списком задач:

Листинг 16.9

```
class TaskList:
    def __init__(self):
        self.tasks = [] # Список задач

    def add_task(self, task):
        # Добавляет задачи в список
        self.tasks.append(task)

    def remove_task(self, task):
        # Удаляет задачи из списка
        self.tasks.remove(task)

    def show_tasks(self):
        # Отображает задачи на экране
        print("Текущие задачи:", self.tasks)
```

Теперь создадим другой класс Manager, который делегирует управление списком задач объекту TaskList.

Листинг 16.10

```
class Manager:
    def __init__(self):
        # Делегируем задачи объекту TaskList
        self.task_list = TaskList()

    def add_task(self, task):
        # Текущая задача отдается методу add_task класса TaskList
        self.task_list.add_task(task)

    def remove_task(self, task):
        # Текущая задача отдается методу remove_task класса TaskList
        self.task_list.remove_task(task)

    def show_tasks(self):
        # Текущая задача отдается методу remove_task класса TaskList
        self.task_list.show_tasks()

# Создали список для задач
manager = Manager()
# Добавили задачи в список
manager.add_task("Отправить документ")
manager.add_task("Написать отчет")
```

```
# Отобразили задачи
manager.show_tasks()
# Удалили задачу
manager.remove_task("Отправить документ")
manager.show_tasks()
```

В данном примере все задачи класса Manager были переданы на выполнение классу TaskList (одному сотруднику). Однако Manager каждую отдельную задачу может передавать нескольким разным классам (нескольким сотрудникам), каждый из которых может выполнять свои определенные функции.

Иногда нужно делегировать вызов всех методов, которых нет в вашем классе, другому объекту.

Для этого используется специальный метод __getattr__:

Листинг 16.11

```
class TaskList:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def show_tasks(self):
        print("Задачи:", self.tasks)

class Manager:
    def __init__(self):
        self.task_list = TaskList()

    def __getattr__(self, name):
        # Делегируем вызовы классу TaskList
        return getattr(self.task_list, name)

manager = Manager()
manager.add_task("Отправить документ")
manager.show_tasks()
```

В данном случае мы хотим делегировать все задачи именно одному сотруднику. Поэтому, чтобы не прописывать каждый метод, как в предыду-

щем примере, мы создаем один общий, который абсолютно все задачи будет перенаправлять классу TaskList.

16.4. Расширение и переопределение класса

Расширение класса — это процесс добавления нового функционала к уже существующему классу, без необходимости переписывать или изменять его изначальный код. Это делается путем создания нового класса, который наследует свойства и методы базового (родительского) класса, и добавления к нему новых возможностей.

Подобное расширение мы могли наблюдать в одной из предыдущих глав, когда два класса персонажей наследовали общий метод у родителя, но расширяли его своими уникальными методами.

Листинг 16.12

```
class Character: # Родительский класс
    def sword_skill(self):
        print("Я могу сражаться")

class Merchants(Character): # Наследуем от Character
    # и расширяем новой возможностью
    def can_trade(self):
        print("Я могу торговать")

class Artisans(Character): # Наследуем от Character
    # и расширяем новой возможностью
    def can_create(self):
        print("Я могу создавать")
```

То есть имея определенный класс (Character), мы можем расширить его, создав подкласс, который не влияет на базовый код, а дополняет его новым. Таким образом, мы не переписываем старую часть скрипта, от которой могут зависеть другие классы, которые могут быть сломаны, если новый функционал будет противоречить их логике.

В нашем случае базовый код остается неизменным, поддерживая работоспособность старых скриптов, но благодаря новому подклассу мы получили новые возможности для реализации определенных целей.

Переопределение — используется, когда нужно изменить поведение метода базового класса. Для этого в дочернем классе мы переопределяем этот метод.

Листинг 16.13

```
class Character: # Родительский класс
    def sword_skill(self):
        print("Я могу сражаться")

class Warrior(Character): # Наследуем от Character
    # Переопределили навык меча
    def sword_skill(self):
        print("Могу сражаться двумя мечами")

# Персонаж класса Warrior
Ulrik = Warrior()
# Персонаж класса Character
Alvor = Character()

# Обращаемся к одному и тому же методу, но в разных классах
Ulrik.sword_skill() # Могу сражаться двумя мечами
Alvor.sword_skill() # Я могу сражаться
```

То есть, мы переопределили метод ***sword_skill*** для одного подкласса, который работает в измененном варианте, но не влияет на работу того же метода в родительском классе. Это полезно, когда для определенной части программы нужно слегка изменить базовую логику скрипта, но при этом не нарушая работу других частей программы.

Иерархия наследования такова, что при обращении к методу ***sword_skill*** интерпретатор ищет его внутри текущего класса, и если не находит здесь, то поднимается на уровень выше — в родительский класс — и использует данный метод оттуда. Поэтому Ульрик использовал переопределенный метод, а Алвор — стандартный.

Также бывают случаи, когда вместо переопределенного метода необходимо использовать стандартный. В нашем случае Ульрику необходимо воспользоваться методом из родительского класса ("Я могу сражаться"). Здесь нам поможет функция super(), которая обращается к родительскому классу.

Листинг 16.14

```
class Character: # Родительский класс
    def sword_skill(self):
        print("Я могу сражаться")

class Warrior(Character): # Наследуем от Character
    # Переопределили навык меча
    def sword_skill(self):
        super().sword_skill() # Вызов метода из базового класса
        print("Могу сражаться двумя мечами")

# Персонаж класса Warrior
Ulrik = Warrior()

Ulrik.sword_skill()
# Я могу сражаться
# Могу сражаться двумя мечами
```

В данном примере при обращении к переопределенному методу Пайтон сначала видит строку с функцией super(), которая вызывает родительский метод, а после этого выполняет инструкции ниже, которые относятся непосредственно к дочернему методу. Не забываем, что интерпретатор читает код сверху вниз, поэтому порядок строк в коде важен, так как это может изменить логику программы.

16.5. Атрибут *__slots__*

Атрибут *slots* является специальным атрибутом класса, позволяющим ограничить набор атрибутов, которые могут быть добавлены к объекту.

Для примера, в обычном классе мы можем изменять и добавлять атрибуты как нам угодно:

Листинг 16.15

```
class Character:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Создаем персонажа
Ulrik = Character("Ульрик", 30)
# Изменяем атрибут age
Ulrik.age = 35
# Добавляем новый атрибут
Ulrik.skill = "Владение мечом"

#Проверяем
print(Ulrik.age) # 35
print(Ulrik.skill) # Владение мечом
```

При использовании атрибута *slots* мы можем запретить создание новых атрибутов, указав разрешенные:

Листинг 16.16

```
class Character:
    __slots__ = ('name', 'age') # Разрешенные атрибуты

    def __init__(self, name, age):
        self.name = name
        self.age = age

# Создаем персонажа
Ulrik = Character("Ульрик", 30)
# Изменяем атрибут age
Ulrik.age = 35
# Добавляем новый атрибут
Ulrik.skill = "Владение мечом" # Ошибка!
```

В итоге, если мы имеем механизм в коде, который добавляет и изменяет новые атрибуты, с помощью __slots__ мы можем ограничить это действие для определенных классов. Например, персонажи класса "Воин" могут получить новый атрибут "Владение мечом", а персонажам класса "Торговцы" можно установить ограничение.

Кроме того, __slots__ применяется для экономии памяти. В Пайтон каждый объект содержит словарь для хранения атрибутов. Даже если у нас всего 2 атрибута, сам объект словаря занимает место в памяти. С __slots__ интерпретатор использует фиксированную структуру, которая хранит атрибуты напрямую, без использования словаря.

Листинг 16.17

```
class Without_slots: # Класс без __slots__
    def __init__(self, name, age):
        self.name = name
        self.age = age

class With_slots: # С использованием __slots__
    __slots__ = ('name', 'age')

    def __init__(self, name, age):
        self.name = name
        self.age = age

import sys

# Проверяем размер одинаковых объектов разных классов
Ulrik1 = Without_slots("Ульрик", 30)
Ulrik2 = With_slots("Ульрик", 30)

print(sys.getsizeof(Ulrik1.__dict__)) # Словарь (296 байт)
print(sys.getsizeof(Ulrik2)) # Объект с __slots__ (48 байт)
```

Так как словарь исключается, доступ к атрибутам становится быстрее, а объекты занимают меньше памяти. Поэтому в больших программах *slots* используется еще и для оптимизации.

16.6. Полиморфизм

Полиморфизм буквально означает "много форм" (от греческого "поли" — много, "морф" — форма). В программировании это используется, когда одна и та же операция может выполняться по-разному в зависимости от контекста.

Например, если мы попросим кошку издать звук, она мяукнет. Если мы попросим собаку издать звук, она залает. Если мы попросим утку, она закрякает. Одна и та же команда "издай звук" (операция) выполняется по-разному в зависимости от объекта, к которому она применена. Это и есть полиморфизм.

Python, как динамически типизированный язык, поддерживает полиморфизм "из коробки". Это значит, что функции и методы могут работать с любыми объектами, которые имеют необходимые методы или свойства. Для этого не важно, к какому классу принадлежит объект, главное, чтобы он имел нужное поведение.

Ранее мы уже сталкивались с полиморфизмом, когда изучали первые разделы. К примеру, мы использовали символ "+", когда нам нужно было сложить два числа, или сцепить две строки, или соединить два списка. В зависимости от полученных объектов знак плюс выполнял определенные операции.

Листинг 16.18

```
def func_plus(x, y):
    # В зависимости от полученных данных
    # функция выполняет разные действия
    return x + y

print(func_plus (2, 3)) # 5
print(func_plus ('поли', 'морф')) # Полиморф
print(func_plus(['список_1'],['список_2']))
# ['список_1', 'список_2']
```

Еще один пример встроенного полиморфизма — функция len(). Она принимает разные объекты и работает с ними, если они поддерживают соответствующие методы.

Листинг 16.19

```
print(len("полиморф")) # Считаем количество букв
print(len([1, 2, 3, 4])) # Количество элементов в списке
print(len((1, 2, 3))) # Количество элементов в кортеже
```

Пример с животными может выглядеть следующим образом:

Листинг 16.20

```
class Cat:
    def make_sound(self):
        return "Мяу"

class Dog:
    def make_sound(self):
        return "Гав"

class Duck:
    def make_sound(self):
        return "Кря"

# Функция для вызова метода make_sound
def animal_sound(animal):
    print(animal.make_sound())

# Создаем животных
cat = Cat()
dog = Dog()
duck = Duck()

# Проверяем
animal_sound(cat) # Мяу
animal_sound(dog) # Гав
animal_sound(duck) # Кря
```

В примере функция ***animal_sound*** принимает объект любого класса. А затем вызывает метод **make_sound**. Так как у каждого класса этот метод выполняет разные действия, мы получаем разный результат, используя при этом одну общую команду для всех. Это позволяет значительно сократить код и сделать его универсальным для разных случаев.

Полиморфизм часто используется в связке с наследованием. Давайте создадим базовый класс и несколько дочерних классов:

Листинг 16.21

```
class Animal:
    def make_sound(self):
        return "Общий звук для остальных животных"

class Cat(Animal):
```

```
    def make_sound(self):
        return "Мяу"

class Dog(Animal):
    def make_sound(self):
        return "Гав"

class Duck(Animal):
    def make_sound(self):
        return "Кря"

class Cow(Animal):
    pass # Нет присвоенного звука

# Создаем разных животных
cat = Cat()
dog = Dog()
duck = Duck()
cow = Cow()

spisok = [cat, dog, duck, cow]

# Цикл для запуска всех звуков из списка
for i in spisok:
    print(i.make_sound())

# Мяу
# Гав
# Кря
# Общий звук для всех животных
```

Из предыдущих разделов мы знаем, что по принципу иерархии интерпретатор сначала ищет метод в текущем классе и, если не находит его, отправляется в родительский класс. Поэтому в нашем примере каждое животное, имеющее свой класс, использует свой звук, а к остальным животным применяется общий — из родительского класса.

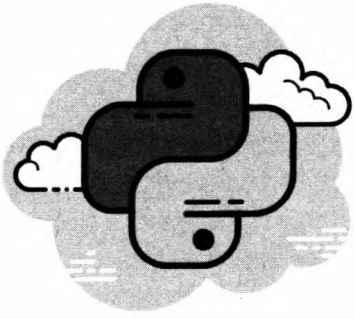
Таким образом мы можем создавать большое количество животных, персонажей, подпрограмм, которые выполняют свои инструкции, но подчиняются общим командам для всех. Это значительно сокращает объем кода, улучшает его читаемость и облегчает разработку.

Однако, несмотря на то что полиморфизм — мощный инструмент, он требует аккуратного подхода. Его использование может привести к ошибкам,

если вы передадите объект, который не поддерживает ожидаемые методы. Также всегда важно документировать, какие методы и свойства требуются, чтобы избежать путаницы.

16.6.1. Практические задания

1. Создайте базовый класс с методом, который выполняет любые действия. Затем создайте два дочерних, которые переопределяют этот метод. Далее создайте два объекта — по одному для каждого подкласса — и вызовите для них измененный метод.
2. Создайте базовый класс для сотрудников с атрибутами имени и зарплаты, а также с методом, который выводит информацию о сотруднике. Затем создайте подкласс, который добавляет атрибут офиса (место работы сотрудника), и переопределите метод, выводящий информацию, чтобы дополнительно отобразить место работы. Используйте функцию super() для вызова метода из базового класса.



Глава 17.

Модули и пакеты

17.1. Что такое модуль?

Модули в Python — это файлы с расширением .py, содержащие код. Это может быть набор функций, классов, переменных или даже исполняемый код.

Главная цель модулей — разделение скриптов на логически связанные части, чтобы сделать программы более структурированными, читаемыми и удобными для повторного использования.

Вместо того чтобы писать одни и те же функции в каждом проекте, вы можете хранить их в модуле (файле .py) и использовать где угодно по необходимости. Также мы можем импортировать чужие модули, если кто-то ранее уже реализовал функционал, который нам нужен. Это позволяет экономить время и не изобретать колесо, так как его уже изобрели до нас. Осталось только взять его и встроить в свой проект. Кроме того, модули помогают изолировать части кода, чтобы переменные и функции из одного модуля не мешали другим.

Чтобы использовать код из модуля, его нужно импортировать с помощью ключевого слова **import**.

Листинг 17.1

```
import math # Импортируем модуль math для математических операций

print(math.sqrt(16)) # Вычисляем квадратный корень числа 16
```

В строке *print(math.sqrt(16))* мы обращаемся к встроенному модулю **math**, содержащему функции для работы с математическими операциями.

.sqrt — вызываем функцию вычисления квадратного корня, передав число 16.

Если нам нужны только отдельные функции или переменные, мы можем импортировать их напрямую с помощью связки **from ... import**. Преимущество такого подхода в том, что нам не нужно писать имя модуля перед каждой функцией (**math**.sqrt(16)).

Листинг 17.2

```
from math import sqrt, pi # из модуля math импортировали только sqrt и pi

print(sqrt(25)) # Используем функцию sqrt без обращения к модулю math.
print(pi) # Используем константу pi без обращения к модулю math.
```

Иногда модули могут иметь длинные названия. Поэтому для удобства им можно присвоить псевдоним с помощью ключевого слова **as**.

Листинг 17.3

```
import math as m # Присвоили модулю псевдоним "m"

print(m.sqrt(9)) # Теперь используем псевдоним
```

С помощью связки **from ... import *** (со звездочкой) можно импортировать все содержимое из модуля. Однако этот способ не рекомендуется, так как он может привести к конфликтам имен.

То есть, если в вашем скрипте существует переменная или функция с определенным именем и при импорте будет перенесена переменная с таким же именем, одна из них перезапишет другую. В итоге вы потеряете определенные данные или функциональность.

Листинг 17.4

```
from math import * # Импортировали все из модуля

print(sin(0)) # Используем функцию sin из модуля
```

```
# Создали свою переменную с тем же именем
sin = "Привет"
# Снова обращаемся к функции из модуля
print(sin(0)) # Ошибка!
```

Помимо того, что вы можете случайно использовать то же имя, перезаписав функцию, в проекте может понадобиться импорт нескольких разных модулей, в которых также под одинаковыми именами могут быть сохранены разные данные или функции. Именно потому, что все имена предусмотреть невозможно, импорт всех имен из модуля связкой со звездочкой **from math import *** не рекомендуется. Но о такой возможности нужно знать, так как в некоторых случаях это может пригодиться.

17.2. Стандартная библиотека Python

При установке интерпретатора Python в комплекте с ним идет большое количество модулей, входящие в его стандартную библиотеку. Эти модули предлагают готовые решения для множества задач: от работы с файлами и сетями до математических вычислений и управления временем. Благодаря стандартной библиотеке мы можем использовать различные готовые решения без необходимости создавать их с нуля или устанавливать дополнительные пакеты.

Например, модуль **math**, с которым мы встречались в предыдущих разделах, является одним из них. Также популярными модулями являются следующие приведенные далее в главе.

Модуль **datetime** — для работы с датой и временем.

Листинг 17.5

```
from datetime import datetime

now = datetime.now() # Текущая дата и время
print(now)
print(now.strftime("%Y-%m-%d %H:%M:%S")) # Форматирование даты
```

Модуль **random** — для генерации случайных чисел.

Листинг 17.6

```
import random

# Случайное целое число от 1 до 10
print(random.randint(1, 10))
# Случайный выбор из списка
print(random.choice(["красный", "синий", "зеленый"]))
```

Модуль **os** — для работы с файловой и операционной системами.

Листинг 17.7

```
import os

print(os.getcwd()) # Текущая рабочая директория
os.mkdir("новая_папка") # Создание новой папки
```

Модуль **sys** — для работы с интерпретатором Python.

Листинг 17.8

```
import sys

print(sys.version) # Версия Python
print(sys.path) # Пути поиска модулей
```

Модуль **json** — для работы с JSON-данными.

Листинг 17.9

```
import json

data = {"имя": "Иван", "возраст": 30}
```

```
json_data = json.dumps(data) # Преобразуем словарь в JSON-строку
print(json_data)

parsed_data = json.loads(json_data) # Обратно в словарь
print(parsed_data)
```

Модуль **re** — для работы с регулярными выражениями.

Листинг 17.10

```
import re

numbers = r"\d+" # Ищем числа
text = "В этой строке 123 есть числа 456"
matches = re.findall(numbers, text)
print(matches) # ['123', '456']
```

Модуль **collections** — расширяет возможности стандартных структур данных.

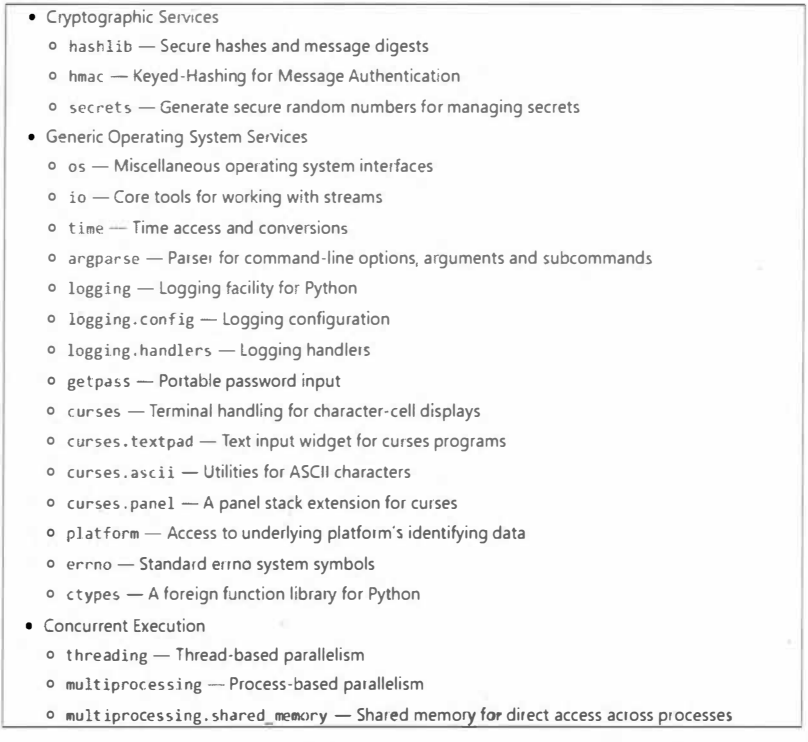
Листинг 17.11

```
from collections import Counter

text = "яблоко банан яблоко"
# Считаем количество повторений слов в тексте
word_counts = Counter(text.split())
print(word_counts) # Counter({'яблоко': 2, 'банан': 1})
```

Подробнее об этих модулях и множестве других из стандартной библиотеки можно почитать в официальной документации, на странице:

https://docs.python.org/3/library/.

Если вы столкнулись с какой-то стандартной задачей, велика вероятность, что какой-то модуль уже решает ее.



Изображение 17.1.

17.3. Создание собственного модуля

Итак, модуль в Python — это просто файл, который содержит скрипты с функциями, классами, переменными и т.д. Когда вы пишете большой проект, логично разделить его на небольшие части, чтобы каждая выполняла свою задачу. Например, один модуль может отвечать за работу с базой данных, другой — за обработку пользовательского ввода, третий — за отображение данных.

Создание собственного модуля — это просто сохранение кода в файле с расширением .py. Например, напишите код, который выполняет определенные задачи, и сохраните файл с любым удобным названием.

first_module.py

```
# first_module.py

# функция для приветствия
def greet(name):
    return f"Привет, {name}!"

# функция для вычисления суммы
def add(a, b):
    return a + b

# Переменная
PI = 3.14159
```

Изображение 17.2.

Теперь, чтобы воспользоваться модулем в другой программе, его нужно импортировать. Создайте новый файл и импортируйте в него свой модуль.

file_1.py

```
import first_module # Импортируем модуль

print(first_module.greet("Иван"))     # Используем функцию greet
print(first_module.add( a: 10,  b: 5)) # Используем функцию add
print(first_module.PI)                # Получаем значение переменной PI
```

first_module.py

```
# first_module.py

# функция для приветствия
def greet(name):
    return f"Привет, {name}!"

# функция для вычисления суммы
def add(a, b):
    return a + b

# Переменная
PI = 3.14159
```

Run file_1

```
Привет, Иван!
15
3.14159

Process finished with exit code 0
```

Изображение 17.3.

Как видим, из-за длинного названия модуля вызов функций также имеет длинную запись. Для удобства мы можем присвоить модулю псевдоним или импортировать только некоторые элементы модуля.

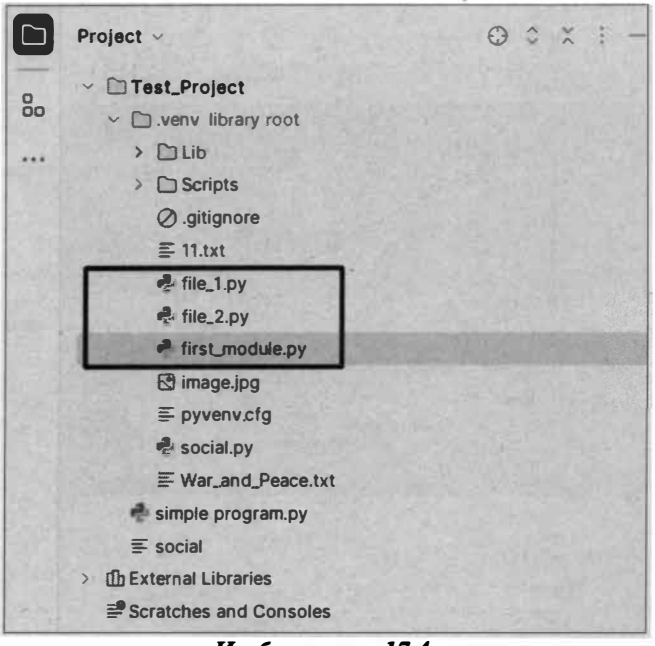
Листинг 17.12

```
import first_module as fm # Псевдоним

print(fm.greet("Иван"))
print(fm.add(10, 5))
print(fm.PI)
```

Листинг 17.13

```
from first_module import greet, add # Импорт двух функций

print(greet("Иван"))
print(add(10, 5))
```

Обратите внимание, что Пайтон ищет модули в определенных местах. Сначала в текущей рабочей директории (где запущен ваш скрипт). Затем в стандартных системных каталогах Python. И далее в путях, указанных в переменной окружения sys.path.



Изображение 17.4.

Чтобы узнать, где интерпретатор ищет модули, используйте следующую команду:

Листинг 17.14

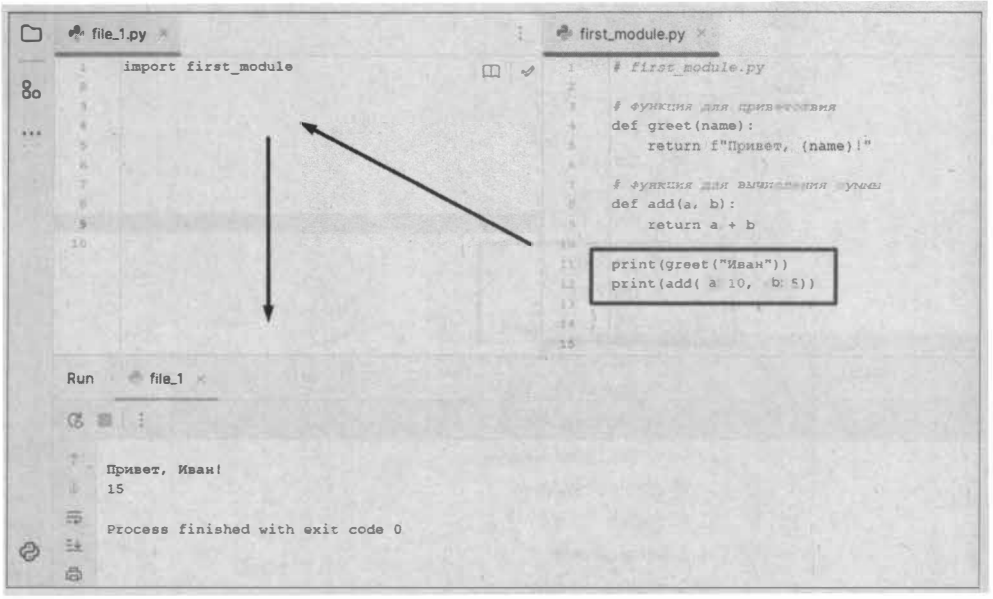
```
import sys
print(sys.path)
```

Вы также можете добавить свои директории, если модуль находится в нестандартном месте.

Листинг 17.15

```
import sys
sys.path.append('/путь/к/директории/с/модулем')

import first_module
```

Еще один важный момент — это запись условия для выполнения определенного кода в нашем скрипте. Представим ситуацию, что в модуле у нас записаны определенные команды на выполнение. Если импортировать данный модуль, эти команды будут выполнены автоматически.



Изображение 17.5.

На изображении запущен file_1.py, в котором импортирован *first_module*, и инструкции из него были выполнены. Однако в некоторых случаях нам не нужно, чтобы определенные функция из импортируемого файла выполнялись самостоятельно.

Для этого используется условие if __name__ == "__main__".

> Каждый файл в Пайтон имеет специальную переменную __name__.

Эта переменная указывает, как был запущен данный файл. Если файл запущен напрямую (например, через команду *python first_module.py*), то значение __name__ будет равно "__main__".

first_module.py

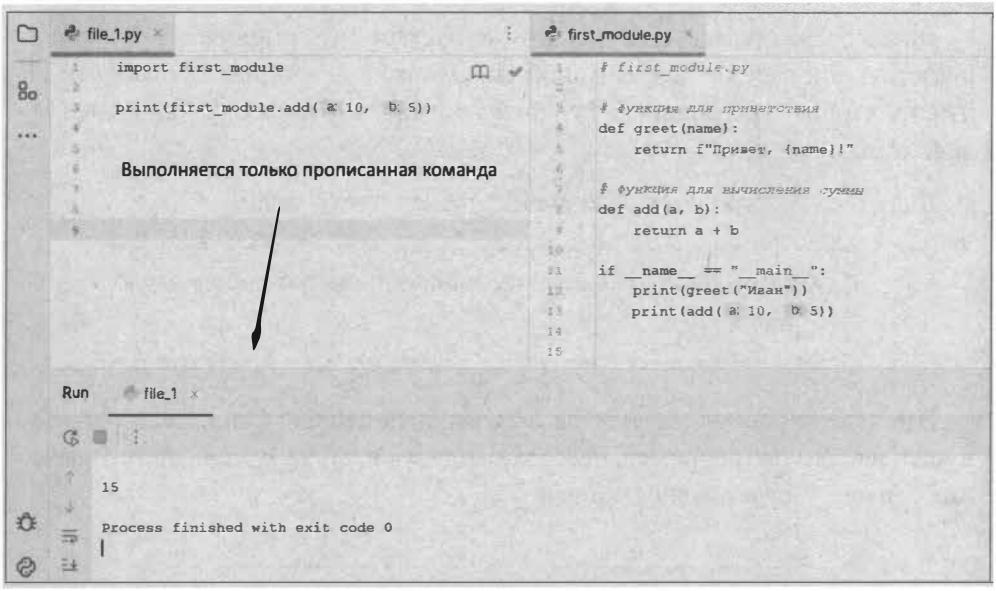
```
# first_module.py

# Функция для приветствия
def greet(name):
    return f"Привет, {name}!"

# Функция для вычисления суммы
def add(a, b):
    return a + b

if __name__ == "__main__":
    print(greet("Иван"))
    print(add( a: 10,  b: 5))
```

При запуске этого файла __name__ равно __main__

Изображение 17.6.

Если файл импортируется как модуль, то значение __name__ будет равно имени этого модуля (first_module.py). То есть условие не будет выполнено, и команды не сработают.

Теперь мы можем запустить только те инструкции из модуля, которые нам нужны:



Изображение 17.7.

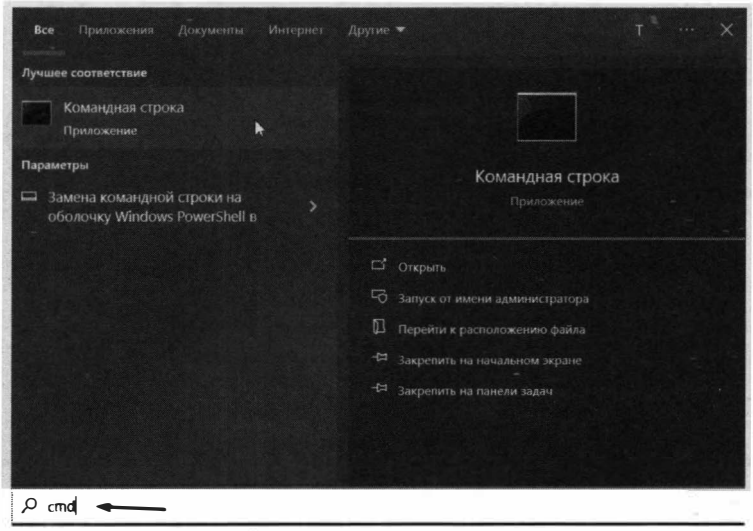
Таким образом, конструкция if __name__ == "__main__" позволяет разделить исполняемый код и определения (функций, классов и т.д.) в модуле. Это полезно, чтобы модуль мог выполняться как самостоятельная программа или использоваться как библиотека, без запуска лишнего кода.

Также не забывайте документировать ваши модули, чтобы при обращении к ним через долгий промежуток времени вы могли вспомнить, для чего именно они предназначены. Это облегчит работу не только вам, но и другим разработчикам, использующим ваш код. Добавляйте комментарии и строку документации (docstring). Не называйте модули именами, совпадающим с названиями стандартных модулей, например math.py.

17.4. Установка внешних модулей

Помимо модулей, идущих в комплекте с интерпретатором Python, существует множество других, которые разрабатываются сторонними разработчиками. Такие модули называются внешними (не входящими в стандартную библиотеку). Внешний модуль представляет собой такую же программу на Пайтон, которая была написана другим программистом, и в ней уже заложена какая-то функциональность.

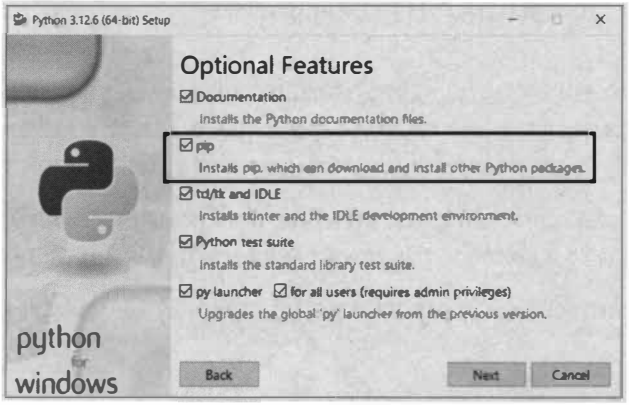
Существует несколько способов установить сторонний модуль. Основной — это использовать командную строку Windows. В поисковую строку на компьютере введите *cmd* и откройте приложение "Командная строка".



Изображение 17.8.

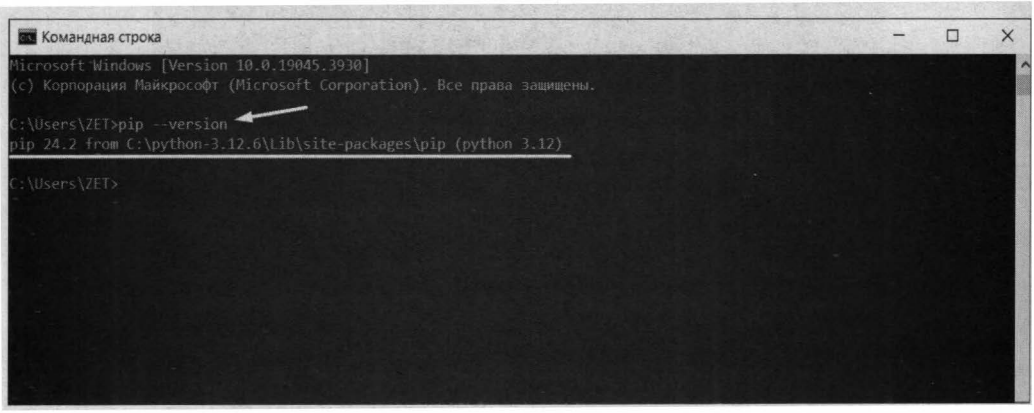
> Для установки модулей используется программа **pip** — пакетный менеджер, который автоматически скачивает и устанавливает библиотеки.

В последних версиях Пайтона он устанавливается автоматически вместе с самим интерпретатором, но, если у вас возникли какие-либо проблемы, его можно установить дополнительно или переустановить Python, поставив галочку на соответствующий пункт.



Изображение 17.9.

Для проверки, установлен ли у вас пакетный менеджер, введите в консоли команду *pip --version*. Ответом будет отображена версия менеджера и директория, если он установлен.



Изображение 17.10.

Для установки модуля используется команда:

pip install имя_модуля

Одними из самых популярных внешних модулей являются:

NumPy — для работы с массивами и числовыми вычислениями;

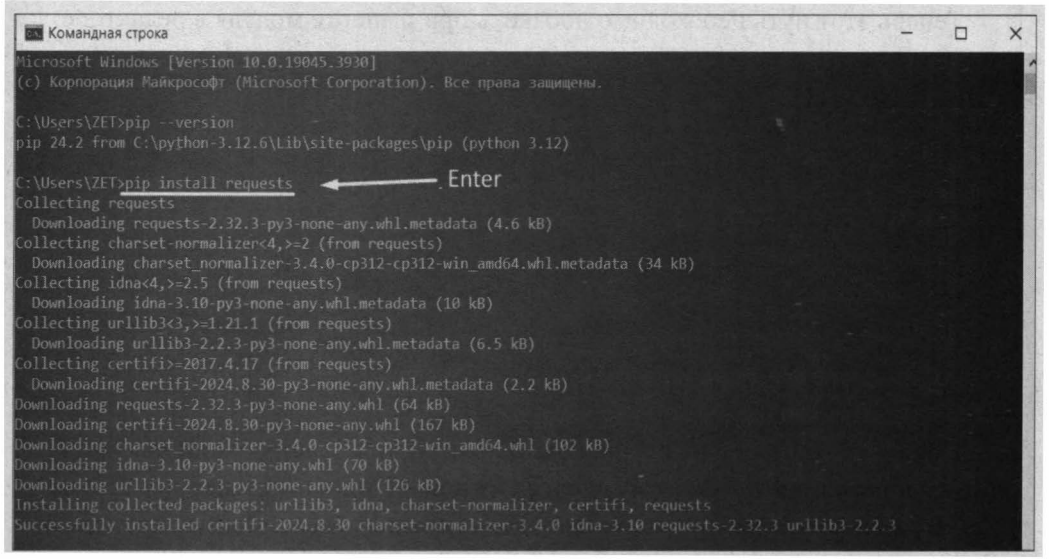
Pandas — для анализа данных;

Requests — для работы с HTTP-запросами.

Также на сайте https://pypi.org/ можно найти внешние модули практически для любых задач. В поисковой строке сайта введите интересующую вас тему, и в ответ будет предложено множество модулей данной тематики.

Давайте установим один их них для практики инсталляции сторонних модулей. Введите в консоли ***pip install requests*** и нажмите **Enter**.

Затем дождитесь, когда **pip** скачает и установит библиотеку.



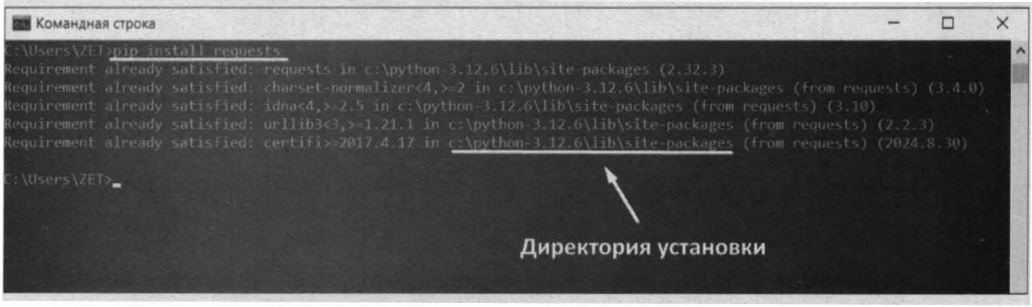
Изображение 17.11.

Чтобы убедиться, что модуль установлен, можно импортировать его в вашу программу. Напишите в редакторе кода:

Листинг 17.16

```
import requests
print("Requests успешно установлен!")
```

Если ошибок нет, значит, модуль установлен корректно. В противном случае есть вероятность, что интерпретатор не может найти путь к нему. Снова введите в консоли команду *pip install requests* и посмотрите путь установки.



Изображение 17.12.

Теперь этот путь необходимо добавить при импорте модуля в редакторе:

Листинг 17.17

```
import sys
# Добавляем путь к модулю в список директорий
sys.path.append('C:/python-3.12.6/Lib/site-packages')

# Снова импортируем модуль
import requests
# Проверяем
print("Requests успешно установлен!")
```

Обратите внимание на слеши (косые черты), при копировании они могут быть "обратными" (\), поэтому следует их исправить на "прямые" (/). Также имейте в виду, что, если у вас установлено несколько версий интерпретатора Python, для каждого из них требуется отдельная установка модулей.

Иногда нужно установить определенную версию модуля. Для этого используется команда *pip install имя_модуля==версия*.

Листинг 17.18

```
pip install requests==1.1.0
```

Чтобы обновить модуль до последней версии, используется команда *pip install --upgrade имя_модуля*.

Листинг 17.19

```
pip install --upgrade requests
```

Для удаления модуля требуется команда *pip uninstall имя_модуля*.

Листинг 17.20

```
pip uninstall requests
```

Для просмотра списка всех установленных модулей введите в консоли команду *pip list*.

Листинг 17.21

```
pip list
```

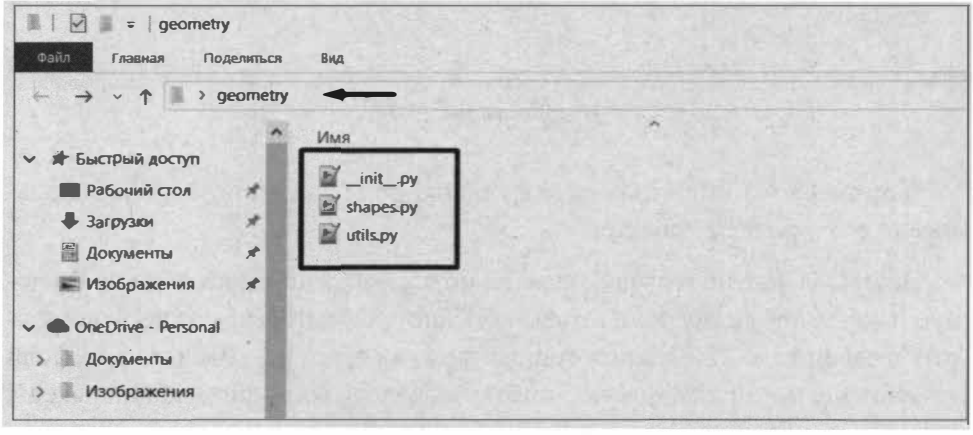
Иногда на одном компьютере установлено несколько версий Python (например, Python 2 и Python 3). Чтобы использовать **pip** для конкретной версии, используйте *pip3 install имя_модуля* — для третьей версии и *pip install имя_модуля* — для второй версии.

17.5. Пакеты

Модули помогают организовать код в одном файле, а пакеты — это способ объединить связанные модули в одну структуру.

Пакет — это каталог (папка) с модулями и другими пакетами, организованный таким образом, чтобы Python мог работать с ними как с единым целым.

В отличие от модуля, который представляет собой отдельный .py-файл, пакет состоит из нескольких файлов и может включать обычные .py-файлы, другие каталоги с модулями, а также специальный файл __init__.py, который позволяет интерпретатору считать папку пакетом.

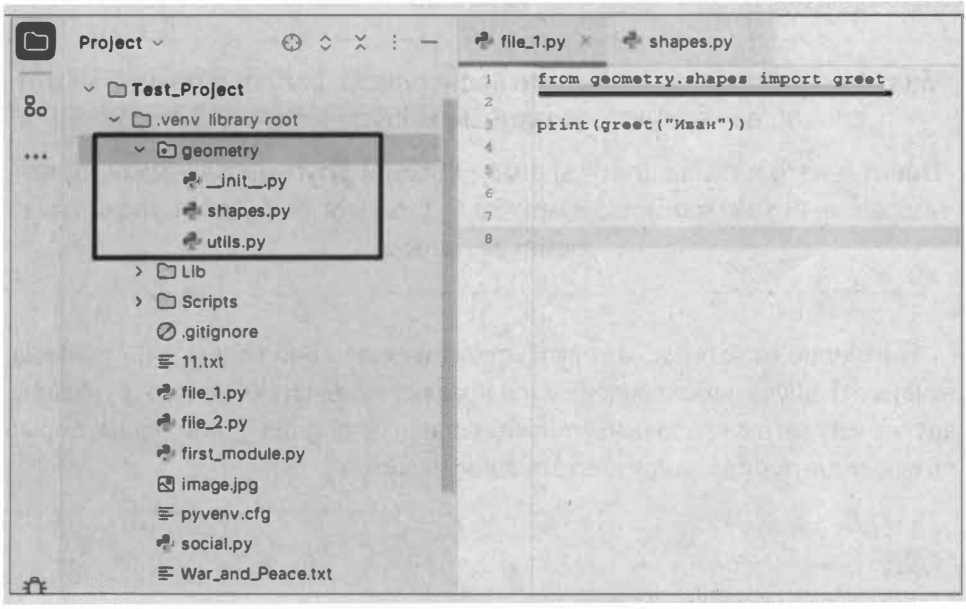


Изображение 17.13.

На изображении мы создали папку geometry, содержащую все файлы пакета. Файл __init__.py (может быть пустым), сообщает Пайтону, что папка является пакетом. Файлы shapes.py и utils.py являются модулями, в которых прописан определенный функционал.

Преимущество пакетов в том, что они позволяют структурировать большую программу. Кроме того, мы можем использовать один и тот же пакет в разных программах. Вместе с тем пакеты позволяют разделить логику программы по независимым модулям, чтобы инкапсулировать определенный функционал.

Чтобы использовать созданный пакет, поместите его в директорию вашей программы, а затем импортируйте его также как модули.

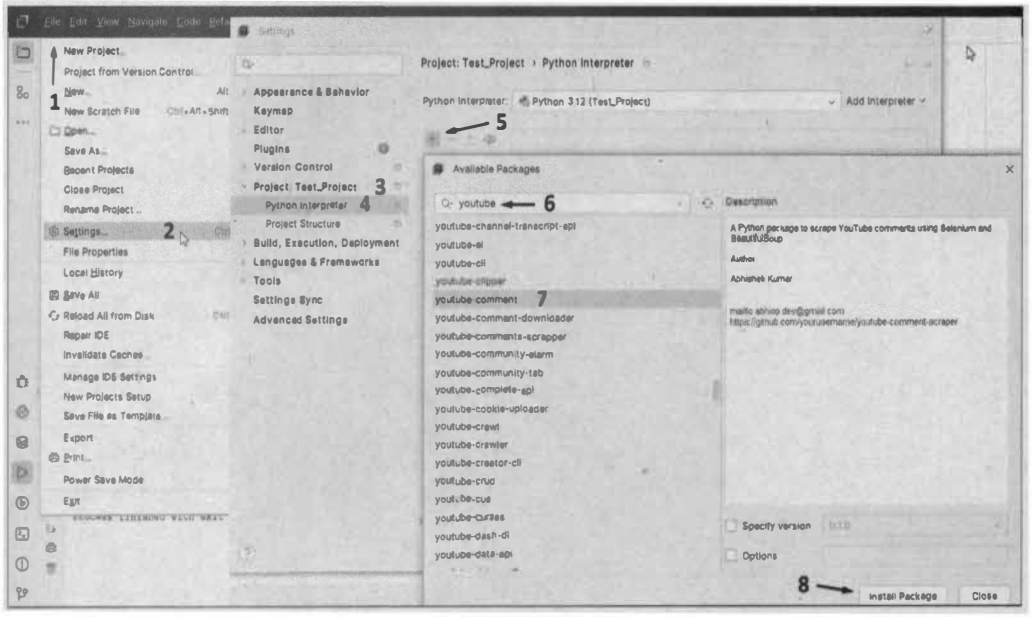


Изображение 17.14.

В примере мы импортировали функцию ***greet*** из пакета, а затем использовали ее в текущем проекте.

Пакеты можно не только создавать, но и устанавливать сторонние. Внешние пакеты предоставляют готовые библиотеки, которые можно использовать в своем коде. Их можно установить также через **pip**. Введите в консоли команду *pip install имя пакета*, чтобы установить соответствующий пакет.

Некоторые среды разработки включают в себя инструменты для установки пакетов. Например, если вы используете PyCharm, пакеты можно устанавливать с его помощью. Откройте настройки: File > Settings > Project > Python Interpreter. Здесь вы увидите список установленных пакетов. Чтобы установить новые, нажмите "+", в открывшемся окне введите в поисковую строку название интересуемого пакета, в появившемся списке похожих выберите необходимый и нажмите Install Package.



Изображение 17.15.

После этого запустится процесс установки, а затем установленный пакет отобразится в списке установленных. Далее вы можете импортировать его в свой проект стандартной командой *import*.

17.5.1. Практические задания

1. В папке **geometry** в модуле shapes.py реализуйте две любые функции. Напишите программу, которая импортирует эти функции и производит соответствующие вычисления.

2. Импортирование всего пакета: создайте пакет text_utils, содержащий модули string_ops.py и word_counter.py. Каждый из модулей должен содержать любую функцию для обработки данных типа "строка". Напишите программу, которая импортирует весь пакет и выполняет функции из обоих модулей.

Глава 18.

Тестирование и отладка

18.1. Зачем нужно тестирование

Тестирование в программировании — это процесс проверки кода на наличие ошибок, корректность выполнения скриптов и соответствие ожидаемому поведению. В Python существует множество инструментов для ручного и автоматизированного тестирования, которые помогают убедиться в том, что код работает так, как нужно, даже после внесения изменений.

Отладка — это пошаговый процесс поиска и исправления ошибок в коде. Она позволяет понять, где и почему программа ведет себя неожиданным образом, и внести соответствующие изменения.

Прежде чем начать отладку, важно понимать, какие ошибки можно встретить:

- **Синтаксические ошибки (SyntaxError)** — возникают, если код написан с нарушением правил Пайтона. Например, пропущено двоеточие, скобка или неправильно написан оператор.
- **Ошибка выполнения (RuntimeError)** — появляется при запуске программы. Например, при делении на ноль или попытке обращения к несуществующему элементу списка.
- **Логические ошибки** — программа работает, но результат отличается от ожидаемого. Например, неправильная формула расчета.

Python предоставляет несколько инструментов и подходов для отладки. Самый простой и популярный метод — добавление команды ***print*** в код, чтобы увидеть значения переменных или шаги выполнения.

Листинг 18.1

```
def calculate_area(radius):
    print(f"Radius: {radius}") # Проверяем значение
    area = 3.14 * radius ** 2
    print(f"Calculated area: {area}") # Проверяем результат
    return area

calculate_area(5)
```

То есть на каждом шаге выполнения программы мы, с помощью ***print***, можем отследить текущие значения, чтобы понимать, в какой момент в скрипте появляется ошибка. Недостаток этого метода в том, что в больших программах это неудобно и затратно по времени.

Следующий способ — логирование. Это более профессиональный подход, чем ***print***. Он позволяет записывать информацию о выполнении программы в консоль или файл.

Листинг 18.2

```
import logging

logging.basicConfig(level=logging.INFO)

def calculate_area(radius):
    logging.info(f"Radius: {radius}") # Логируем значение
    area = 3.14 * radius ** 2
    logging.info(f"Calculated area: {area}") # Логируем результат
    return area

calculate_area(5)
```

Преимущество этого метода в том, что мы можем сохранять логи в файл для последующего анализа.

Современные среды разработки, такие как PyCharm, VS Code и другие, предлагают удобные встроенные инструменты для отладки.

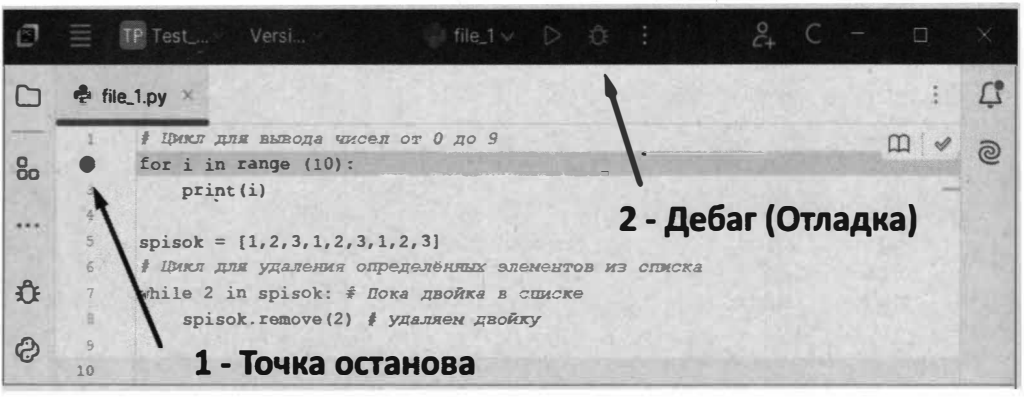
Их преимущества в том, что мы можем пошагово пройти всю программу или определенный участок кода, чтобы выявить ошибки.

Разберем на примере PyCharm, напишите любой небольшой скрипт:

Листинг 18.3

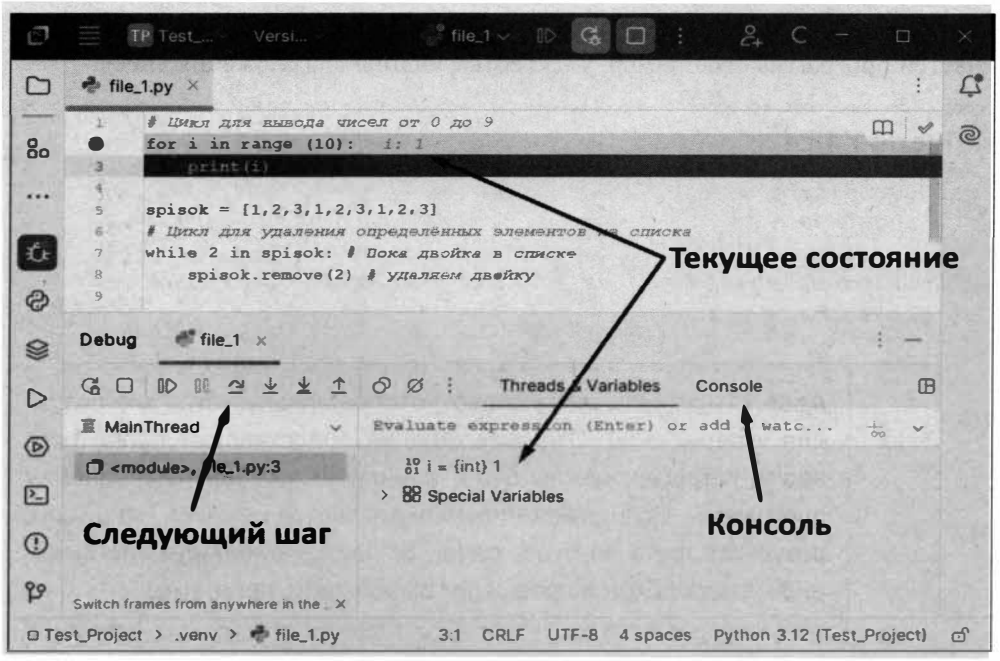
```
# Цикл для вывода чисел от 0 до 9
for i in range (10):
    print(i)

spisok = [1,2,3,1,2,3,1,2,3]
# Цикл для удаления определенных элементов из списка
while 2 in spisok: # пока двойка в списке
    spisok.remove(2) # Удаляем двойку
```

Для тестирования определенного участка кода существуют точки останова, которыми мы можем указать интерпретатору, что именно мы хотим изучить пошагово. Затем необходимо нажать кнопку дебага (отладки кода). Если у вас открыто несколько файлов, нажмите правую кнопку мыши на нужном файле, и выберите пункт "Debug имя_файла", чтобы запустить нужный.



Изображение 18.1.

После этого в нижней части редактора откроется панель отладки. В ней пошагово, нажимая на соответствующую кнопку, можно отследить выполнение кода. Например, в какой момент, какая переменная меняет свое значение. Нажав на вкладку консоли (Console), можно увидеть, что сейчас происходит в программе.



Изображение 18.2.

Для остановки отладки нужно нажать клавишу **Stop** в виде красного квадрата, расположенного на месте кнопки запуска дебага. Таким образом мы можем пройти всю программу по шагам, чтобы увидеть, в каком месте у нас возникает ошибка. Также это поможет разобраться в чужом коде, который на первый взгляд может быть не понятен. Пройдя его последовательно, вы увидите, за что отвечает каждая инструкция и что происходит после ее выполнения.

18.2. Модульное тестирование

Программирование без тестирования — это как строительство дома без проверки устойчивости стен. Вы можете создать что-то масштабное, но как долго это продержится? Модульное тестирование позволяет проверить, как работает каждый отдельный "строительный блок" в вашей программе.

В Пайтон модулем может быть функция, класс или метод. Цель тестирования — убедиться, что модуль работает корректно в изоляции от остальных частей программы. Например, у нас есть функция для сложения чисел:

Листинг 18.4

```
def summa(a, b):
    return a + b

# Проверяем, возвращает ли функция правильный результат
assert summa(2, 3) == 5
```

> **Assert** — это ключевое слово, которое используется для создания утверждений. Эти утверждения представляют собой проверки, которые должны быть истинными во время выполнения программы. Если утверждение оказывается ложным, например, результат теста не будет равен 5, Пайтон генерирует исключение *AssertionError* и прерывает выполнение программы.

Assert помогает обнаружить логические ошибки в коде на ранней стадии разработки. Например, вы можете использовать ***assert*** для проверки того, что функция получает данные правильного типа или что переменная имеет ожидаемое значение.

Благодаря модульному тестированию можно проверить, что каждая часть программы выполняет свою задачу. Если что-то сломается, тесты помогут сразу понять, где именно произошла ошибка. Когда программа будет доработана или отредактирована, модульное тестирование подтвердит, что изменения не привели к новым ошибкам. Python также предоставляет встроенные и сторонние инструменты для тестирования.

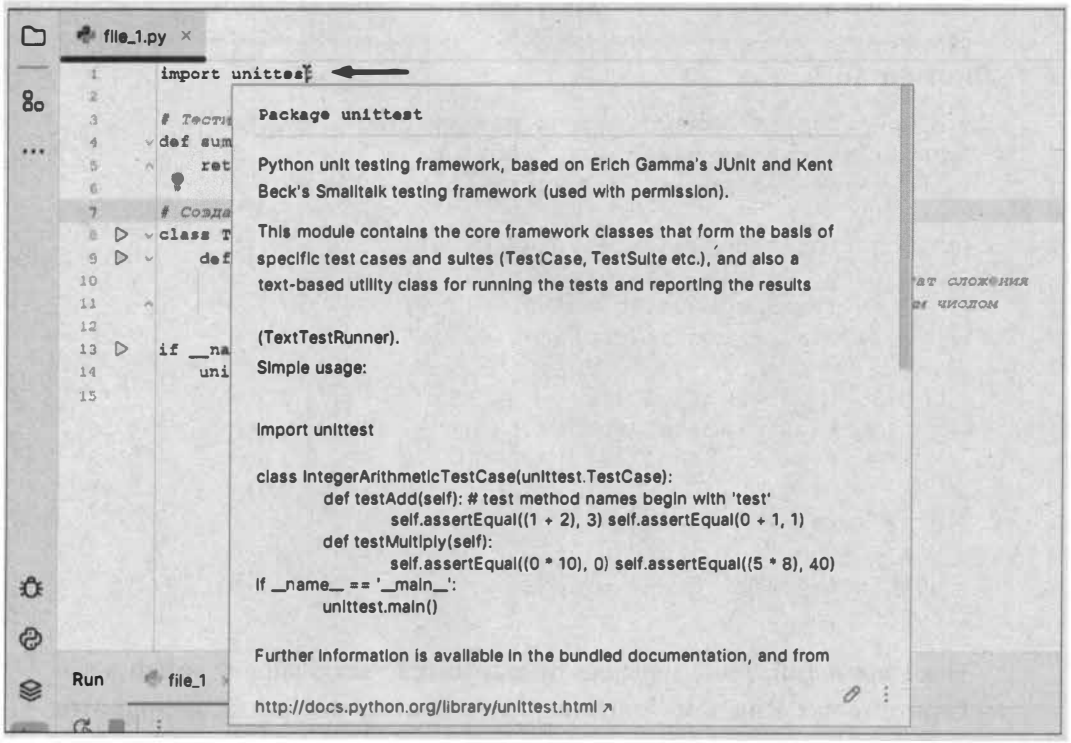
> Модуль **unittest** является встроенной библиотекой для создания и выполнения тестов. Он предлагает функционал для написания тестов, проверки их выполнения и получения отчетов.

Листинг 18.5

```
import unittest

# Тестируемая функция
```

```
def summa(a, b):
    return a + b

# Создаем класс для тестов
class Test_Functions(unittest.TestCase):
    def test_add(self):
        # Проверяем результат сложения
        self.assertEqual(summa(2, 3), 5)
        # Тест с отрицательным числом
        self.assertEqual(summa(-1, 1), 0)

if __name__ == "__main__":
    unittest.main() # Запуск тестов
```

Подробнее о работе модуля можно узнать в документации. Если вы работаете в PyCharm, наведите курсор на название модуля и держите пару секунд. После этого откроется справка о модуле. По ссылке в документации можно перейти на соответствующую страницу в сети и перевести текст на русский язык.



Изображение 18.3.

Чтобы писать хорошие тесты, изолируйте тестируемый модуль. То есть тестируйте только одну функцию или метод за раз. Используйте разные входные данные, проверяйте обычные, крайние и неожиданные случаи.

Пример создания модульного тестирования

1. Создадим файл с именем functions.py, в котором будут находиться определенные функции:

Листинг 18.6

```
def summa(a, b): # Сложение
    return a + b

def subtract(a, b): # Вычитание
    return a - b
```

2. Создадим второй файл с именем test_functions.py, в котором будет содержаться функционал для теста функций из файла выше:

Листинг 18.7

```
import unittest # Импорт модуля для тестов
# Импорт функций из файла functions.py
from functions import summa, subtract

class Test_Functions(unittest.TestCase):
    def test_summa(self):
        self.assertEqual(summa(2, 3), 5)
        self.assertEqual(summa(-1, 1), 0)

    def test_subtract(self):
        self.assertEqual(subtract(5, 3), 2)
        self.assertEqual(subtract(0, 3), -3)

# Запуск теста в этом файле
if __name__ == "__main__":
    unittest.main()
```

Находясь в файле test_functions.py, нажмите правую кнопку мыши и выберите пункт **Run test_functions**, это позволит запустить именно этот файл, чтобы удостовериться в корректной работе наших функций.

Модульное тестирование помогает повысить качество кода, сократить количество ошибок и облегчить поддержку проекта. Сначала это может показаться сложным, но с практикой вы научитесь писать тесты быстро и эффективно. Главное помнить, что каждый написанный тест делает ваш код надежнее.

18.3. Интеграционное тестирование

Интеграционное тестирование позволяет провести проверку взаимодействия между модулями или компонентами приложения. Отличие от модульного тестирования заключается в том, что модульное проверяет отдельные модули (функции, классы) изолированно. Тогда как интеграционное тестирование проверяет, как эти модули взаимодействуют друг с другом.

Интеграционное тестирование помогает обнаружить проблемы, которые не видны при модульном тестировании. Позволяя убедиться, что функции и классы корректно "общаются" между собой. Такое тестирование можно выполнять вручную или автоматически. В Пайтон чаще всего используют автоматизированное тестирование.

К примеру, у нас есть приложение, которое обрабатывает заказы пользователей и включает в себя два модуля: модуль расчета цены (calculate_price) и модуль оформления заказа (process_order). Нам нужно убедиться, что эти модули корректно взаимодействуют между собой.

Создадим модуль calculate_price.py:

Листинг 18.8

```
def calculate_price(items):
    # Рассчитывает общую стоимость заказа
    return sum(item['price'] * item['quantity'] for item in items)
```

Далее создадим модуль process_order.py:

Листинг 18.9

```
from calculate_price import calculate_price

def process_order(order):
    # Обработка заказа
    total = calculate_price(order['items'])
    return {
        "order_id": order['order_id'],
        "total_price": total
    }
```

Теперь создадим модуль для тестирования test_integration.py:

Листинг 18.10

```
import unittest
from process_order import process_order

class TestOrderProcessing(unittest.TestCase):
    def test_order_processing(self):
        # Создаем тестовый заказ
        order = {
            "order_id": 1,
            "items": [
                {"name": "Apple", "price": 1.0, "quantity": 5},
                {"name": "Banana", "price": 0.5, "quantity": 10}
            ]
        }

        # Ожидаемый результат
        expected = {
            "order_id": 1,
            "total_price": 10.0
        }

        # Проверяем, что модули работают корректно вместе
        self.assertEqual(process_order(order), expected)

if __name__ == "__main__":
    unittest.main()
```

После запуска модуля test_integration.py (ПКМ > Run test_integration) будет получен результат теста. Если все прошло успешно, модули взаимо-

действуют корректно. Если нет, отобразятся ошибки, которые необходимо исправить. Например, вы можете изменить одно из значений "price" или "quantity" и перезапустить скрипт, чтобы обнаружить соответствующую ошибку. Так как теперь результат не будет совпадать с общей ценой (total_ price).

Существует несколько стратегий интеграционного тестирования:

- **Большой взрыв (Big Bang)** — все модули тестируются сразу вместе. Недостаток в том, что сложно найти источник ошибки, если что-то пошло не так.
- **Инкрементальное тестирование (Incremental)** — тестируются сначала небольшие группы модулей, а затем их объединяют. Удобно для больших систем.
- **Смешанный подход** — сначала проводится модульное тестирование, затем интеграционное.

18.4. Системное тестирование

Системное тестирование — это следующий этап тестирования, на котором проверяется работа всей системы целиком. В отличие от модульного и интеграционного тестирования, которые сосредоточены на отдельных модулях и их взаимодействии между собой.

Продолжая аналогию со строящимся домом, модульное тестирование проверяет прочность каждой отдельной стены, интеграционное тестирование проверяет взаимодействие между стенами, крышей и фундаментом. Системное тестирование проверяет весь дом, работают ли двери, свет, отопление и водопровод.

Данный этап позволяет выявить проблемы, которые не видны на уровнях модульного или интеграционного тестирования, а также убедиться, что приложение удобно и интуитивно для конечного пользователя. Это последний этап проверки перед переходом к приемочному тестированию или релизу.

Системное тестирование включает в себя проверку всех аспектов приложения, от функциональности до производительности и безопасности. На данном этапе происходят следующие тесты:

- **Функциональное тестирование** — проверяется, выполняет ли система заявленные функции.
- **Нефункциональное тестирование** — производительность, надежность, удобство использования и безопасность.
- **Регрессионное тестирование** — позволяет убедиться, что внесенные изменения не нарушили работу системы.
- **Тестирование совместимости** — проверяется работа на разных устройствах, операционных системах или с различными версиями браузеров.

Предположим, вы разрабатываете веб-приложение для онлайн-магазина. Системное тестирование в таком случае может включать в себя следующие проверки:

1. Может ли пользователь успешно зарегистрироваться, войти в систему как ранее зарегистрированный пользователь, выйти из системы.
2. Корректно ли товары добавляются в корзину, можно ли их купить.
3. Корректно ли обрабатывается оплата.
4. В случае ошибок выводится ли пользователю понятное сообщение.

Для системного тестирования есть несколько популярных инструментов:

- Модули **unittest и pytest** — используются для написания и выполнения тестов. Подходят для функционального тестирования.
- Библиотека **Selenium** — используется для автоматизации тестирования веб-приложений. Позволяет взаимодействовать с браузером так, как это делает пользователь.
- **locust** — инструмент для нагрузочного тестирования.
- **pytest-django и flask-testing** — интеграция для тестирования Django или Flask-приложений.
- Библиотеки **requests** или **Postman** — используются для проверки взаимодействия с REST API (API-тестирование).

18.5. Покрытие кода

Покрытие кода — это показатель, который говорит нам, насколько тесты проверяют код программы. Покрытие измеряет, сколько строк, функций или блоков кода были выполнены (или "покрыты") во время тестирования.

Для примера, представьте, что у нас есть 100 строк кода и наши тесты проверяют только 50 из них. Это значит, что покрытие составляет 50%. Если все 100 строк проверяются тестами, то покрытие будет 100%.

Покрытие помогает понять, какие части нашего кода остались непроверенными. Например, логика обработки ошибок или ветки в условиях **if**. Чем больше покрытие, тем выше будет уверенность, что код работает правильно и изменения не приведут к новым ошибкам.

Существует несколько расчетов покрытия кода:

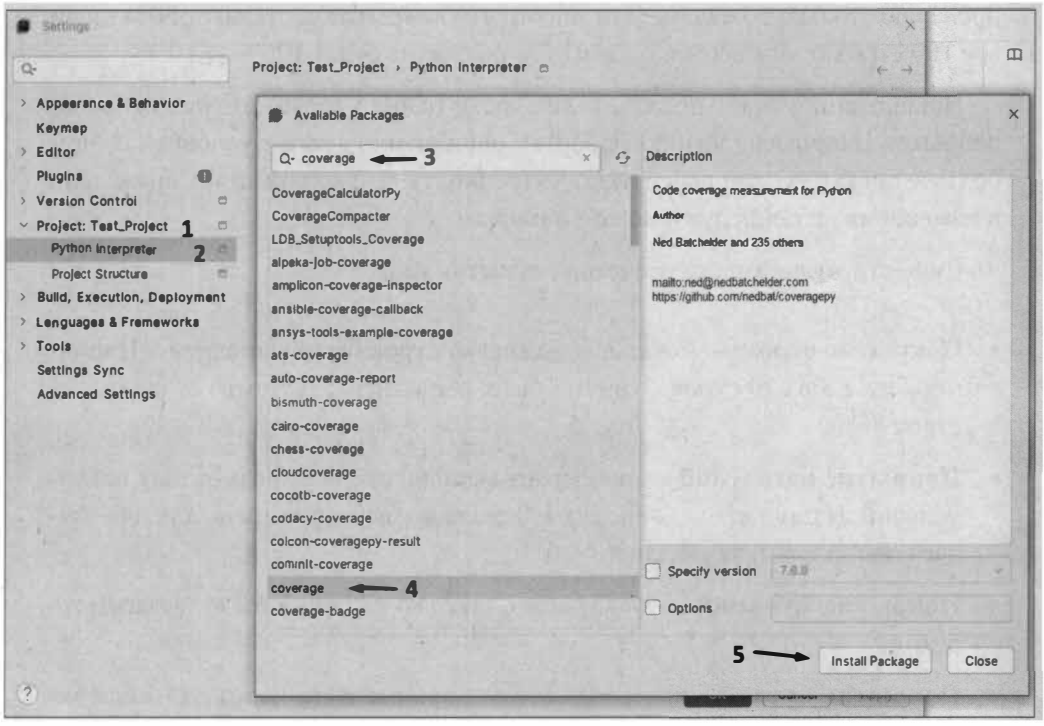
- **Покрытие строк** — показывает, сколько строк было выполнено. Например, если 8 из 10 строк функции были выполнены тестами, то покрытие строк 80%.
- **Покрытие ветвлений** — проверяет выполнение всех возможных ветвей условий. Например, условие if x > 0: должно быть проверено как для случаев, когда x > 0, так и для x <= 0.
- **Покрытие функций** — показывает, сколько функций было вызвано тестами.
- **Покрытие путей** — проверяет все возможные пути через код, включая комбинации условий.

Для измерения покрытия используется библиотека **coverage.py**. Она предоставляет простой способ увидеть, насколько хорошо ваш код протестирован. Чтобы установить модуль, используйте стандартный способ, введя соответствующую команду в командную строку — *pip install coverage*.

```
Командная строка
Microsoft Windows [Version 10.0.19045.3930]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\ZET>pip install coverage
Collecting coverage
  Downloading coverage-7.6.9-cp312-cp312-win_amd64.whl.metadata (8.4 kB)
Downloading coverage-7.6.9-cp312-cp312-win_amd64.whl (210 kB)
Installing collected packages: coverage
Successfully installed coverage-7.6.9
```

Изображение 18.4.

В PyCharm можно установить через встроенный менеджер:



Изображение 18.5.

Для примера работы с инструментом напишем несколько простых функций, а затем протестируем их и определим покрытие:

Листинг 18.11

```
# Файл functions.py
def summa(a, b):
```

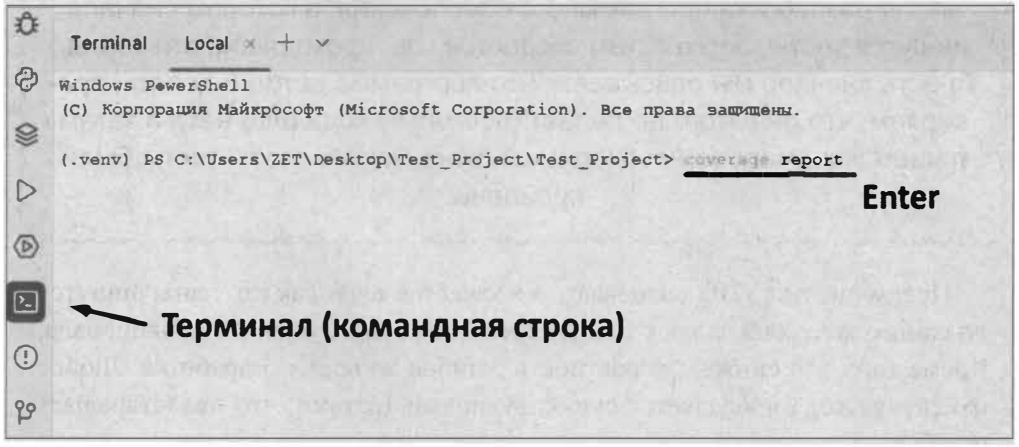
```
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b
```

Протестируем две функции из трех:

Листинг 18.12

```
# Файл test_functions.py
from functions import summa, subtract

def test_summa():
    assert summa(2, 3) == 5

def test_subtract():
    assert subtract(5, 3) == 2
```

Далее в терминале, который аналогичен командной строке, введите команду *coverage report*, чтобы увидеть результат.



Изображение 18.6.

Ответом будет табличка с покрытием кода:

Листинг 18.13

```
Name                          Stmts   Miss  Cover
-------------------------------------------------
.venv\test_functions.py           6      2    66%
```

Здесь 66% означает, что 2 строки остались непокрытыми. Данный отчет можно сохранить в HTML-документе, введя в терминал команду *coverage html*. После этого в вашем проекте будет создана папка htmlcov с html-страничкой, которую можно посмотреть в любом браузере или отправить кому-то для отчетности.

Стоит иметь в виду, что хорошим покрытием считается 80–90% кода. Это позволяет охватить основную логику, не тратя ресурсы на избыточные тесты. Также нужно понимать, что покрытие показывает, какие строки были выполнены, но не говорит о качестве самих тестов. Часто тесты не проверяют обработку ошибок или редкие случаи.

18.6. Тест-драйвенная разработка (TDD)

Тест-драйвенная разработка (Test-Driven Development, TDD) — это метод разработки программного обеспечения, в котором сначала пишутся тесты, а уже затем создается код, проходящий эти тесты. То есть сначала мы описываем, что программа должна делать, проверяем, что она этого не делает (поскольку кода еще нет), а затем пишем минимально необходимый функционал, чтобы тесты были пройдены.

Преимущества TDD заключаются в качестве кода. Так как тесты пишутся до самого кода, они задают точные требования для будущего функционала. Кроме того, это снижает вероятность ошибок во время разработки. Любое изменение кода проверяется существующими тестами, что предотвращает ошибки.

TDD позволяет понять, как код будет использоваться, еще до того, как мы его напишем. Таким образом, тесты выступают в роли живой документации, показывая, как работает код.

Для примера напишем тест для функции, которой еще не существует:

Листинг 18.14

```
# Файл test_functions.py
# Импортируем функцию, которую мы еще не написали
from functions import max_of_two

def test_max_of_two():
    # Находим большее число из двух
    assert max_of_two(3, 5) == 5
    assert max_of_two(10, 7) == 10
    assert max_of_two(4, 4) == 4
```

При запуске этого скрипта (ПКМ > Run test_functions) мы получим ошибку, в которой сообщается, что интерпретатор не знает о существовании функции *max_of_two*. Поэтому далее создается соответствующая функция с минимальными возможностями, которых будет достаточно для прохождения теста.

Листинг 18.15

```
def max_of_two(a, b):
    return a if a > b else b
# Вернуть "а", если "а" больше "б"
# Иначе вернуть "б"
```

Теперь после запуска тесты будут пройдены успешно. Следующий этап — рефакторинг.

Рефакторинг — это улучшение кода, с сохранением его функциональности и повторным тестированием, чтобы убедиться в корректности работы программы после ее доработки.

В нашем случае можно доработать программу таким образом, чтобы при получении двух одинаковых чисел выводилось сообщение, что они равны.

18.6.1. Практические задания

1. Установите самостоятельно модуль **pytest**, аналогично тому, как мы устанавливали модуль **coverage**.
2. Создайте функцию, которая возвращает результат перемножения двух чисел. Напишите тесты для этой функции, чтобы проверить ее корректность.
3. Создайте функцию, которая выполняет деление чисел. Если знаменатель равен нулю, выбрасывается исключение *ValueError* с сообщением "Деление на ноль!". Напишите тесты для этой функции.
4. Запустите каждую из написанных вами функций в режиме отладки и посмотрите пошагово, как они работают.

Глава 19.

Создание графических интерфейсов

19.1. Введение в графические интерфейсы

Графический интерфейс (GUI Graphical User Interface) — это способ взаимодействия пользователя с программой с помощью графических элементов, таких как кнопки, текстовые поля, выпадающие списки, окна, иконки и т.д. Вместо ввода команд в консоль пользователь может кликать мышкой, вводить текст в формы или использовать другие визуальные элементы.

Графический интерфейс делает приложение интуитивно понятным для людей, не знакомых с программированием или командной строкой. С помощью GUI можно создавать сложные приложения, которые легко использовать, от калькуляторов до текстовых редакторов или даже игр. Кроме того, программы с красивым интерфейсом воспринимаются более профессиональными и удобными.

Пайтон предоставляет несколько библиотек для создания графических интерфейсов:

- **Tkinter** — встроенная библиотека.
- **PyQt/PySide** — для создания сложных интерфейсов.
- **Kivy** — для мобильных приложений.
- **wxPython** — мощная библиотека для настольных приложений.

В данной книге мы рассмотрим библиотеку Tkinter, так как она встроена по умолчанию и проста в освоении.

19.2. Основы работы с Tkinter

Для реализации графического интерфейса, в первую очередь, необходимо импортировать в нашу будущую программу библиотеку Tkinter. Для этого создайте новый проект в PyCharm или аналогичном редакторе.

Листинг 19.1

```
# Импортируем библиотеку и присваиваем псевдоним для удобства
import tkinter as tk
```

Следующий шаг — создание окна.

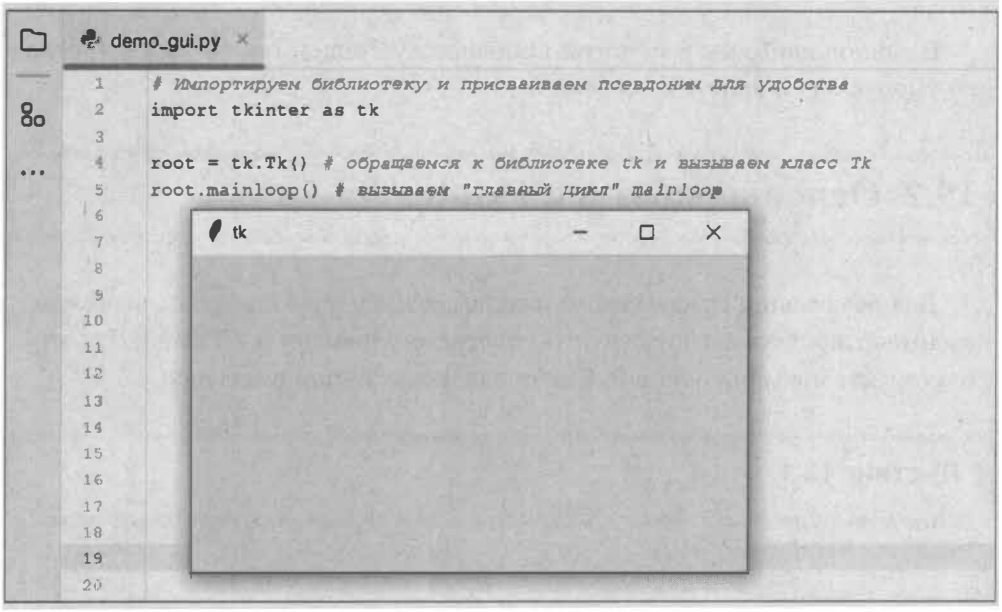
Окно — это основа любого приложения, в котором будут располагаться остальные виджеты (кнопки, флажки, переключатели). В Tkinter его называют "главное окно" (root). Хотя вы можете называть его по-другому. Это просто переменная, хранящая определенные данные.

Листинг 19.2

```
root = tk.Tk() # Обращаемся к библиотеке tk и вызываем класс Tk
root.mainloop() # Вызываем "главный цикл" mainloop
```

Функция **mainloop** отвечает за отображение окна программы в режиме ожидания, пока пользователь не закроет ее или не выполнит другие действия. Если сейчас запустить программу, будет вызвано пустое окно.

Мы можем изменить название программы на свое с помощью соответствующей функции ***title***.



Изображение 19.1.

Листинг 19.3

```
root = tk.Tk()
root.title("Демонстрационная программа")

root.mainloop()
```

Обратите внимание, что команда главного цикла (root.mainloop) располагается в самом низу нашей программы. Хотя на этот счет строгого правила нет, это общепринятая практика. Обычно перед запуском главного цикла необходимо выполнить всю инициализацию приложения: создать окна, разместить элементы управления, настроить обработчики событий и т.д. Размещение ***mainloop*** в конце программы создает логическую последовательность действий.

Кроме того, если функция ***mainloop*** будет вызвана слишком рано, приложение может завершиться до того, как все элементы интерфейса будут отображены и начнут реагировать на события.

Далее, запуская программу, вы можете увидеть, что она всегда открывается в определенном месте и с определенным размером окна. Эти параметры также можно изменить на свои.

Листинг 19.4

```
root.geometry("600x400+200+100")
```

Обратившись к методу **geometry**, мы можем указать размеры окна в пикселях. Между ними обязательно устанавливается английская буква "x". Плюсами устанавливается точка верхнего левого угла программы. В нашем случае 200 пикселей будет отступ от левого края экрана и 100 пикселей от верхнего края.

Вместо точных значений можно прописать переменные, в зависимости от которых размер программы будет изменяться.

Листинг 19.5

```
x = 600
y = 400
root.geometry(f"{x}x{y}+200+100")
```

Это может быть полезно, когда у пользователя маленький экран и в зависимости от его размеров переменные могут принимать меньшие значения, чтобы приложение корректно отображалась на его устройстве.

Следующий параметр *resizable* отвечает за возможность растягивания окна по горизонтали и вертикали. По умолчанию установлен в значениях True. Для примера мы можем изменить эти значения на False, чтобы запретить пользователю изменять размер окна.

Листинг 19.6

```
root.resizable(False, False) # По ширине False, по высоте False
```

Также мы можем указать минимально и максимально возможные размеры, до которых можно уменьшать или растягивать окно программы.

Листинг 19.7

```
root.minsize(300,200) # Минимальный размер окна
root.maxsize(1000, 700) # Максимальный размер окна
```

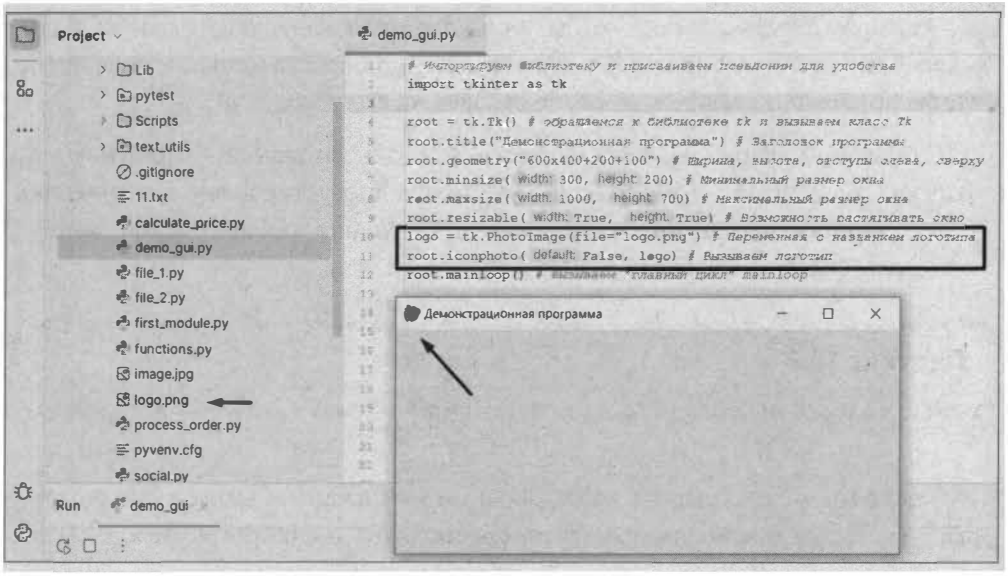
В верхнем левом углу программы вы можете заметить стандартный логотип, установленный Пайтоном по умолчанию. Чтобы его изменить, необходимо подобрать понравившееся изображение и поместить его в папку нашего проекта. Затем нужно подгрузить изображение в программу.

Листинг 19.8

```
logo = tk.PhotoImage(file="logo.png") # Переменная с названием
логотипа
root.iconphoto(False, logo) # Вызываем логотип
```

iconphoto — метод для установки изображения в качестве логотипа.

Первый параметр в скобках позволяет установить размер иконки, например 15. Если вставить вместо этого значение False, размер иконки будет подобран автоматически. Второй параметр — это переменная, в которой сохранен файл с картинкой.



Изображение 19.2.

Чтобы изменить цвет фона нашего приложения, необходимо указать соответствующий оттенок с помощью метода **config**.

Листинг 19.9

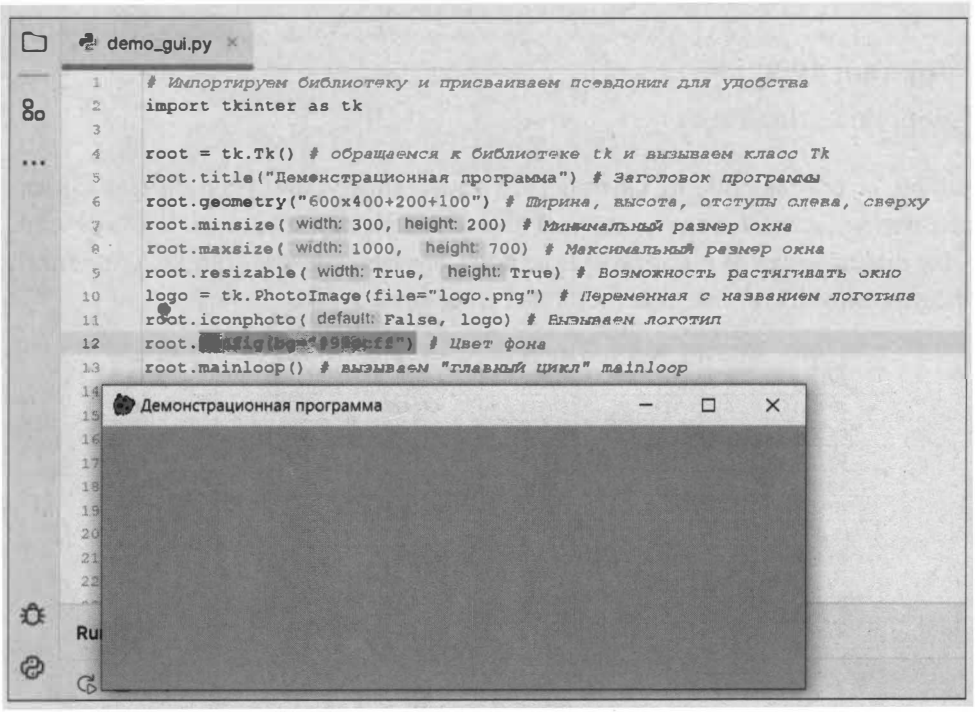
```
root.config(bg="#99ccff")
```

bg — сокращение от английского слова *background* (задний фон), в кавычках указывается цвет, который должен быть ему присвоен. Чтобы узнать код цвета, введите в поисковую строку браузера фразу "rgb коды цветов", результатом будут выданы таблички, подобные этой:

#FFFFFF	#CCFFFF	#99FFFF	#66FFFF	#33FFFF	#00FFFF
#FFFFCC	#CCFFCC	#99FFCC	#66FFCC	#33FFCC	#00FFCC
#FFFF99	#CCFF99	#99FF99	#66FF99	#33FF99	#00FF99
#FFFF66	#CCFF66	#99FF66	#66FF66	#33FF66	#00FF66
#FFFF33	#CCFF33	#99FF33	#66FF33	#33FF33	#00FF33
#FFFF00	#CCFF00	#99FF00	#66FF00	#33FF00	#00FF00
#FFCCFF	#CCCCFF	#99CCFF	#66CCFF	#33CCFF	#00CCFF
#FFCCCC	#CCCCCC	#99CCCC	#66CCCC	#33CCCC	#00CCCC
#FFCC99	#CCCC99	#99CC99	#66CC99	#33CC99	#00CC99
#FFCC66	#CCCC66	#99CC66	#66CC66	#33CC66	#00CC66
#FFCC33	#CCCC33	#99CC33	#66CC33	#33CC33	#00CC33
#FFCC00	#CCCC00	#99CC00	#66CC00	#33CC00	#00CC00
#FF99FF	#CC99FF	#9999FF	#6699FF	#3399FF	#0099FF
#FF99CC	#CC99CC	#9999CC	#6699CC	#3399CC	#0099CC
#FF9999	#CC9999	#999999	#669999	#339999	#009999
#FF9966	#CC9966	#999966	#669966	#339966	#009966
#FF9933	#CC9933	#999933	#669933	#339933	#009933
#FF9900	#CC9900	#999900	#669900	#339900	#009900
#FF66FF	#CC66FF	#9966FF	#6666FF	#3366FF	#0066FF
#FF66CC	#CC66CC	#9966CC	#6666CC	#3366CC	#0066CC
#FF6699	#CC6699	#996699	#666699	#336699	#006699
#FF6666	#CC6666	#996666	#666666	#336666	#006666
#FF6633	#CC6633	#996633	#666633	#336633	#006633
#FF6600	#CC6600	#996600	#666600	#336600	#006600
#FF33FF	#CC33FF	#9933FF	#6633FF	#3333FF	#0033FF
#FF33CC	#CC33CC	#9933CC	#6633CC	#3333CC	#0033CC
#FF3399	#CC3399	#993399	#663399	#333399	#003399
#FF3366	#CC3366	#993366	#663366	#333366	#003366
#FF3333	#CC3333	#993333	#663333	#333333	#003333
#FF3300	#CC3300	#993300	#663300	#333300	#003300
#FF00FF	#CC00FF	#9900FF	#6600FF	#3300FF	#0000FF
#FF00CC	#CC00CC	#9900CC	#6600CC	#3300CC	#0000CC
#FF0099	#CC0099	#990099	#660099	#330099	#000099
#FF0066	#CC0066	#990066	#660066	#330066	#000066
#FF0033	#CC0033	#990033	#660033	#330033	#000033
#FF0000	#CC0000	#990000	#660000	#330000	#000000

Изображение 19.3.

Выберите подходящий цвет и укажите его код в кавычках:



Изображение 19.4.

19.3. Виджеты

Виджеты, как упоминалось ранее, это отдельные элементы управления, которые используются для создания интерактивных интерфейсов. Они позволяют создавать элементы, с которыми пользователь может взаимодействовать.

Например, кнопка может запускать какую-то функцию, поле ввода позволяет получить данные от пользователя, а список — выбрать нужный вариант из предложенных.

Каждый виджет создается с помощью соответствующего класса, которому можно задать дополнительные свойства. Например, цвет текста, цвет фона, расположение и т.д.

Виджет **Label** — текстовая метка. Предназначен для отображения текста.

Листинг 19.10

```
label_greet = tk.Label(root, text="Привет, пользователь!")
label_greet.pack() # Размещаем метку в окне
```

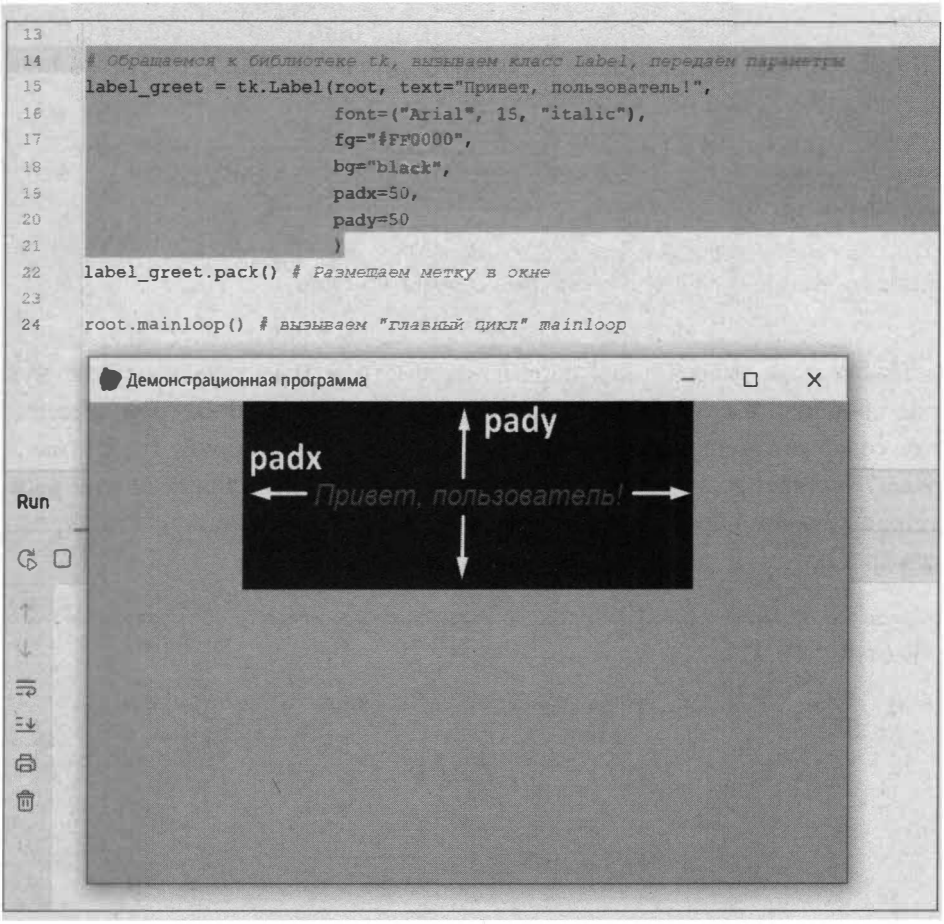
При вызове класса Label первым параметром мы указали переменную *root*, сообщив, в каком окне должен располагаться текст. Второй параметр **text** содержит непосредственно текст виджета. Во второй строке метод .pack() отвечает за вывод лейбла (упаковку). Также дополнительными параметрами можно передать цвет фона для текста, цвет самого текста, шрифт, размер и пр.

Листинг 19.11

```
label_greet = tk.Label(root, text="Привет, пользователь!",
                       font=("Arial", 15, "italic"),
                       fg="#FF0000",
                       bg="black",
                       padx=50,
                       pady=50
                       )
```

Вы можете заметить, что, помимо кода цветов, можно писать их название на английском языке (black).

- Параметр *fg* — это сокращение от английского *foreground* (передний план).
- Параметр *Arial* является названием шрифта, который вы хотели бы применить, число 15 — его размер, *italic* — наклонный текст (курсив). Вместо этого вы также можете указать жирный текст (bold) или вовсе не упоминать этот атрибут, тогда текст будет обычным.
- Параметры *padx* и *pady* отвечают за отступы текста от края рамки по координатам *x* и *y* соответственно.



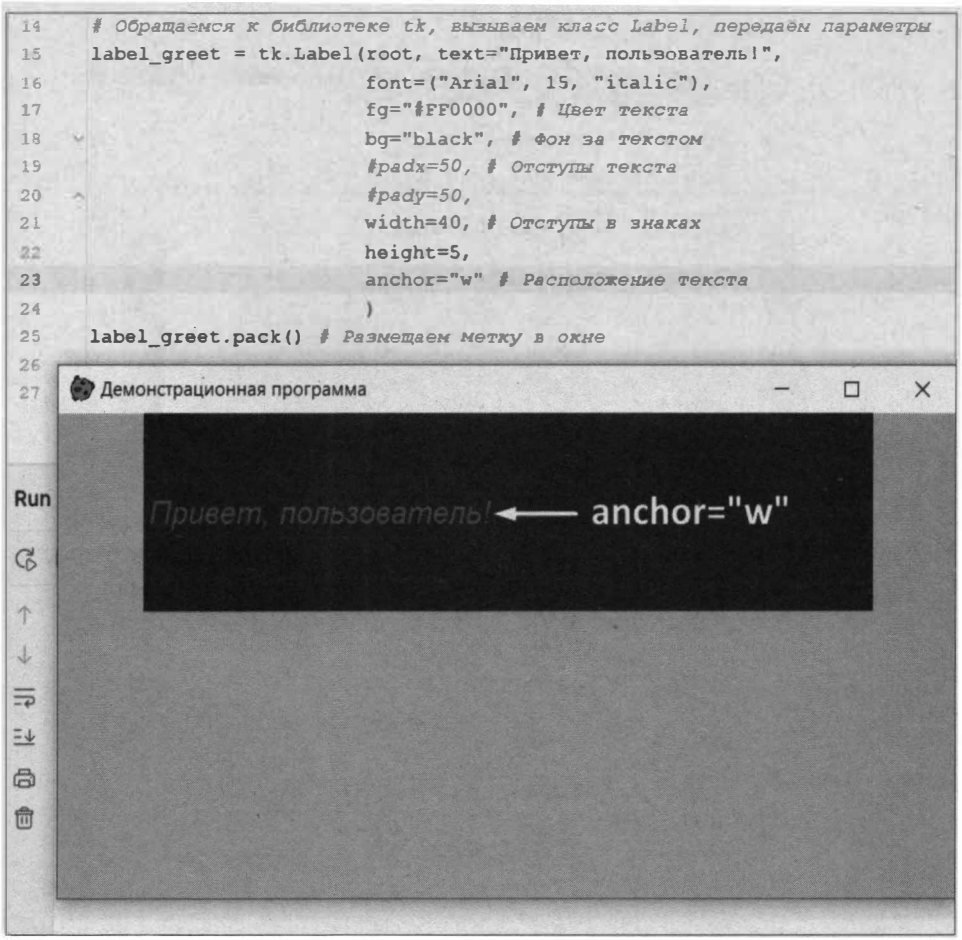
Изображение 19.5.

Вместо параметра *pad* можно установить отступы с помощью параметров *width=10* и *height=10*. В таком случае числа будут означать не пиксели, а количество знаков, которые можно расположить по ширине и высоте. Размер этих знаков зависит от размера шрифта в параметре *font*.

Чтобы изменить расположение текста внутри метки, существует параметр *anchor*, в котором можно указать сторону света: север, юг, запад, восток, прописав первую букву направления на английском языке.

Например, "n" для севера, "s" — для юга, "w" — для запада, "e" — для востока. Также можно написать сочетание двух сторон света, для расположения текста в одном из углов (ne, se, sw, nw, center).

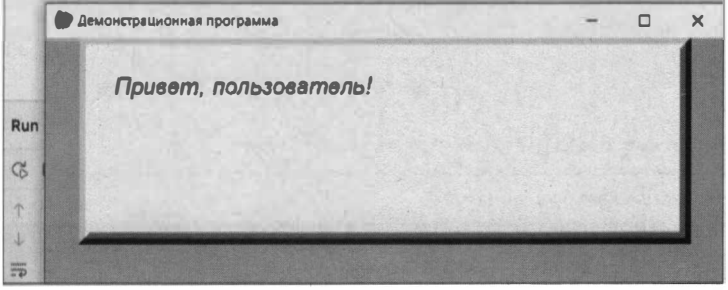
С помощью параметров *padx* и *pady* текст можно смещать на определенное расстояние.



Изображение 19.6.

- Параметр *relief=tk.RAISED* создает эффект выпуклости метки, оттеняя ее.
- Параметр *bd* — устанавливает ширину границы оттенения.

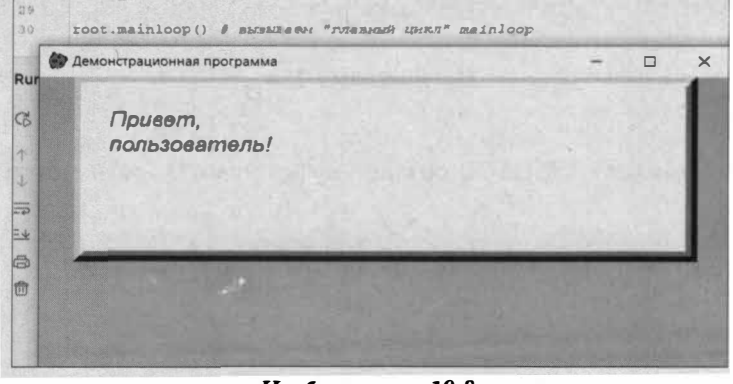
```
# Обращаемся к библиотеке tk, вызываем класс Label, передаем параметры
label_greet = tk.Label(root, text="Привет, пользователь!",
                       font=("Arial", 15, "italic"),
                       fg="#FF0000", # Цвет текста
                       bg="white", # Фон за текстом
                       padx=20, # Отступы текста
                       pady=20,
                       width=40, # Отступы в знаках
                       height=5,
                       anchor="nw", # Расположение текста
                       relief=tk.RAISED, # Эффект выпуклости
                       bd=10, # Ширина затенения
                       )
label_greet.pack() # Размещаем метку в окне

root.mainloop() # вызываем "главный цикл" mainloop
```



Изображение 19.7.

- Еще один полезный параметр — *justify=tk.LEFT* — позволяет выравнивать многострочный текст по левой или правой стороне (RIGHT).

```
# Обращаемся к библиотеке tk, вызываем класс Label, передаем параметры
label_greet = tk.Label(root, text="Привет, \nпользователь!",
                       font=("Arial", 15, "italic"),
                       fg="#FF0000", # Цвет текста
                       bg="white", # Фон за текстом
                       padx=20, # Отступы текста
                       pady=20,
                       width=40, # Отступы в знаках
                       height=5,
                       anchor="nw", # Расположение текста
                       relief=tk.RAISED, # Эффект выпуклости
                       bd=10, # Ширина затенения
                       justify=tk.LEFT
                       )
label_greet.pack() # Размещаем метку в окне

root.mainloop() # вызываем "главный цикл" mainloop
```



Изображение 19.8.

Виджет **Button** — создает кнопки, вызывая соответствующий класс. Используется для взаимодействия с программой и выполнения заданных событий.

Синтаксис кнопки аналогичен метке (Label):

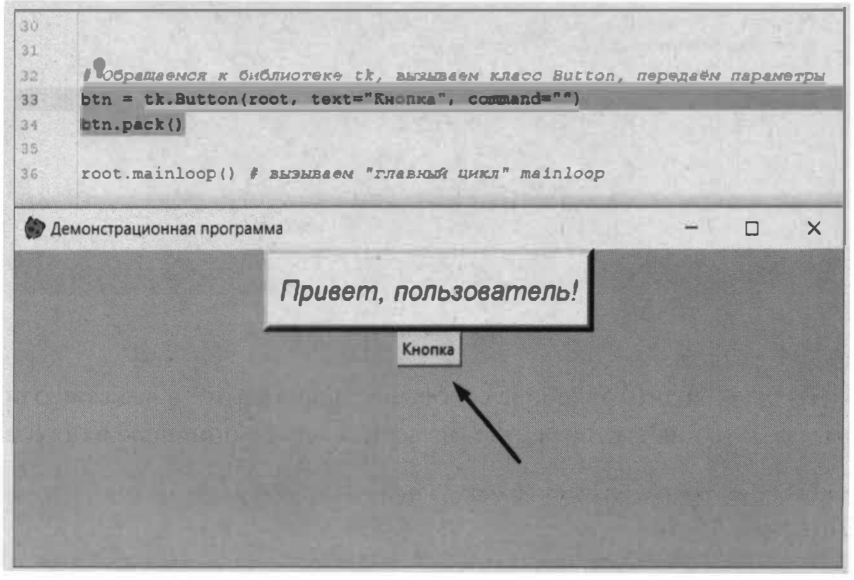
Листинг 19.12

```
btn = tk.Button(root, text="Кнопка", command=имя_функции)
btn.pack()
```

Также как с *label_greet* и *root*, мы можем задать любое удобное имя переменной, например *btn*. Далее мы обращаемся к классу Button из библиотеки tk (Tkinter) и задаем параметры.

Где первым мы также указываем окно (root), в котором будет находиться кнопка, затем текст на кнопке и параметр *command*=, который отвечает за действие после нажатия. В нем необходимо указать название функции, которая будет запущена после клика.

Не забываем прописывать метод btn.pack() для размещения кнопки в окне.



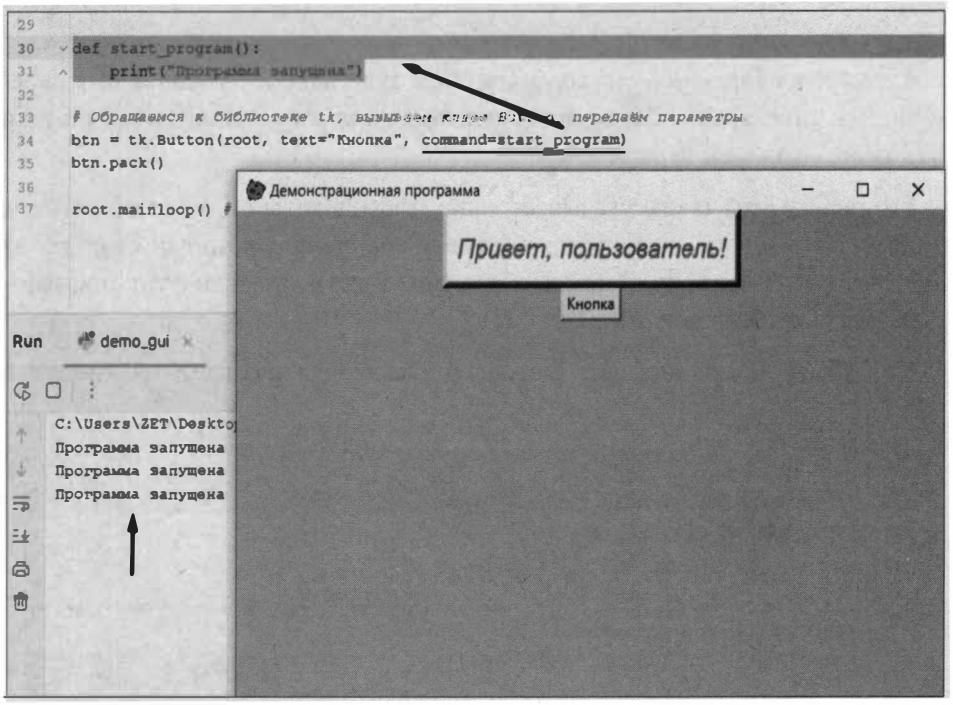
Изображение 19.9.

Давайте создадим простую функцию, которая будет выводить сообщение в консоль, чтобы проверить работоспособность кнопки.

Листинг 19.13

```
def start_program():
    print("Программа запущена")
```

После каждого нажатия на кнопку будет срабатывать заданное действие:



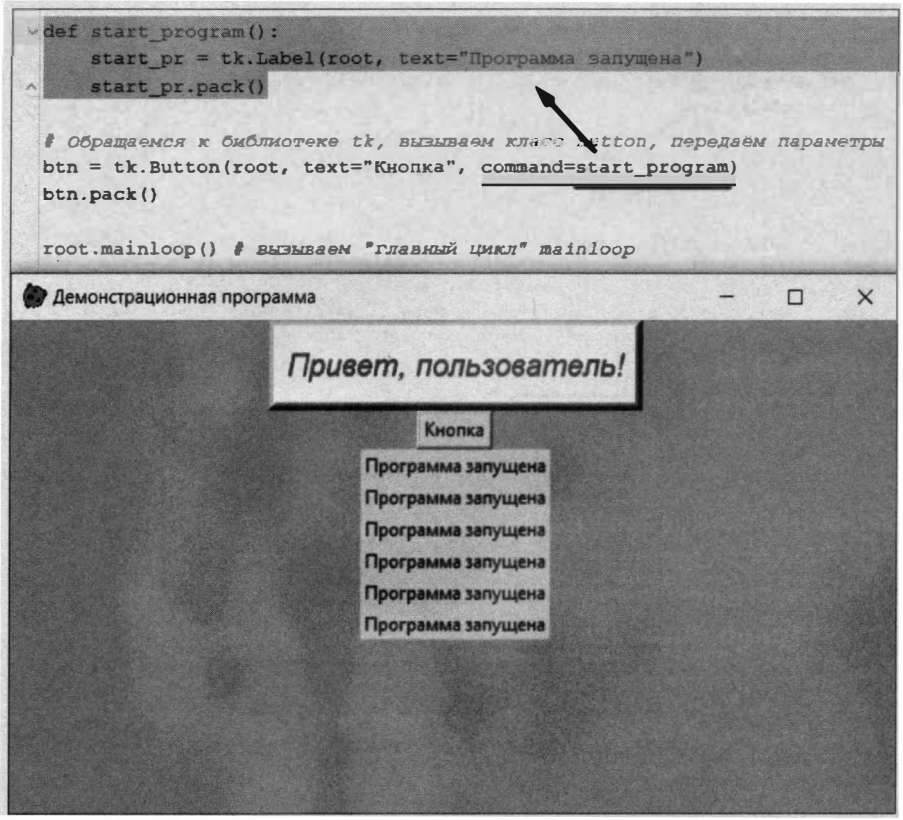
Изображение 19.10.

Чтобы выводить подобное сообщение в программе, а не в консоли, нужно создать функцию, которая будет выводить текст с помощью виджета Label:

Листинг 19.14

```
def start_program():
    start_pr = tk.Label(root, text="Программа запущена")
```

```
    start_pr.pack()
```

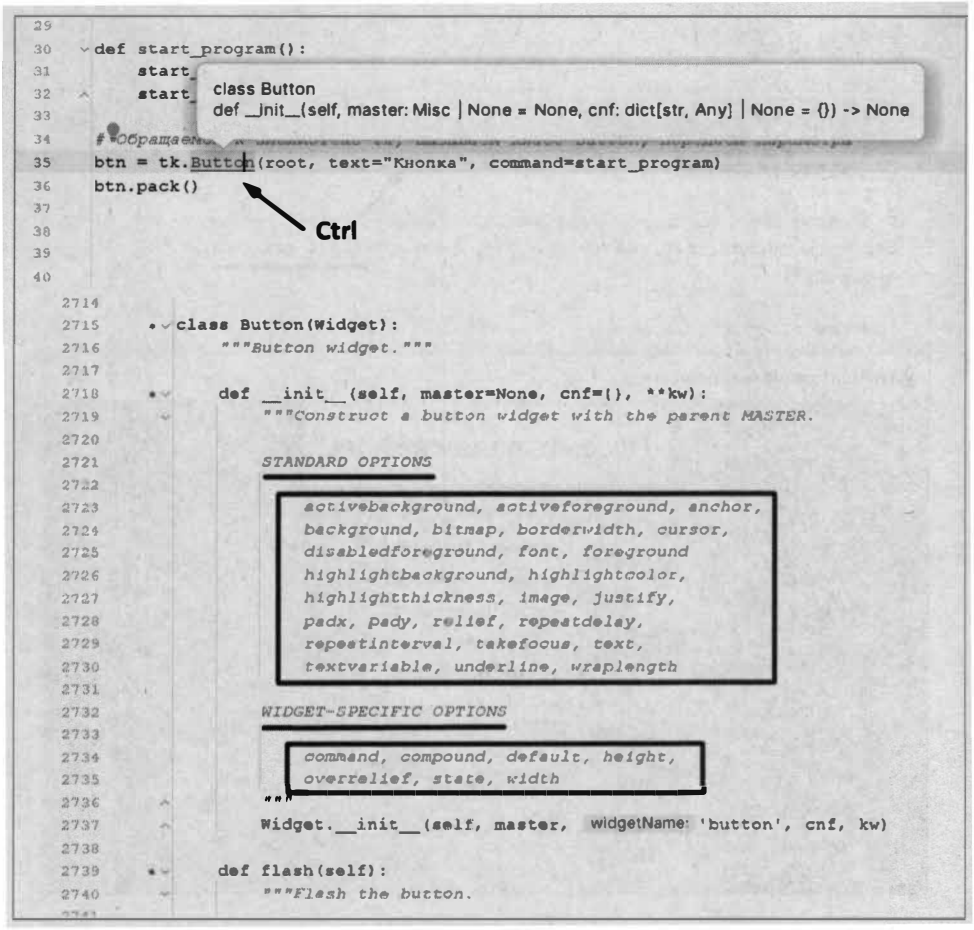


Изображение 19.11.

Внешний вид и расположение кнопки на экране настраиваются аналогично виджету Label.

Кроме того, вы можете навести курсор на класс Button и нажать клавишу Ctrl, после чего он станет ссылкой, при клике на которую откроется новый файл с этим классом и перечислением всех его свойств.

Здесь вы можете увидеть стандартные свойства для виджетов, а также особые свойства для кнопок. Таким же образом вы можете ознакомиться со свойствами любых других виджетов.

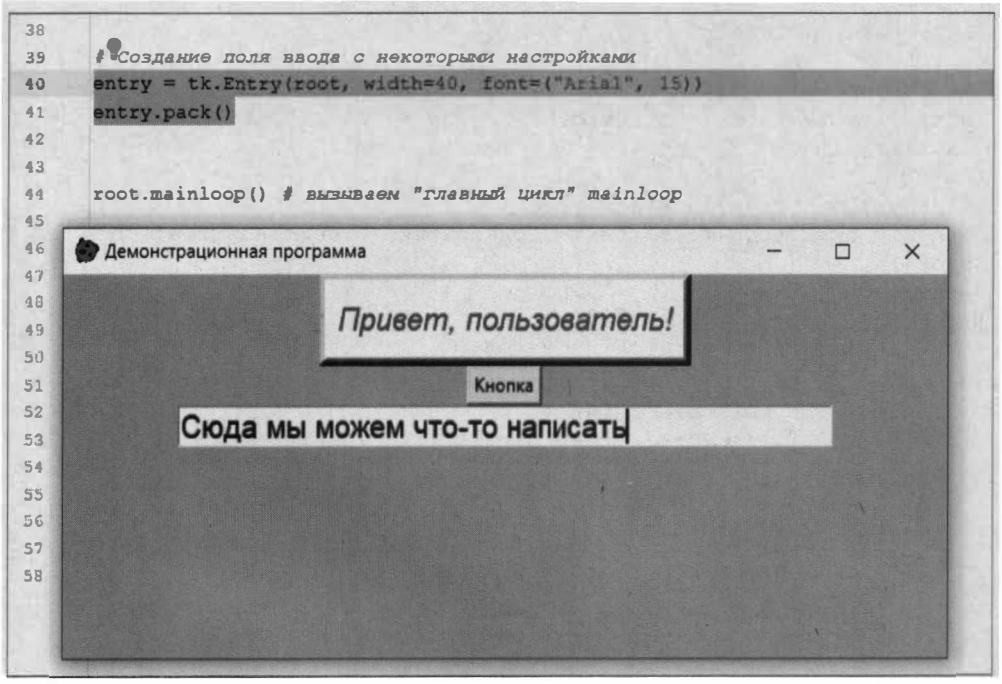


Изображение 19.12.

Виджет **Entry** — используется для создания однострочных текстовых полей, в которые пользователь может вводить текст.

Листинг 19.15

```
entry = tk.Entry(root, width=40, font=("Arial", 15))
entry.pack()
```



Изображение 19.13.

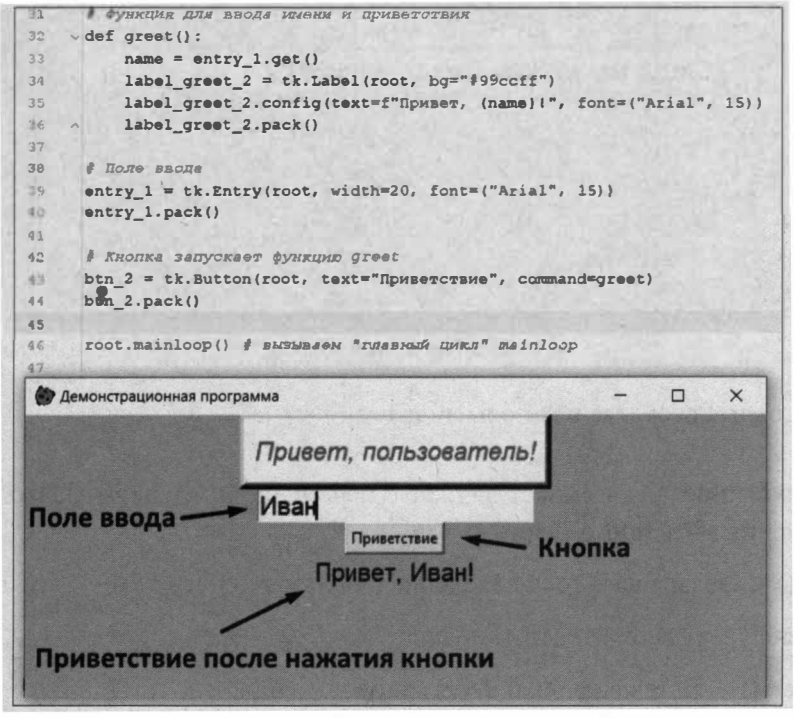
Основные свойства и методы виджета Entry:

- **textvariable** — связывает виджет с переменной, чтобы отслеживать изменения введенного текста.
- **get()** — возвращает текст, введенный пользователем.
- **delete()** — удаляет текст из поля.
- **insert()** — вставляет текст в указанную позицию.
- **config()** — используется для настройки различных свойств виджета, таких как цвет фона, шрифт, состояние (активное или неактивное).

Листинг 19.16

```
# функция для ввода имени и приветствия
def greet():
    name = entry_1.get()
    label_greet_2 = tk.Label(root, bg="#99ccff")
    label_greet_2.config(text=f"Привет, {name}!", font=("Arial", 15))
```

```
label_greet_2.pack()

# Поле ввода
entry_1 = tk.Entry(root, width=20, font=("Arial", 15)) entry_1.
pack()

# Кнопка запускает функцию greet
btn_2 = tk.Button(root, text="Приветствие", command=greet) btn_2.
pack()

root.mainloop()
```



Изображение 19.14.

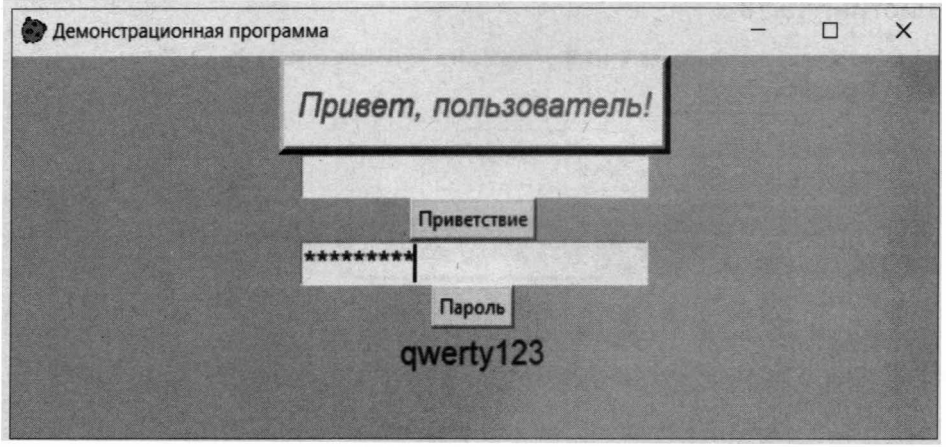
В примере мы создали функцию, которая будет возвращать текст (get) из поля ввода entry_1 и сохранять в переменной *name*. В строках *label_greet_2* будет создаваться метка с отображением приветствия и переменной *name*.

Далее мы создали поле ввода entry_1, информация из которого будет использоваться в функции ***greet***. Затем создаем кнопку, которая активирует функцию. В итоге после нажатия на экране появляется сообщение с приветствием пользователя.

Рассмотрим пример с сохранением введенного текста:

Листинг 19.17

```
password = None # Переменная с паролем — None по умолчанию

# Функция отображения пароля
def show_pas():
    password = entry_2.get()
    label_greet_3 = tk.Label(root, bg="#99ccff",
                             text=f"{password}",
                             font=("Arial", 15))
    label_greet_3.pack()

# Поле ввода пароля
entry_2 = tk.Entry(root, width=20, font=("Arial", 15),
                   show="*", # Звездочки вместо пароля
                   textvariable=password) # Сохраняем в переменную
entry_2.pack()

# Кнопка вызова функции
btn_2 = tk.Button(root, text="Пароль", command=show_pas)
btn_2.pack()

root.mainloop()
```



Изображение 19.15.

В примере выше функция ***show_pas*** выводит на экран содержимое переменной *password* аналогично тому, как мы это делали в примере с приветствием. По умолчанию пароль пустой, он перезаписывается при вводе текста

в поле *entry_2*. С помощью параметра *show* мы можем заменить вводимый текст на другие символы. Параметр *textvariable* сохраняет в указанную переменную (password) введенный текст.

Дополнительные параметры, которые можно использовать в поле ввода:

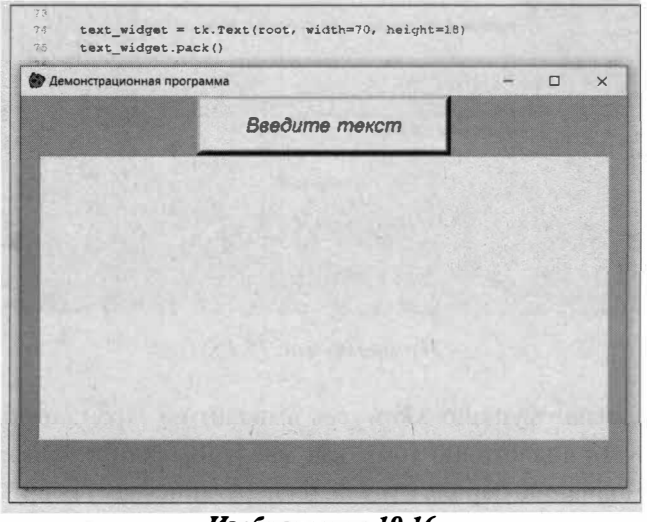
- *state* — может быть установлено в значение DISABLED (неактивен) или READONLY (только для чтения), чтобы отобразить поле ввода, но запретить ввод текста. Значение NORMAL установлено по умолчанию.
- *highlightcolor* — цвет рамки при фокусировке.
- *selectbackground*, *selectforeground* — цвета выделенного текста.

Полный перечень параметров также можно увидеть, наведя курсор на класс Entry и нажав Ctrl.

Виджет **Text** — предназначен для создания многострочных текстовых полей, он работает аналогично виджету Entry. Благодаря многострочному вводу с его помощью можно создавать текстовые редакторы, чаты, окна с логами и т.д.

Листинг 19.18

```
text_widget = tk.Text(root, width=70, height=18)
text_widget.pack()
```



Изображение 19.16.

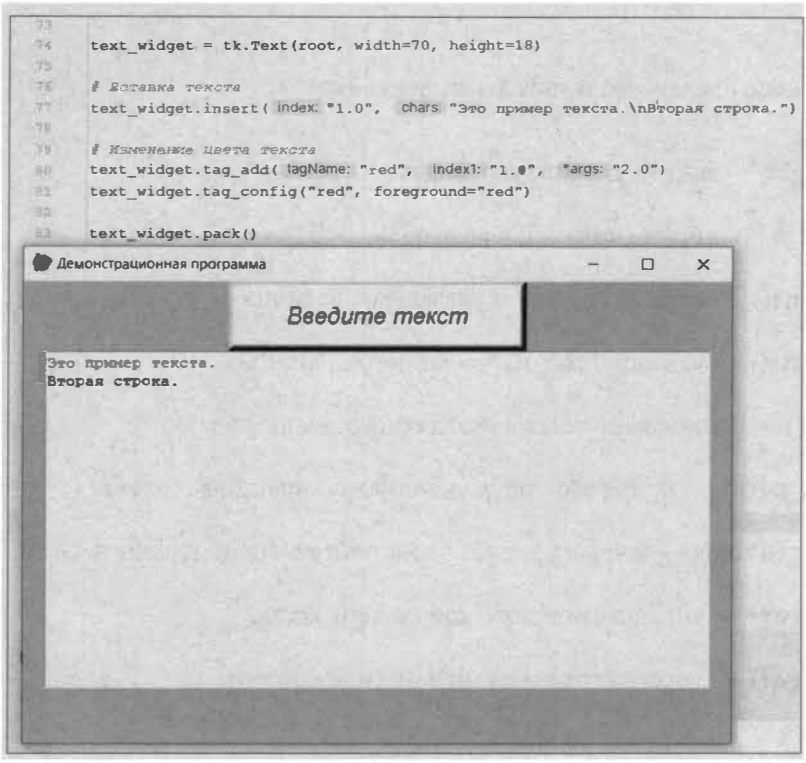
Основные свойства и методы виджета Text:

- **height** — высота виджета в строках.
- **width** — ширина виджета в символах.
- **insert()** — вставляет текст в указанную позицию.
- **delete()** — удаляет текст из указанного диапазона.
- **get()** — возвращает текст из указанного диапазона.
- **tag_add()** — применяет тег к указанному диапазону текста.
- **tag_config()** — конфигурирует свойства тега (цвет, шрифт и т.д.).
- **yview()** — управляет прокруткой по вертикали.
- **xview()** — управляет прокруткой по горизонтали.

Примеры использования:

Листинг 19.19

```
text_widget = tk.Text(root, width=70, height=18)

# Вставка текста
text_widget.insert("1.0", "Это пример текста.\nВторая строка.")

# Изменение цвета текста
text_widget.tag_add("red", "1.0", "2.0")
text_widget.tag_config("red", foreground="red")

text_widget.pack()
```



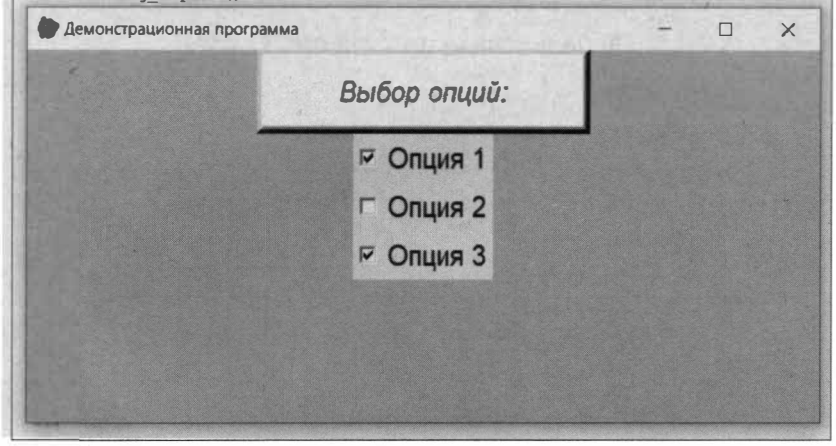
Изображение 19.17.

Виджет **Checkbutton (флажок)** — используется для создания элементов управления, которые позволяют пользователю выбирать один или несколько вариантов из предлагаемого списка. Каждый флажок имеет два состояния: отмечено (включено) или не отмечено (выключено).

Листинг 19.20

```
flag_1 = tk.Checkbutton(root, text="Опция 1", font=("Arial", 15))
flag_1.pack()

flag_2 = tk.Checkbutton(root, text="Опция 2", font=("Arial", 15))
flag_2.pack()

flag_3 = tk.Checkbutton(root, text="Опция 3", font=("Arial", 15))
flag_3.pack()
```

```
89
90  # Создание флажков с выбором опций
91  flag_1 = tk.Checkbutton(root, text="Опция 1", font=("Arial", 15))
92  flag_1.pack()
93
94  flag_2 = tk.Checkbutton(root, text="Опция 2", font=("Arial", 15))
95  flag_2.pack()
96
97  flag_3 = tk.Checkbutton(root, text="Опция 3", font=("Arial", 15))
98  flag_3.pack()
```



Изображение 19.18.

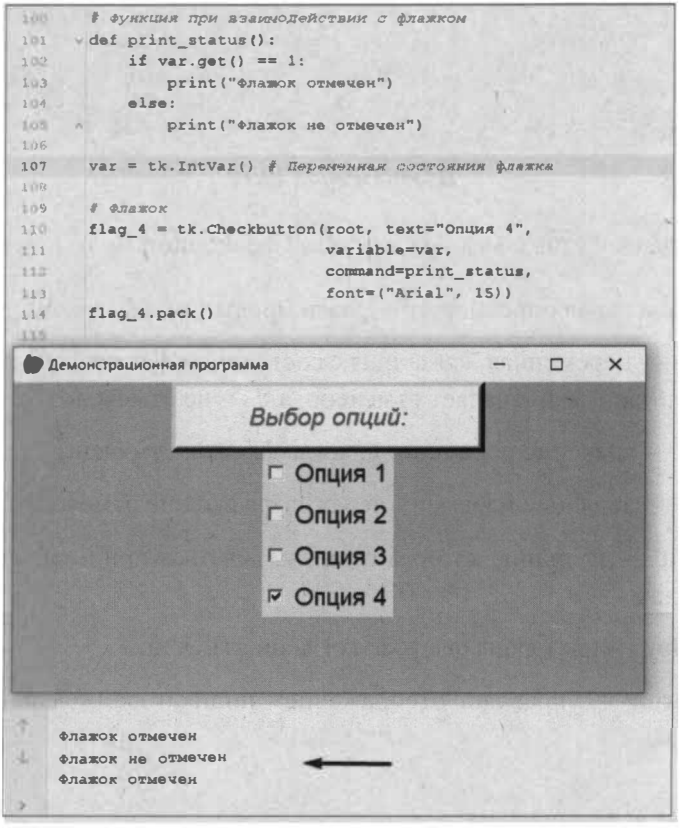
Основные свойства и методы виджета Checkbutton:

- **text** — текстовое описание, отображаемое рядом с флажком.
- **variable** — переменная, связанная с состоянием флажка. Обычно используется IntVar, где 1 означает отмечено, а 0 — не отмечено.
- **onvalue** — значение переменной, когда флажок отмечен.
- **offvalue** — значение переменной, когда флажок не отмечен.
- **command** — функция, которая будет вызываться при изменении состояния флажка.
- **selectcolor** — цвет фона флажка, когда он отмечен.
- **indicatoron** — управляет отображением индикатора (квадратика) рядом с текстом.

Примеры использования:

Листинг 19.21

```
# функция при взаимодействии с флажком
def print_status():
    if var.get() == 1:
        print("Флажок отмечен")
    else:
        print("Флажок не отмечен")

var = tk.IntVar() # Переменная состояния флажка

# Флажок
flag_4 = tk.Checkbutton(root, text="Опция 4",
                        variable=var,
                        command=print_status,
                        font=("Arial", 15))
flag_4.pack()
```



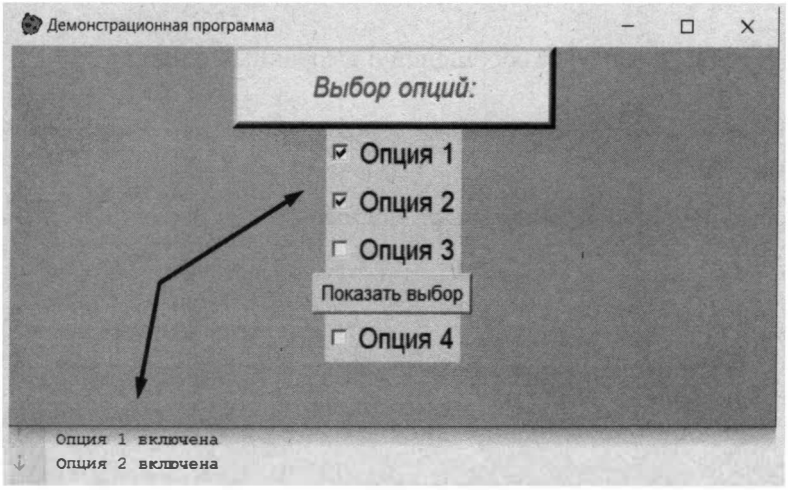
Изображение 19.19.

Еще пример, с выводом сообщений о выбранных опциях:

Листинг 19.22

```
# Функция сообщает о выбранных опциях
def print_selection():
    if var1.get() >0:
        print("Опция 1 включена")
    if var2.get() >0:
        print("Опция 2 включена")
    if var3.get() >0:
        print("Опция 3 включена")

# Переменные для трёх опций
var1 = tk.IntVar()
var2 = tk.IntVar()
var3 = tk.IntVar()

# Создание флажков с выбором опций
flag_1 = tk.Checkbutton(root, text="Опция 1",
                        font=("Arial", 15), variable=var1)
flag_1.pack()

flag_2 = tk.Checkbutton(root, text="Опция 2",
                        font=("Arial", 15), variable=var2)
flag_2.pack()

flag_3 = tk.Checkbutton(root, text="Опция 3",
                        font=("Arial", 15), variable=var3)
flag_3.pack()

# Кнопка вызывает функцию print_selection
button = tk.Button(root, text="Показать выбор",
                   font=("Arial", 11), command=print_selection)
button.pack()
```

В данном случае, вместо вывода сообщений мы можем запускать определенные функции, которые будут выполнять действия в зависимости от выбора пользователя.



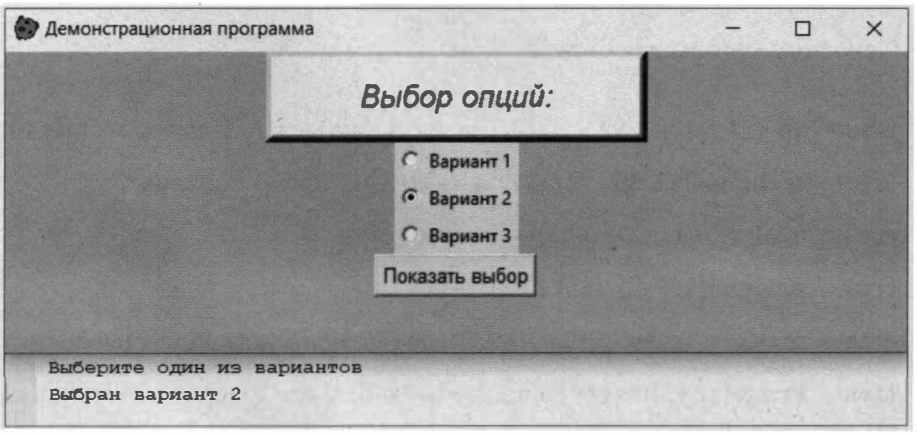
Изображение 19.20.

Виджет **Radiobutton (радиокнопка)** — используется для создания группы взаимоисключающих вариантов выбора. Когда пользователь выбирает одну радиокнопку в группе, другие автоматически отключаются. Виджет подходит для ситуаций, когда необходимо выбрать только один вариант из нескольких.

Листинг 19.23

```
# Функция отображения выбранного варианта
def print_choice():
    if var_5.get() =="1":
        print("Выбран вариант 1")
    elif var_5.get() =="2":
        print("Выбран вариант 2")
    elif var_5.get() =="3":
        print("Выбран вариант 3")
    else:
        print("Выберите один из вариантов")

# Создаём переменную для хранения выбранного значения
var_5 = tk.StringVar()

# Создаём группу радиокнопок, связав их с переменной var
radio1 = tk.Radiobutton(root, text="Вариант 1",
                        variable=var_5, value="1")
```

```
radio1.pack()

radio2 = tk.Radiobutton(root, text="Вариант 2",
                        variable=var_5, value="2")
radio2.pack()

radio3 = tk.Radiobutton(root, text="Вариант 3",
                        variable=var_5, value="3")
radio3.pack()

# Кнопка вызывает функцию print_choice
btn5 = tk.Button(root, text="Показать выбор",
                 font=("Arial", 10), command=print_choice)
btn5.pack()

root.mainloop()
```



Изображение 19.21.

Основные свойства и методы виджета Radiobutton:

- **text** — текстовая метка, отображаемая рядом с радиокнопкой.
- **variable** — переменная, связанная со всеми радиокнопками в группе. Когда пользователь выбирает кнопку, значение этой переменной меняется на значение, указанное в параметре *value* для выбранной кнопки.
- **value** — значение, которое будет присвоено переменной *variable*, когда кнопка будет выбрана.
- **command** — функция, которая будет вызываться при изменении состояния кнопки.

Стоит иметь в виду, что, если прописать функции ***command*** непосредственно в радиокнопки, они будут запускаться сразу, как только пользователь сделает выбор. Поэтому в примере выше запуск функции выведен на отдельную кнопку, чтобы в зависимости от выбора произошло одно из событий.

Однако в некоторых случаях может понадобиться прописать каждой радиокнопке функцию ***command***. Например, для вывода окна подтверждения, с номером или названием выбранной опции.

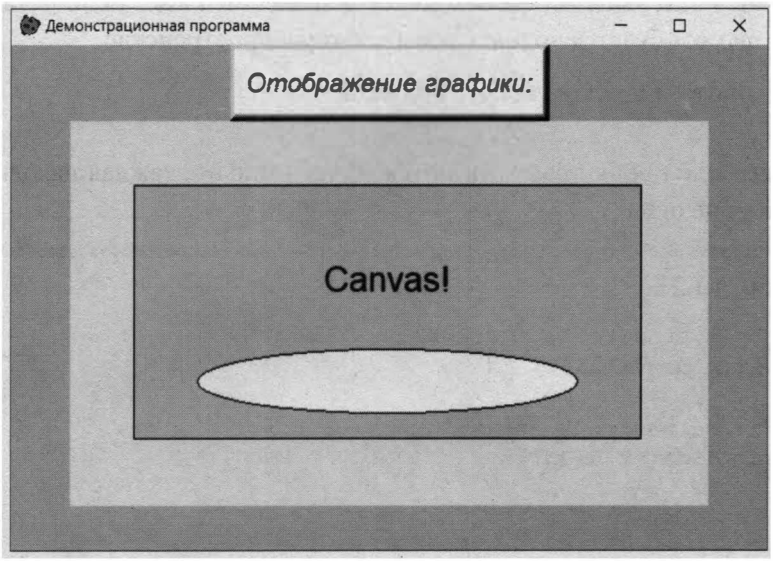
Виджет **Canvas** — предоставляет среду для отображения графических элементов. Он позволяет создавать линии, добавлять текст и изображения, а также управлять их размещением.

Основные методы для Canvas:

- **create_line(x1, y1, x2, y2, ...)** — создает линию между заданными точками.
- **create_rectangle(x1, y1, x2, y2, ...)** — создает прямоугольник.
- **create_oval(x1, y1, x2, y2, ...)** — создает овал.
- **create_polygon(x1, y1, x2, y2, ...)** — создает многоугольник.
- **create_text(x, y, text="", ...)** — добавляет текст в указанную точку.
- **create_image(x, y, image=None, ...)** — добавляет изображение в указанную точку.

Листинг 19.24

```
canvas = tk.Canvas(root, width=500, height=300) canvas.pack()

# Рисуем прямоугольник
canvas.create_rectangle(50, 50, 450, 250, fill="pink")
# Рисуем овал внутри прямоугольника
canvas.create_oval(100, 180, 400, 230, fill="white")
# Добавляем текст
canvas.create_text(250, 125, text="Canvas!", font=("Arial", 20))

root.mainloop()
```



Изображение 19.22.

19.4. Расположение виджетов

Для размещения виджетов в окне программы используются три основных менеджера геометрии: pack(), grid() и place(). Каждый из них имеет свои особенности и подходит для разных задач.

Метод **pack()** — виджеты упаковываются в контейнер, как коробки в ящик. Они занимают все доступное пространство, прижимаясь друг к другу.

Ранее, при знакомстве с виджетами, мы уже могли видеть его работу.

Метод имеет следующие параметры:

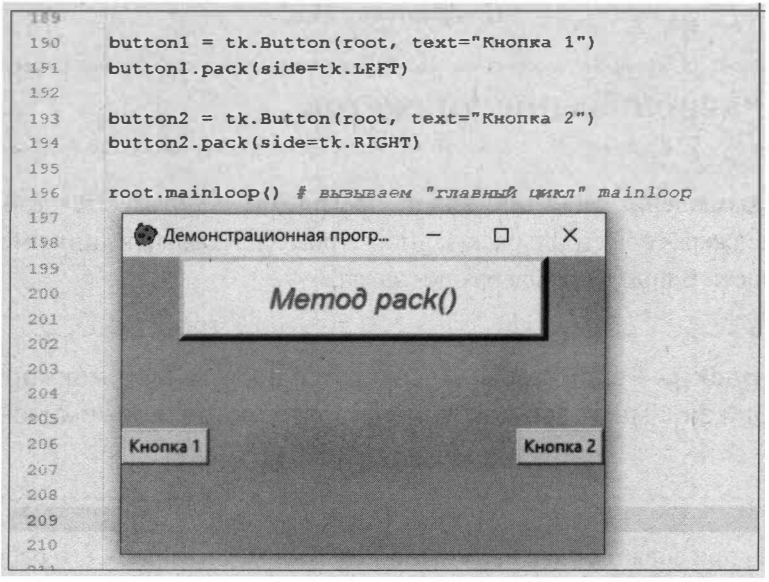
- **side** — указывает сторону, к которой будет смещен виджет (TOP, BOTTOM, LEFT, RIGHT — верх, низ, лево, право).
- **fill** — указывает, как виджет будет заполнять свободное пространство (X, Y, BOTH).

- **expand** — по умолчанию находится в значении False. Если установить True, виджет будет заполнять все доступное пространство.
- **padx, pady** — отступы вокруг виджета.

Пример, где кнопки располагаются горизонтально, каждая находится на своей стороне окна:

Листинг 19.25

```
btn1 = tk.Button(root, text="Кнопка 1")
btn1.pack(side=tk.LEFT)

btn2 = tk.Button(root, text="Кнопка 2")
btn2.pack(side=tk.RIGHT)

root.mainloop()
```



Изображение 19.23.

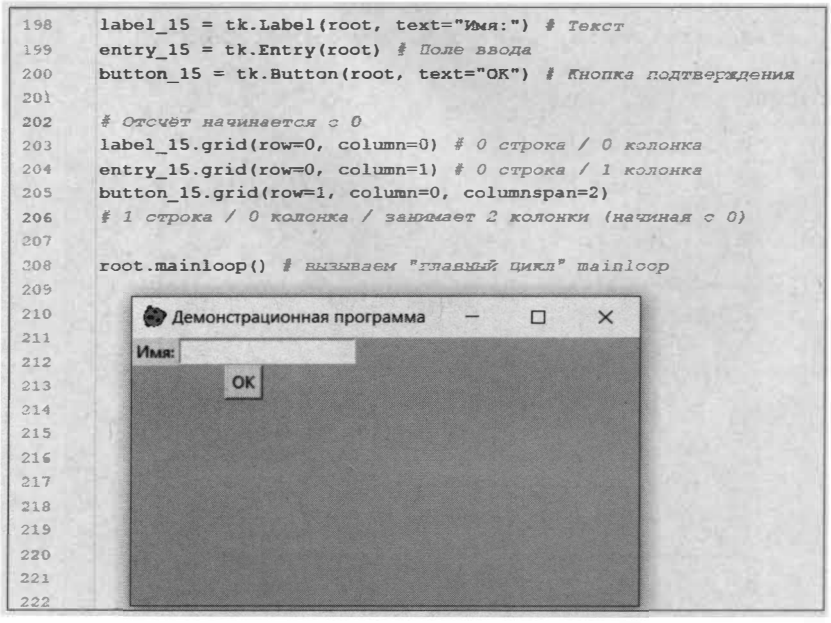
Метод **grid()** — создает таблицу с ячейками, в которых раскладываются виджеты. Ячейки нумеруются по строкам и столбцам, начиная с нулевой позиции.

Метод имеет следующие параметры:

- **row, column** — определяют строку и столбец, в которые помещается виджет.
- **columnspan, rowspan** — занимает несколько ячеек по горизонтали или вертикали.
- **sticky** — указывает, к какой стороне ячейки будет прижиматься виджет (N, S, E, W — по сторонам света).
- **padx, pady** — отступы вокруг виджета.

Листинг 19.26

```
lbl_25 = tk.Label(root, text="Введите имя:") # Текст подсказки
entry_25 = tk.Entry(root) # Поле для ввода текста
btn_25 = tk.Button(root, text="OK") # Кнопка подтверждения

# Отсчет начинается с 0
lbl_25.grid(row=0, column=0) # 0 строка / 0 колонка
entry_25.grid(row=0, column=1) # 0 строка / 1 колонка
btn_25.grid(row=1, column=0, columnspan=2)
# 1 строка / 0 колонка / занимает 2 колонки (начиная с 0)

root.mainloop()
```



Изображение 19.24.

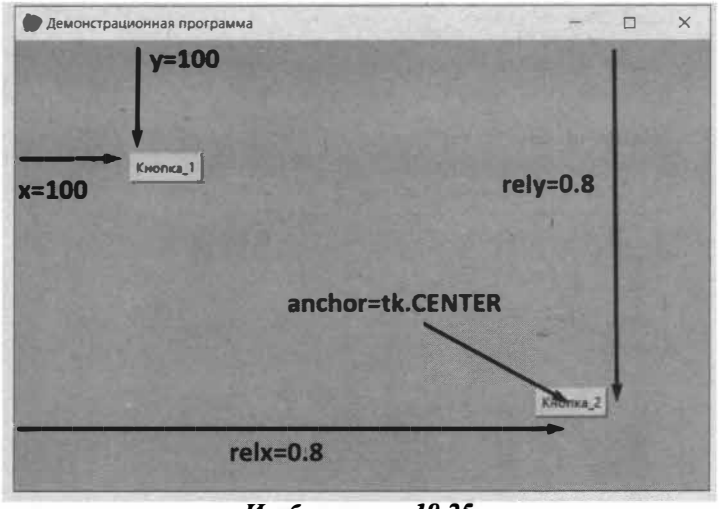
Метод **place()** — позволяет установить точное расположение виджета в пикселях в окне программы.

Параметры:

- **x, y** — координаты левого верхнего угла виджета.
- **width, height** — ширина и высота виджета.
- **anchor** — точка привязки виджета (N, S, E, W, CENTER).
- **relx, rely** — относительное положение виджета от 0 до 1.
  - » где relx=0.0 — это левый край окна, 0.5 — центр, 1.0 — правый край окна;
  - » аналогично rely: 0.0 — верхний край окна, 0.5 — центр, 1.0 — нижний край.

Листинг 19.27

```
# Точное размещение в пикселях
btn_26 = tk.Button(root, text="Кнопка_1")
btn_26.place(x=100, y=100)

# Относительное размещение
btn_27 = tk.Button(root, text="Кнопка_2")
btn_27.place(relx=0.8, rely=0.8, anchor=tk.CENTER)

root.mainloop()
```



Изображение 19.25.

В примере для первой кнопки отсчет ведется в пикселях от краев окна до левого верхнего угла кнопки. В примере для второй кнопки мы указали anchor=tk.CENTER — это центральная точка кнопки, до которой будет производиться отсчет от краев окна.

Часто при создании сложных интерфейсов комбинируют несколько методов. Например, можно использовать grid() для создания основной структуры, pack() или place() для точного позиционирования отдельных элементов внутри ячеек.

Важно помнить, что порядок вызова методов grid(), pack() и place() для одного виджета имеет значение. Каждый виджет может иметь только один геометрический менеджер. При использовании place() необходимо учитывать размеры родительского окна.

19.5. Расширенные возможности GUI

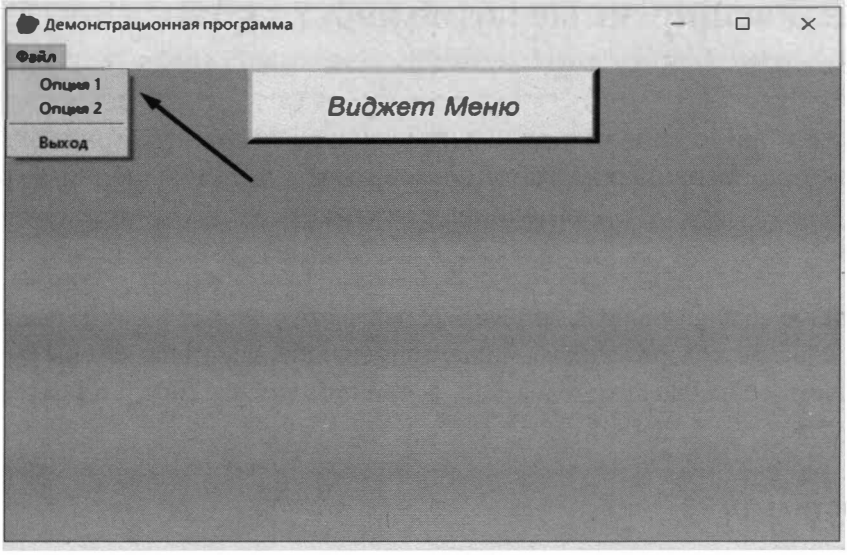
Виджет **Меню** является одним из ключевых элементов графического интерфейса. Он помогает пользователю найти и выполнить определенные команды.

Меню обычно располагается в верхней части окна и состоит из выпадающих списков, содержащих команды, подменю или действия, которые можно добавить с помощью методов **add_command**, **add_cascade, add_separator** и других.

Листинг 19.28

```
# Функции для пунктов меню
def set1():
    print("Включена настройка 1")

def set2():
    print("Включена настройка 2")

def exit():
    root.quit() # Закрываем приложение

# Создаём главное меню
```

```
main_menu = tk.Menu(root)

# Добавляем подменю "Файл"
file_menu = tk.Menu(main_menu, tearoff=0) # tearoff убирает пунктирную
линию
file_menu.add_command(label="Опция 1", command=set1) # Пункты меню
file_menu.add_command(label="Опция 2", command=set2)
file_menu.add_separator() # Разделитель
file_menu.add_command(label="Выход", command=exit) # Команда "Выход"

# Добавляем подменю "Файл" в главное меню
main_menu.add_cascade(label="Файл", menu=file_menu)

# Привязываем главное меню к окну
root.config(menu=main_menu)

root.mainloop()
```



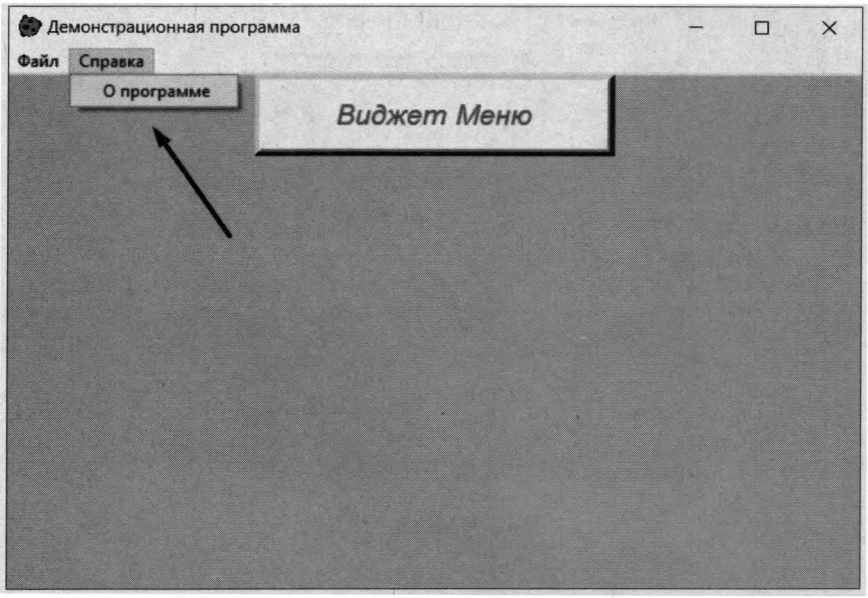
Изображение 19.26.

Аналогичным образом мы можем добавить еще несколько подменю для разных настроек:

Листинг 19.29

```
help_menu = tk.Menu(main_menu, tearoff=0)
help_menu.add_command(label="О программе",
                      command=lambda: print("Пример простой программы"))
```

```
# Добавляем раздел "Справка" в главное меню
main_menu.add_cascade(label="Справка", menu=help_menu)
```



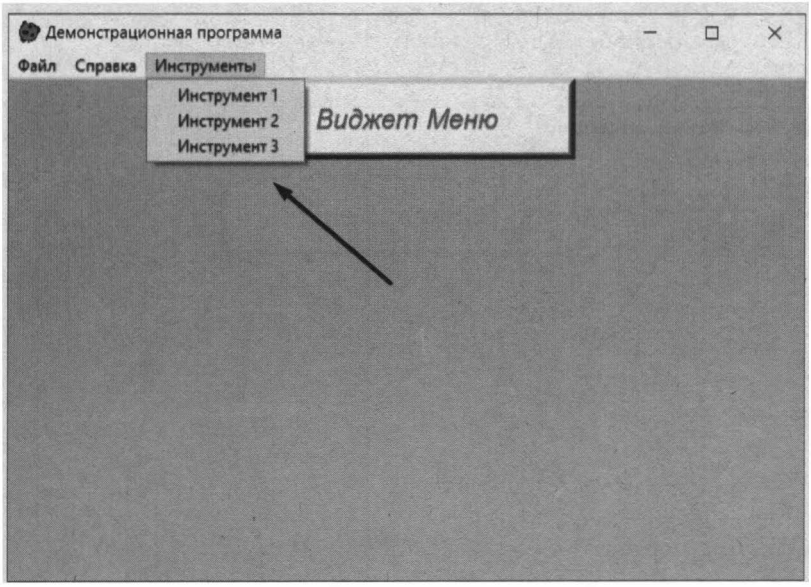
Изображение 19.27.

Обратите внимание, что благодаря однострочным функциям ***lambda*** мы можем писать простые функции непосредственно в параметрах метода.

Для добавления большого количества однотипных пунктов мы можем воспользоваться циклом:

Листинг 19.30

```
# Список подпунктов меню
tools = ["Инструмент 1", "Инструмент 2", "Инструмент 3"]

# Подменю "Инструменты"
tools_menu = tk.Menu(main_menu, tearoff=0)
for tool in tools:
    tools_menu.add_command(label=tool, command=lambda t=tool:
print(f"Вы выбрали {t}"))

main_menu.add_cascade(label="Инструменты", menu=tools_menu)
```



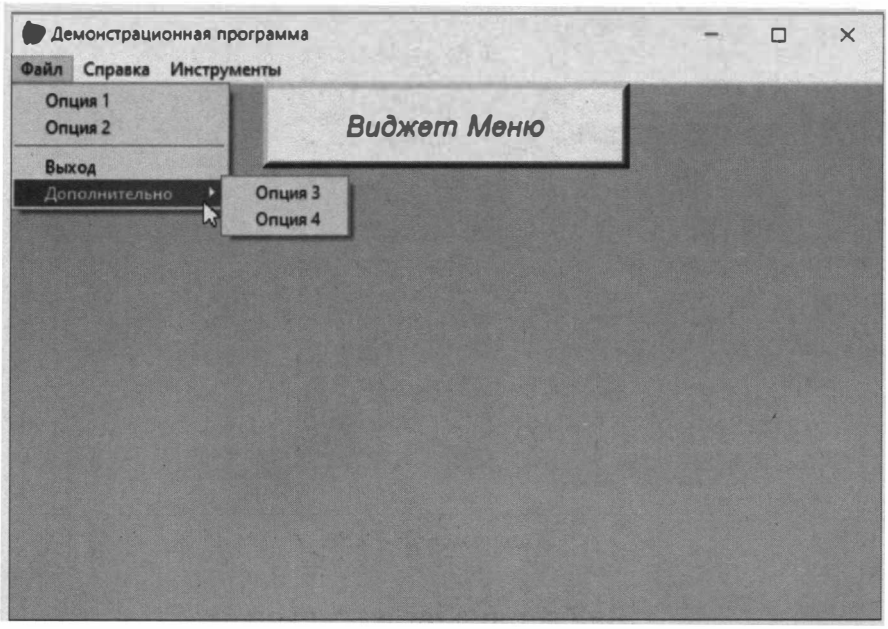
Изображение 19.28.

В итоге каждый пункт в разделе "Инструменты" будет вызывать свою команду. Вместо строк список *tools* может содержать переменные, которые хранят определенные функции, выполняющие разные действия.

Для реализации подменю внутри другого меню используется метод ***add_cascade***:

Листинг 19.31

```
# Создаем подменю "Дополнительно"
advanced_menu = tk.Menu(file_menu, tearoff=0)
advanced_menu.add_command(label="Опция 3",
                          command=lambda: print("Выбрана 3 опция"))
advanced_menu.add_command(label="Опция 4",
                          command=lambda: print("Выбрана 4 опция"))

# Добавляем подменю в раздел "Файл" file_menu.add_
cascade(label="Дополнительно", menu=advanced_menu)
```



Изображение 19.29.

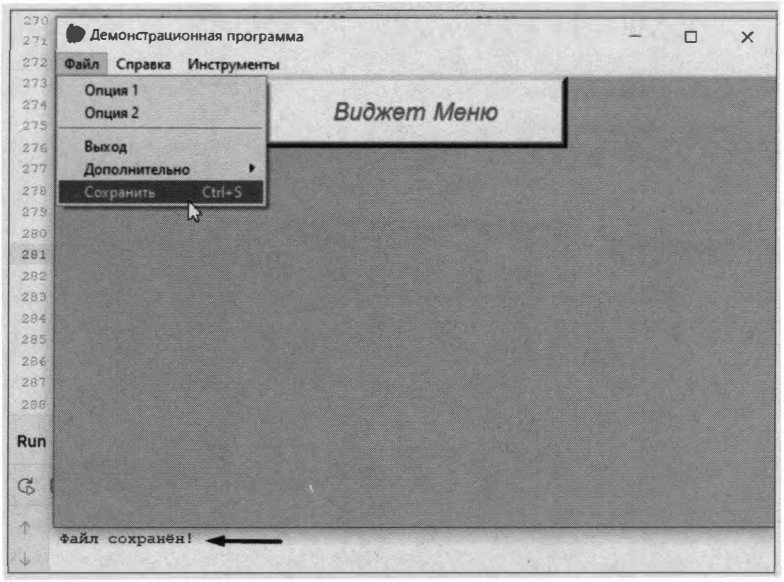
Можно добавить горячие клавиши к пунктам меню, используя параметр *accelerator*.

Но для его работы нужно настроить биндинг.

Листинг 19.32

```
def save_file(): print("Файл сохранен!")

# Привязка клавиши
root.bind("<Control-s>", lambda event: save_file())
file_menu.add_command(label="Сохранить", command=save_file,
accelerator="Ctrl+S")
```

Таким образом, используя комбинацию Ctrl+S, мы запустим функцию save_file для сохранения файла.



Изображение 19.30.

Виджет диалоговых окон

Виджет диалоговых окон, как понятно из названия, позволяет вызывать окошки с различными оповещениями, а также окна для взаимодействия с программой и файлами.

В библиотеке Tkinter такие диалоги реализуются с помощью модулей **tkinter.filedialog, tkinter.messagebox** и **tkinter.simpledialog.**

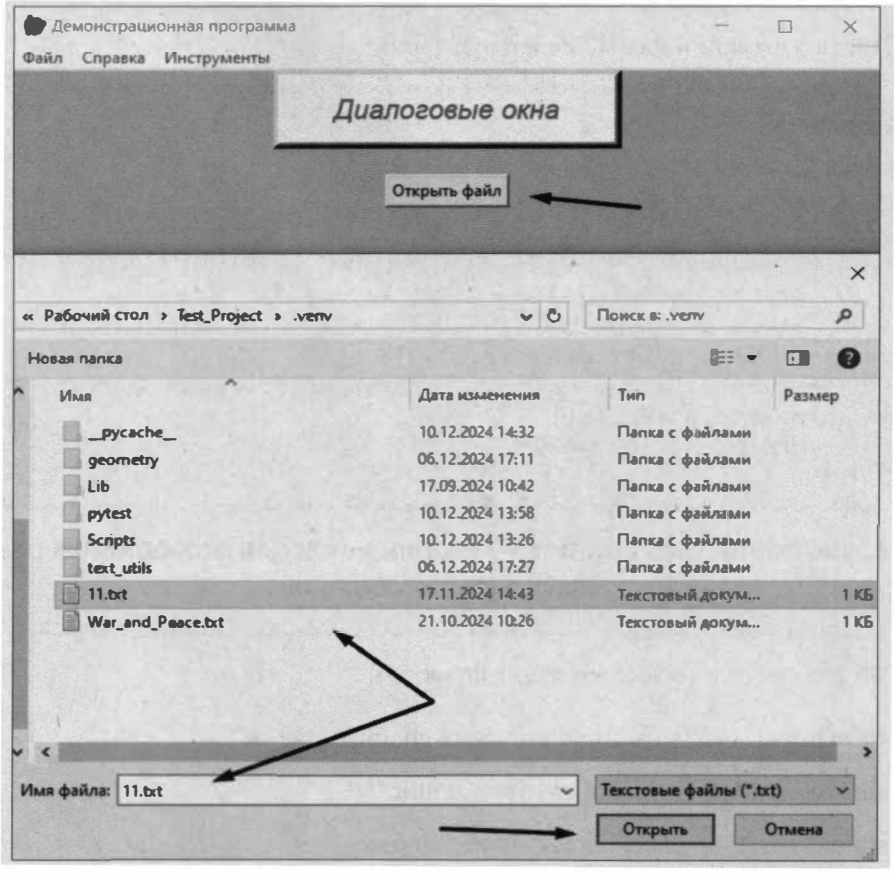
Модуль **tkinter.filedialog** предлагает функции для открытия и сохранения файлов, а также для выбора директорий.

Он включает в себя следующие функции:

- **askopenfilename()** — открывает диалог выбора файла для чтения.
- **asksaveasfilename()** — открывает диалог выбора файла для сохранения.
- **askdirectory()** — открывает диалог выбора директории.

Листинг 19.33

```
# Импортируем модуль filedialog
from tkinter import filedialog

# Функция для работы с файлами
def open_file():
    file_path = filedialog.askopenfilename(
        title="Выберите файл",
        filetypes=(("Текстовые файлы", "*.txt"), ("Все файлы", "*.*")))
    if file_path:
        print(f"Вы выбрали файл: {file_path}")

# Кнопка для вызова диалога
btn_open = tk.Button(root, text="Открыть файл", command=open_file)
btn_open.pack(pady=20)
```



Изображение 19.31.

Пример функции для сохранения файла:

Листинг 19.34

```
# Функция для сохранения файла
def save_file():
    file_path = filedialog.asksaveasfilename(
        title="Сохранить файл",
        defaultextension=".txt",
        filetypes=(("Текстовые файлы", "*.txt"), ("Все файлы", "*.*")))
    if file_path:
        print(f"Файл будет сохранён как: {file_path}")

# Кнопка сохранения файла
btn_save = tk.Button(root, text="Сохранить файл", command=save_file)
btn_save.pack(pady=20)
```

Пример функции для выбора директории:

Листинг 19.35

```
# Функция выбора директории
def select_directory():
    directory = filedialog.askdirectory(title="Выберите папку")
    if directory:
        print(f"Выбрана папка: {directory}")

# Кнопка выбора директории
btn_directory = tk.Button(root, text="Выбрать директорию",
command=select_directory)
btn_directory.pack(pady=20)
```

Модуль **tkinter.messagebox** — используется для отображения различных типов сообщений.

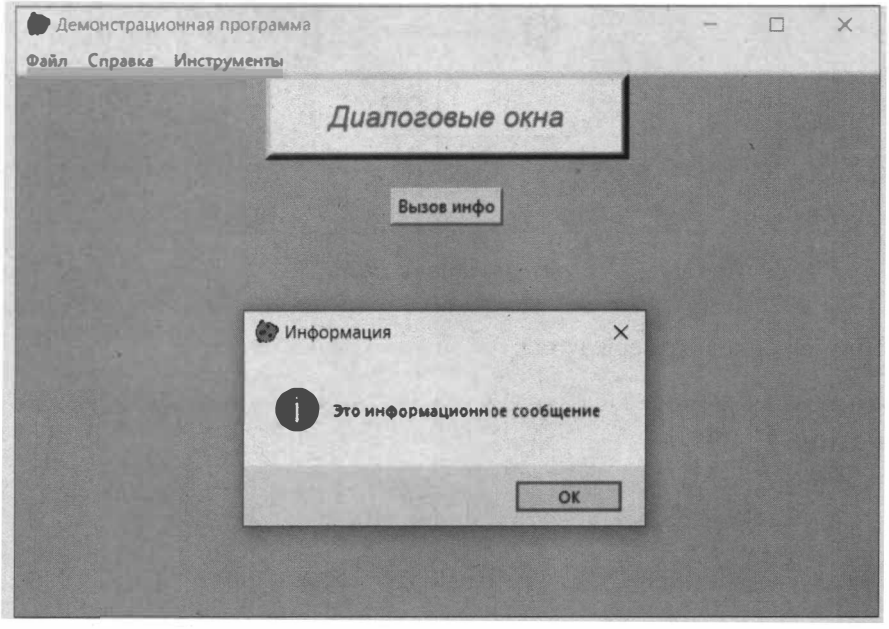
Он включает в себя следующие функции:

- **showinfo()** — для информационных сообщений.
- **showwarning()** — для предупреждений.
- **showerror()** — для сообщений об ошибках.
- **askquestion()** — для вопросов с ответами "Yes" или "No".

- **askokcancel(), askyesno(), askretrycancel()** — для диалогов подтверждения.

Листинг 19.36

```
# Импортируем модуль messagebox
from tkinter import messagebox

# Функция отображения информационного окна
def show_info():
    messagebox.showinfo("Информация", "Это информационное
сообщение")

# Кнопка для демонстрации
btn_win = tk.Button(root, text="Вызов инфо", command=show_info)
btn_win.pack(pady=20)

root.mainloop()
```

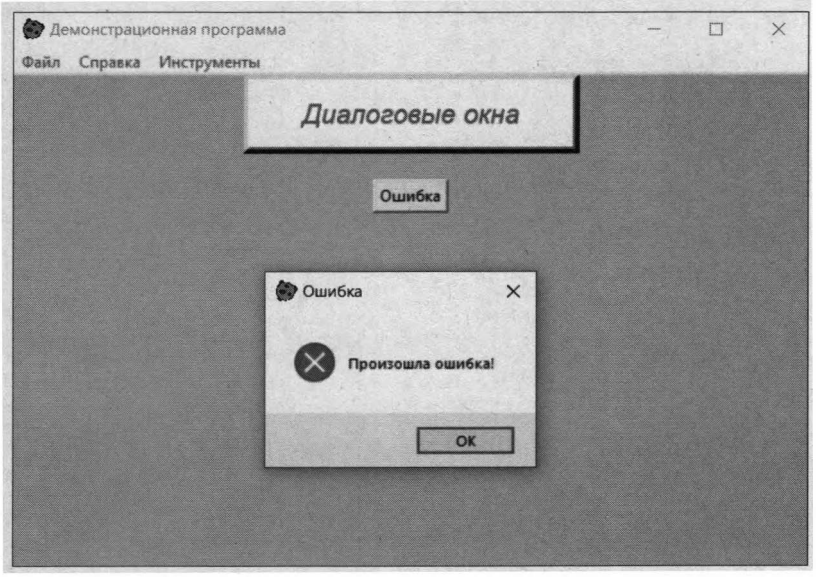


Изображение 19.32.

Пример сообщения об ошибке:

Листинг 19.37

```
def show_error():
    messagebox.showerror("Ошибка", "Произошла ошибка!")

btn_error = tk.Button(root, text="Ошибка", command=show_error)
btn_error.pack(pady=20)
```

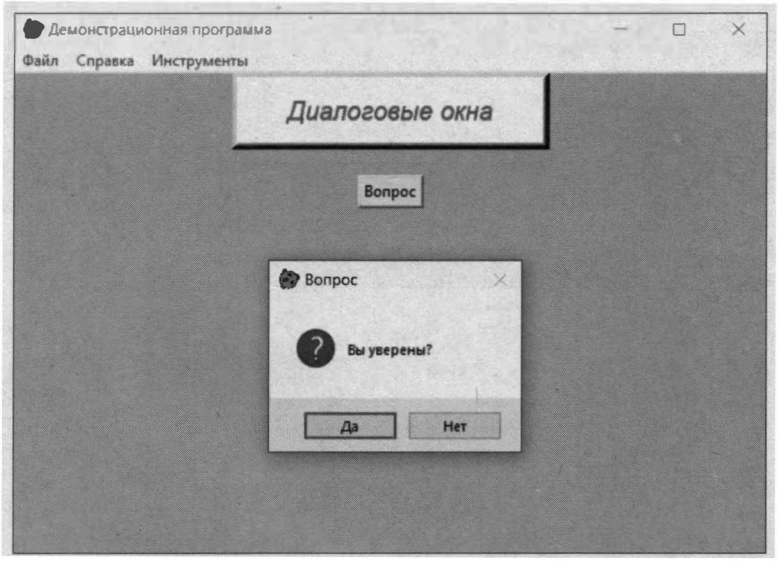


Изображение 19.33.

Пример окна подтверждения:

Листинг 19.38

```
def ask_user():
    response = messagebox.askyesno("Вопрос", "Вы уверены?")
    if response:
        print("Пользователь ответил: Да")
    else:
        print("Пользователь ответил: Нет")

btn_ask_user = tk.Button(root, text="Вопрос", command=ask_
user)
btn_ask_user.pack(pady=20)
```



Изображение 19.34.

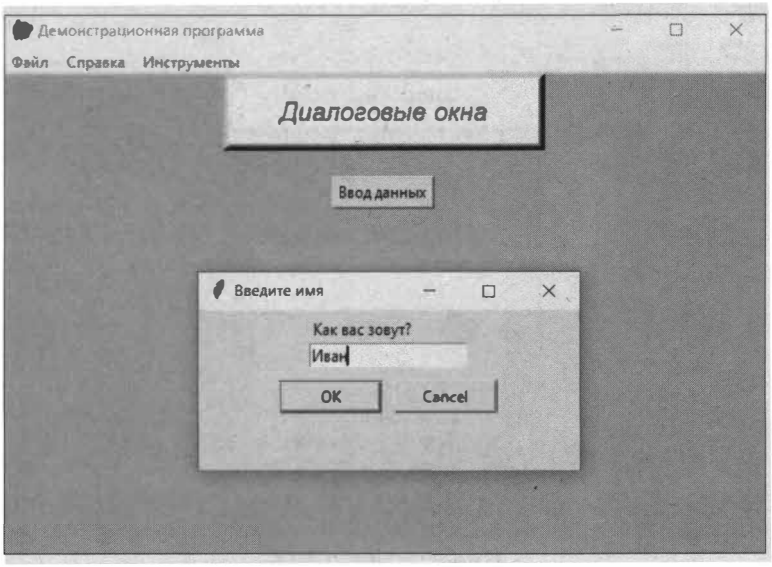
Модуль **tkinter.simpledialog** — используется для запроса данных у пользователя.

Он включает в себя следующие функции:

- **askstring()** — запрашивает строку.
- **askinteger()** — запрашивает целое число.
- **askfloat()** — запрашивает число с плавающей точкой.

Листинг 19.39

```
# Импортируем модуль simpledialog
from tkinter import simpledialog

# Функция вызывает окно для ввода текста
def ask_name():
    name = simpledialog.askstring("Введите имя",
                                  "Как вас зовут?")
    if name:
        print(f"Ваше имя: {name}")

# Кнопка для вызова функции
btn_ask_name = tk.Button(root, text="Ввод данных", command=ask_name)
btn_ask_name.pack(pady=20)
```



Изображение 19.35.

Пример запроса числа:

Листинг 19.40

```
def ask_age():
    age = simpledialog.askinteger("Введите возраст", "Сколько
вам лет?")
    if age:
        print(f"Ваш возраст: {age}")

btn_age = tk.Button(root, text="Ввод данных", command=ask_age)
btn_age.pack(pady=20)
```

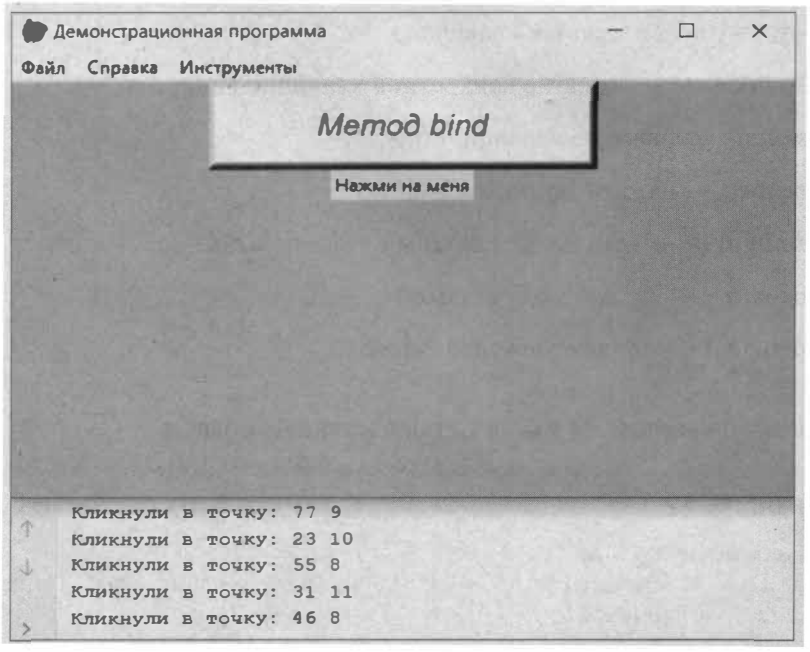
19.6. Обработка событий методом *bind*

Обработка событий — это механизм, благодаря которому программа может реагировать на определенные действия и выполнять определенные функции. Например, когда пользователь нажимает на кнопку, вызывается функция, которая выполняет какое-то действие.

Ранее, во время знакомства с виджетом Button, мы узнали, что за обработку событий при клике на кнопку отвечает параметр *command*. В нем указывается функция, которая будет запущена. Помимо кнопок, активацию событий можно привязать ко многим другим действиям. Для этого используется метод ***bind***.

Листинг 19.41

```
# Функция отображения координаты клика
def prs_click(event):
    print("Кликнули в точку:", event.x, event.y)

# Текстовая метка
lbl_33 = tk.Label(root, text="Нажми на меня")

# Нажимаем ЛКМ (Button-1), срабатывает функция prs_click
lbl_33.bind("<Button-1>", prs_click)
lbl_33.pack()

root.mainloop()
```



Изображение 19.36.

В этом примере при клике левой кнопкой мыши по метке будет вызвана функция ***prs_click***, а в консоли отобразятся координаты клика по *x* и *y*. В атрибут функции передается строка, описывающая тип события (например, <Button-1>). И далее выводятся значения *event.x*, *event.y*.

То есть, даже не имея кнопки в интерфейсе, мы можем запустить событие, если пользователь нажмет в определенную область или определенную клавишу или просто поведет мышью.

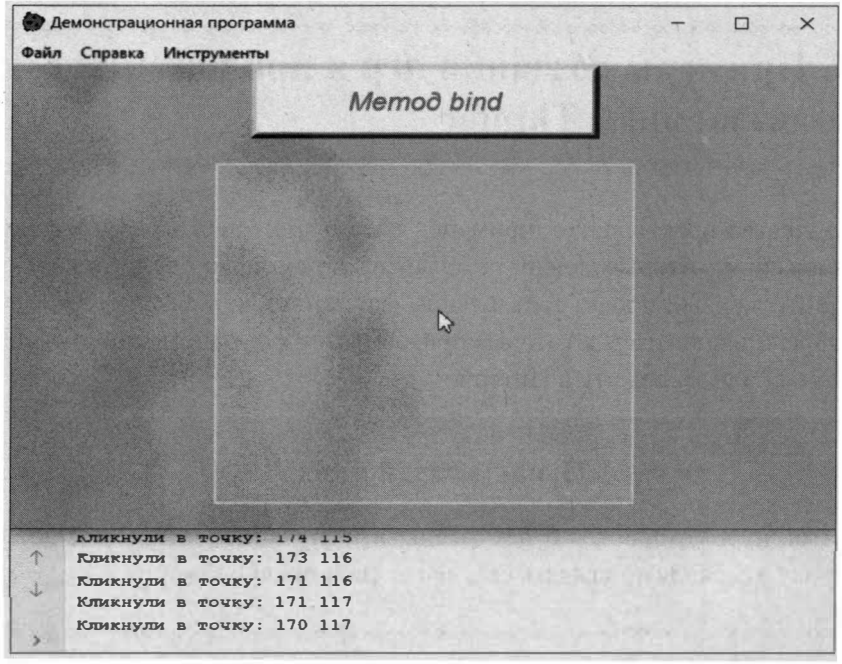
Tkinter поддерживает множество различных событий:

- **<Button-1>** — клик левой кнопкой мыши.
- **<Button-2>** — клик средней кнопкой мыши (колесико).
- **<Button-3>** — клик правой кнопкой мыши.
- **<Motion>** — перемещение мыши.
- **<Leave>** — выход мыши из указанной области.
- **<Double-Button-1>** — двойной клик левой кнопкой мыши.
- **<Key>** — любая нажатая клавиша.
- **<KeyPress-z>** — нажатие конкретной клавиши, например "z".
- **<Return>** — нажатие клавиши Enter.
- **<Escape>** — нажатие клавиши Esc.
- **<Configure>** — изменение ширины и высоты окна.
- **<FocusIn>** — виджет получил фокус.
- **<FocusOut>** — виджет потерял фокус.

Пример с наведением мыши на определенную область:

Листинг 19.42

```
# Создаем поле canvas
canvas_22 = tk.Canvas(root, width=300, height=300, bg="lightblue")
canvas_22.pack(pady=20)

# Привязываем обработчик события клика к холсту
canvas_22.bind("<Motion>", prs_click)
```



Изображение 19.37.

Для удаления события используется метод ***unbind***.

Он может понадобиться, чтобы отключить определенное действие. Например, с его помощью мы можем отключить предыдущее действие canvas_22.bind("<Motion>", on_click):

Листинг 19.43

```
# Кнопка для отключения обработчика
btn_44 = tk.Button(root, text="Отключить",
command=lambda: canvas_22.unbind("<Motion>")) btn_44.pack()
```

События можно комбинировать с Ctrl, Alt и Shift, прописав соответствующие комбинации в треугольных скобочках:

```
<Control-c> – Ctrl + S.

<Shift-Up> – Shift + Стрелка вверх.

<Alt-F3> – Alt + F3.
```

19.7. Примеры создания игр и приложений с использованием Tkinter

В качестве практических примеров рассмотрим несколько простых игр и приложений, которые можно реализовать с помощью библиотеки Tkinter. Напишите каждый проект в отдельном файле, чтобы в следующем разделе мы собрали из некоторых готовые программы, которыми можно поделиться с друзьями или выложить в Интернет.

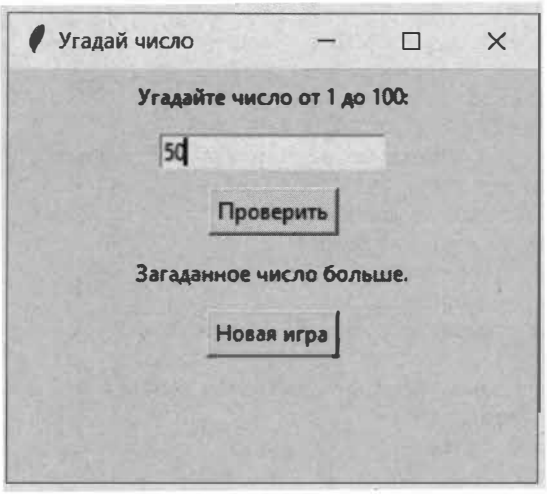
Игра "Угадай число"

В этой игре компьютер будет загадывать случайное число от 1 до 100, а игроку необходимо угадать его, вводя свои предположения.

Листинг 19.44

```
import tkinter as tk
from tkinter import messagebox
import random

def check_guess(): # Функция проверки введённого числа
    try:
        guess = int(entry.get()) # Получаем число из поля ввода
        if guess < 1 or guess > 100:
            messagebox.showerror("Ошибка",
             "Пожалуйста, введите число от 1 до 100.")
            return

        global attempts
        attempts += 1 # Считаем количество попыток

        if guess == secret_number: # Если число совпало
            messagebox.showinfo("Победа!",
                f"Вы угадали число за {attempts} попыток!")
            new_game()
        elif guess < secret_number:
            label_result.config(text="Загаданное число больше.")
        else:
            label_result.config(text="Загаданное число меньше.")
```

```
        entry.delete(0, tk.END) # Очищаем поле ввода

    except ValueError:
        messagebox.showerror("Ошибка",
                             "Пожалуйста, введите целое число.")

def new_game(): # Функция новой игры
    global secret_number, attempts
    # Загадываем новое число
    secret_number = random.randint(1, 100)
    # Сбрасываем счётчик попыток
    attempts = 0
    # Очищаем текстовое поле результата
    label_result.config(text="")
    # Очищаем поле ввода
    entry.delete(0, tk.END)

# Создание окна
root = tk.Tk()
root.title("Угадай число")

# Первая генерация числа
secret_number = random.randint(1, 100)
attempts = 0 # Счётчик попыток

# Текстовая метка с инструкцией
label_instruction = tk.Label(root, text="Угадайте число от 1 до 100:")
label_instruction.pack(pady=5)

# Поле ввода для числа
entry = tk.Entry(root)
entry.pack(pady=5)

# Кнопка вызывает функцию check_guess
check_button = tk.Button(root, text="Проверить", command=check_guess)
check_button.pack(pady=5)

# Метка для отображения подсказок
label_result = tk.Label(root, text="")
label_result.pack(pady=5)

# Кнопка новой игры
new_game_button = tk.Button(root, text="Новая игра", command=new_game)
new_game_button.pack(pady=5)

new_game() # Начинаем новую игру при запуске
root.mainloop()
```



Изображение 19.38.

Игра "Pac Man"

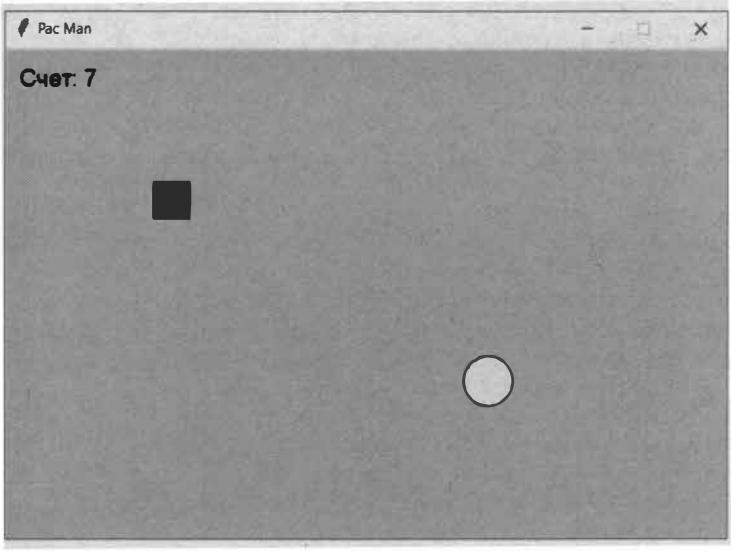
Мини-игра представляет собой упрощенный аналог известной игры Pac Man. Задача игрока — управлять шариком, который нужно перемещать с помощью клавиш-стрелок, чтобы ловить случайно появляющиеся объекты.

Листинг 19.45

```
import tkinter as tk
import random

# Создаём основное окно
root = tk.Tk()
root.title("Pac Man")
root.geometry("600x400")
root.resizable(False, False)

# Параметры игры
ball_speed = 10
target_size = 30
score = 0

# Создаём холст
canvas = tk.Canvas(root, width=600, height=400, bg="lightblue")
canvas.pack()

# Отображаем счётчик очков
```

```
score_label = tk.Label(root, text=f"Счет: {score}",
                       font=("Arial", 14), bg="lightblue")
score_label.place(x=10, y=10)

# Создаём игрока (шарик)
ball = canvas.create_oval(280, 180, 320, 220,
                          fill="yellow", outline="black")
# Создаём цель
target = canvas.create_rectangle(0, 0, target_size,
                         target_size, fill="green", outline="black")
# Перемещаем цель в случайное место
def move_target():
    x = random.randint(0, 570)
    y = random.randint(0, 370)
    canvas.coords(target, x, y, x + target_size, y + target_size)

# Двигаем шарик игрока
def move_ball(event):
    global score
    x1, y1, x2, y2 = canvas.coords(ball)
    if event.keysym == "Up" and y1 > 0:
        canvas.move(ball, 0, -ball_speed)
    elif event.keysym == "Down" and y2 < 400:
        canvas.move(ball, 0, ball_speed)
    elif event.keysym == "Left" and x1 > 0:
        canvas.move(ball, -ball_speed, 0)
    elif event.keysym == "Right" and x2 < 600:
        canvas.move(ball, ball_speed, 0)

    # Проверяем столкновение с целью
    if check_collision():
        score += 1 # Увеличиваем счёт
        # Обновляем отображение счёта
        score_label.config(text=f"Счет: {score}")
        move_target() # Перемещаем цель

# Проверяем столкновение шарика с целью
def check_collision():
    ball_coords = canvas.coords(ball)
    target_coords = canvas.coords(target)
    # Проверяем пересечение координат
    return (ball_coords[2] > target_coords[0]
            and ball_coords[0] < target_coords[2]
            and ball_coords[3] > target_coords[1]
            and ball_coords[1] < target_coords[3])

# Перемещаем цель в начало игры
```

```
move_target()
# Привязываем обработчик событий для клавиш-стрелок
root.bind("<Up>", move_ball)
root.bind("<Down>", move_ball)
root.bind("<Left>", move_ball)
root.bind("<Right>", move_ball)

root.mainloop()
```

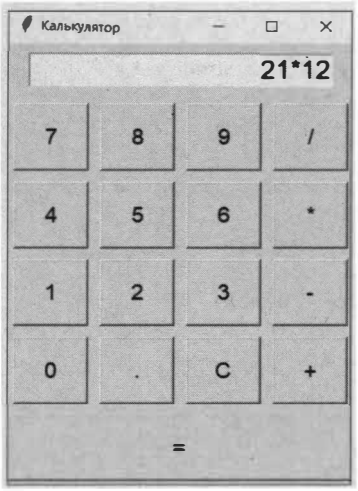


Изображение 19.39.

Приложение "Калькулятор"

```
# Импортируем библиотеку с псевдонимом tk
import tkinter as tk

# Функция для обработки нажатия кнопки "="
def calculate():
    try:
        # Получаем выражение из текстового поля
        expression = entry.get()
        # Вычисляем результат и выводим его в поле
        result = eval(expression)
        entry.delete(0, tk.END)
        entry.insert(0, str(result))
```

```
    except Exception as e:
        # Если ошибка, выводим сообщение
        entry.delete(0, tk.END)
        entry.insert(0, "Ошибка")

# Функция для добавления текста кнопок в поле ввода
def append_to_entry(symbol):
    entry.insert(tk.END, symbol)

# Функция для очистки поля ввода
def clear_entry():
    entry.delete(0, tk.END)

# Создаём главное окно
root = tk.Tk()
root.title("Калькулятор")

# Текстовое поле для ввода и отображения результата
entry = tk.Entry(root, width=20, font=("Arial", 18), justify="right")
entry.grid(row=0, column=0, columnspan=4, padx=10, pady=10)

# Определяем кнопки
buttons = [
    ("7", 1, 0), ("8", 1, 1), ("9", 1, 2), ("/", 1, 3),
    ("4", 2, 0), ("5", 2, 1), ("6", 2, 2), ("*", 2, 3),
    ("1", 3, 0), ("2", 3, 1), ("3", 3, 2), ("-", 3, 3),
    ("0", 4, 0), (".", 4, 1), ("C", 4, 2), ("+", 4, 3),
    ("=", 5, 0)
]
# Раскладываем кнопки, и прописываем условия при нажатии
for (text, row, col) in buttons:
    if text == "=":
        btn = tk.Button(root, text=text, width=10, height=2,
                        font=("Arial", 14), command=calculate)
        btn.grid(row=row, column=col, columnspan=4,
                 sticky="nsew", pady=5)
    elif text == "C":
        btn = tk.Button(root, text=text, width=5, height=2,
                        font=("Arial", 14), command=clear_entry)
        btn.grid(row=row, column=col, padx=5, pady=5)
    else:
        btn = tk.Button(root, text=text, width=5, height=2,
                        font=("Arial", 14),
                        command=lambda t=text: append_to_entry(t))
        btn.grid(row=row, column=col, padx=5, pady=5)

root.mainloop()
```



Список **buttons** состоит из нескольких кортежей, где каждый кортеж включает в себя название кнопки в кавычках, а также номера строки и колонки в таблице. Затем цикл **for** раскладывает их по соответствующим ячейкам.

Текстовое поле (entry) используется для ввода математического выражения и отображения результата. При нажатии кнопок добавляется соответствующий символ в поле ввода или выполняется действие (например, вычисление или сброс).

Функция calculate() запускает функцию eval() для подсчёта результата. При возникновении ошибок (например, при вводе букв вместо цифр), выводится сообщение "Ошибка". Кнопка "=" располагается по всей длине нижней ячейки для симметрии и удобства.

Вы можете запустить встроенный калькулятор windows, и сравнить с тем, что получилось у нас.

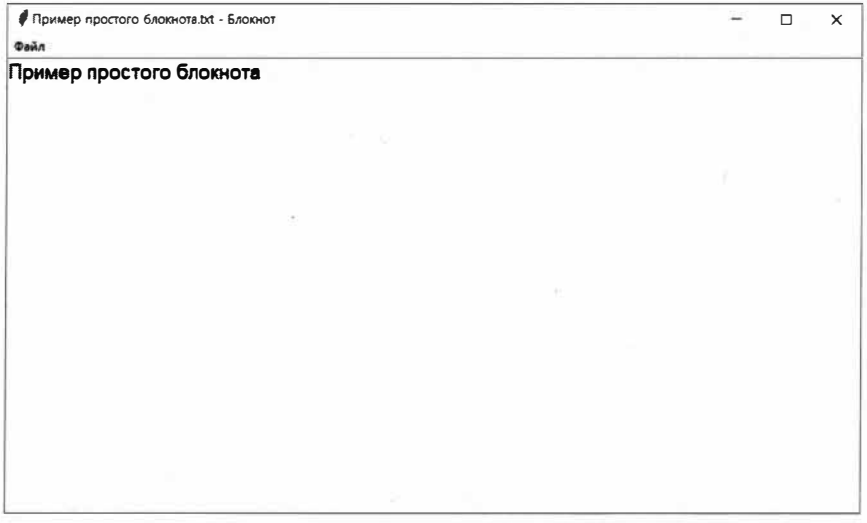
Приложение "Блокнот"

Приложение позволяет создавать, открывать, редактировать и сохранять текстовые файлы.

```
import tkinter as tk
from tkinter import filedialog, messagebox

def new_file():
    """Функция создаёт новый файл"""
    global current_file
```

```
    text_area.delete(1.0, tk.END)
    current_file = None
    root.title("Новый файл - Блокнот")

def open_file():
    """Открывает существующий файл"""
    global current_file
    file_path = filedialog.askopenfilename(
        filetypes=[("Текстовые файлы", "*.txt"),
                   ("Все файлы", "*.*")]
    )
    if file_path:
        with open(file_path, "r", encoding="utf-8") as file:
            content = file.read()
        text_area.delete(1.0, tk.END)
        text_area.insert(1.0, content)
        current_file = file_path
        root.title(f"{current_file} - Блокнот")

def save_file():
    """Сохраняет текущий файл"""
    global current_file
    if current_file:
        with open(current_file, "w", encoding="utf-8") as file:
            file.write(text_area.get(1.0, tk.END).rstrip())
        messagebox.showinfo("Сохранение",
                            "Файл успешно сохранен!")
    else:
        save_file_as()

def save_file_as():
    """Сохраняет файл как..."""
    global current_file
    file_path = filedialog.asksaveasfilename(
        defaultextension=".txt",
        filetypes=[("Текстовые файлы", "*.txt"),
                   ("Все файлы", "*.*")]
    )
    if file_path:
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_area.get(1.0, tk.END).rstrip())
        current_file = file_path
        root.title(f"{current_file} - Блокнот")

def exit_app():
    """Выход из приложения"""
    if messagebox.askyesno("Выход",
                           "Вы уверены, что хотите выйти?"):
        root.destroy()
```

```
# Создаём окно приложения
root = tk.Tk()
root.title("Блокнот")
root.geometry("800x400")

# Создаём текстовое поле
text_area = tk.Text(root, wrap="word", font=("Arial", 14))
text_area.pack(expand=True, fill="both")

# Переменная для текущего файла
current_file = None

# Создаём меню
menu_bar = tk.Menu(root)

# Меню "Файл"
file_menu = tk.Menu(menu_bar, tearoff=0)
file_menu.add_command(label="Новый", command=new_file)
file_menu.add_command(label="Открыть...", command=open_file)
file_menu.add_command(label="Сохранить", command=save_file)
file_menu.add_command(label="Сохранить как...", command=save_file_as)
file_menu.add_separator()
file_menu.add_command(label="Выход", command=exit_app)
menu_bar.add_cascade(label="Файл", menu=file_menu)

# Добавляем меню в окно
root.config(menu=menu_bar)

root.mainloop()
```



Данные программы являются хорошей демонстрацией того, как можно создать графический интерфейс для обычных пользователей, которые не знакомы со средами разработки.

19.8. Создание дистрибутивов

Создание исполняемых файлов из Python-скриптов позволяет сделать ваши программы более доступными и удобными для пользователей. Это полезно при реализации настольных приложений, инструментов автоматизации и других проектов, которые должны работать на разных компьютерах.

Существует несколько пакетов для создания исполняемых файлов:

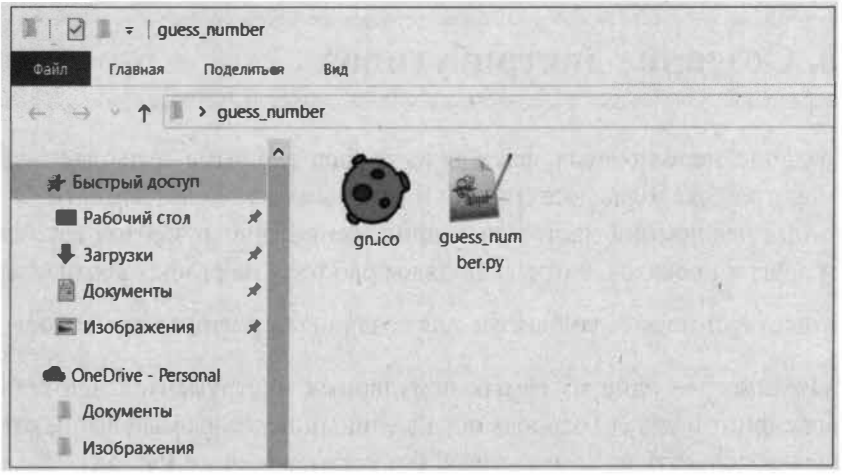
- **PyInstaller** — один из самых популярных инструментов. Он собирает ваш скрипт и все его зависимости в единый исполняемый файл, который можно запускать на компьютерах без установленного Python.
- **cx_Freeze** — похож на PyInstaller, но имеет некоторые отличия в функциональности.
- **auto-py-to-exe** — создан на основе PyInstaller и имеет графический интерфейс для создания EXE-файлов.

Выбор пакета для создания исполняемого файла зависит от ваших конкретных требований и предпочтений. Каждый из них имеет свои особенности и может быть более или менее подходящим для определенных задач. Также стоит иметь в виду, что финальный размер исполняемого файла может быть значительно больше размера вашего проекта из-за включения всех зависимостей.

Итак, давайте создадим готовый дистрибутив для игры "Угадай число", которую мы разработали в предыдущем разделе. Создайте отдельную папку и положите туда файл с кодом программы, а также изображение в формате .ico, оно станет иконкой для запуска приложения с рабочего стола.

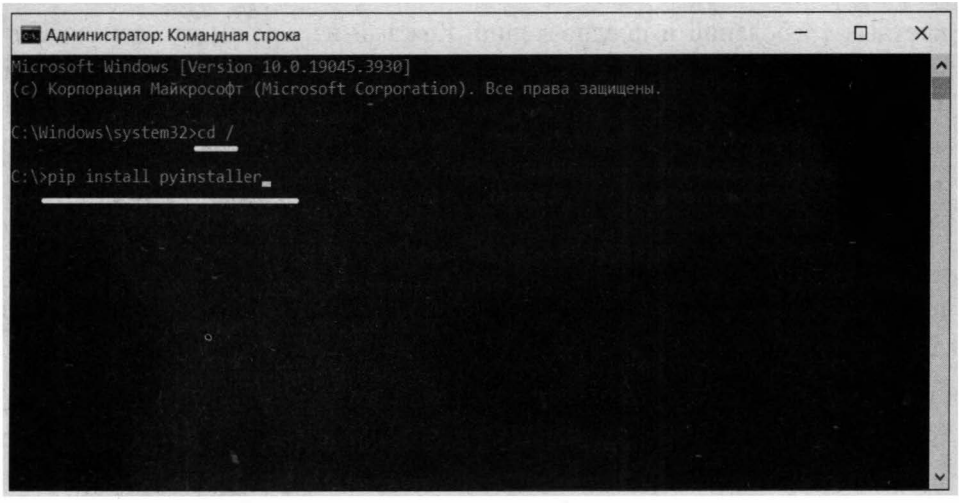
Подобное расширение можно получить из любого jpg- или png-изображения, пересохранив его в нужном формате в графическом редакторе, например в Фотошопе.

Если у вас нет таких программ, можно сконвертировать ваше изображение в онлайн-сервисах, вбив в поисковую строку соответствующий запрос. Например, "конвертировать jpg в ico".



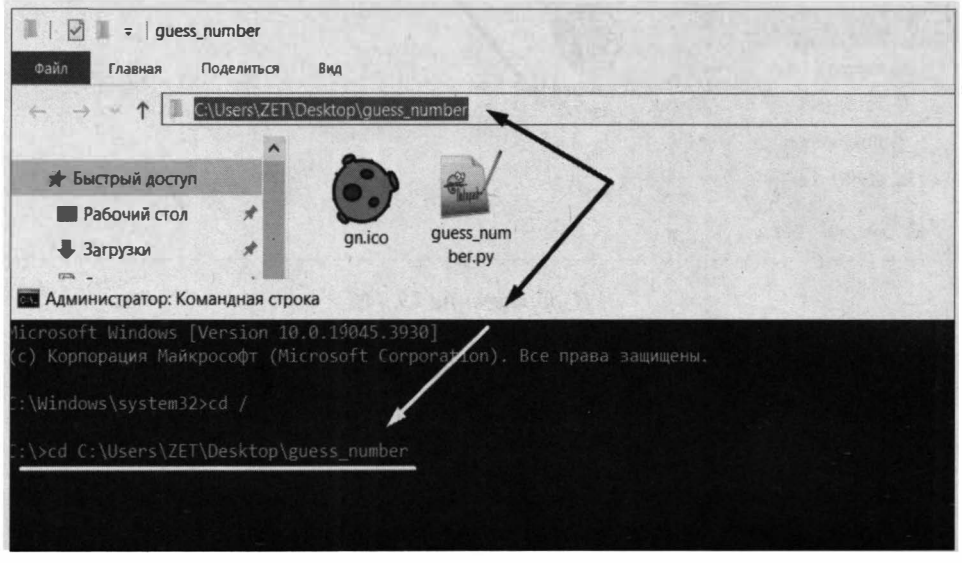
Изображение 19.40.

- Далее откройте командную строку от имени администратора, введите команду *cd /* и нажмите **Enter**, чтобы выйти из текущей директории. А затем команду *pip install pyinstaller*, чтобы установить соответствующий пакет.



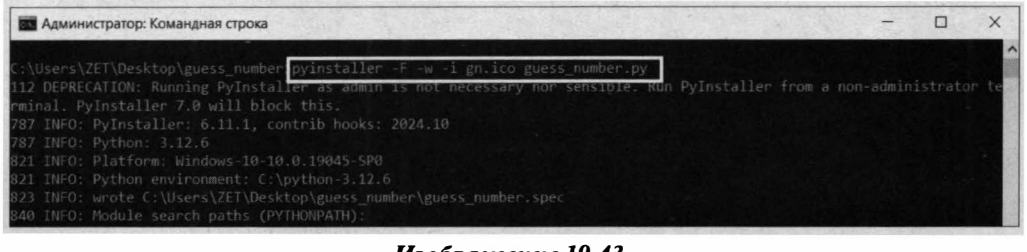
Изображение 19.41.

- После установки пакета необходимо перейти в директорию с нашей программой, прописав команду *cd путь к программе*. Полный путь к ней можно скопировать из адресной строки папки с ней, а затем вставить в командную строку. Таким образом мы попадем в эту папку из консоли.



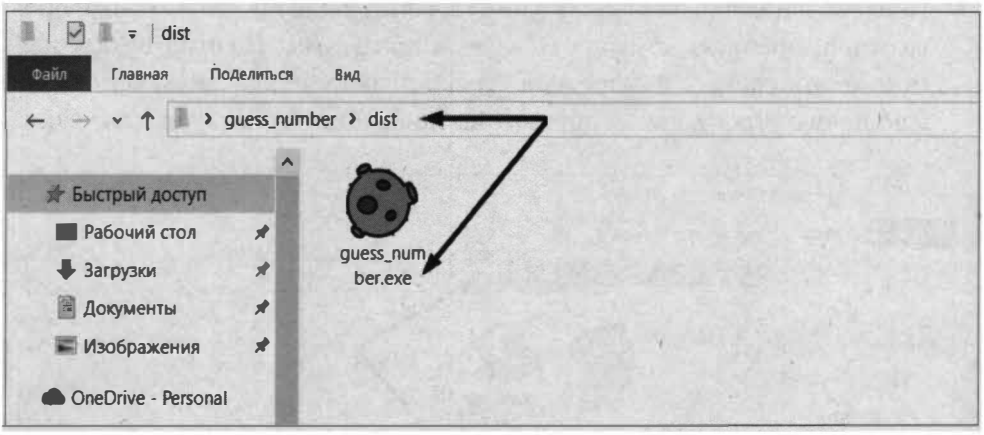
Изображение 19.42.

- Далее пропишите команду *pyinstaller -F -w -i картинка.ico скрипт.py* и нажмите **Enter**. Таким образом мы указываем файлы, для которых нужно сделать исполняемый .exe-файл.



Изображение 19.43.

В итоге, спустя некоторое время, в папке с файлами игры создадутся еще две папки: build и dist. Во второй вы найдете .exe-файл для запуска программы, которым можете поделиться с друзьями или опубликовать в сети.



Изображение 19.44.

19.8.1. Практические задания

1. Создайте окно с несколькими кнопками, каждая из которых меняет цвет фона окна на заданный (например, красный, зеленый, синий).
2. Создать окно с двумя текстовыми полями для ввода длины и ширины прямоугольника. Добавьте кнопку "Вычислить" и метку для отображения результата. При нажатии на кнопку программа должна вычислить площадь прямоугольника и вывести результат в метку.
3. Создайте .exe-файл для игры Pac Man, которую мы написали в разделе 19.7.

Глава 20.

Параллельное программирование и многопоточность

20.1. Основные понятия

В обычной, последовательной программе команды выполняются одна за другой. Параллельное программирование позволяет выполнять несколько частей программы одновременно. Например, у нас есть задача, в которой нужно посчитать сумму чисел от 1 до 100. В последовательном варианте мы бы складывали числа одно за другим: 1+2, затем к получившемуся результату прибавили 3 и так далее.

В параллельном варианте мы можем разбить задачу на несколько частей: одна часть программы считает сумму от 1 до 25, вторая — от 26 до 50, третья — от 51 до 75 и четвертая от 76 до 100. Затем результаты этих промежуточных сумм можно сложить вместе, получив итоговый результат. Очевидно, что при правильной организации это будет быстрее.

Основные понятия параллельного:

- **Процесс** — это как отдельная программа, запущенная на вашем компьютере. У каждого процесса есть свое собственное адресное пространство в памяти, то есть свои собственные данные, которые недоступны другим процессам напрямую.
- **Поток (thread)** — это "легковесный" процесс, который выполняется внутри процесса. Несколько потоков разделяют общее адресное пространство процесса, то есть имеют доступ к одним и тем же данным.
- **Многопроцессорность (multiprocessing)** — использование нескольких процессов для выполнения задач параллельно. Это позволяет в полной мере использовать возможности многоядерных процессоров.
- **Многопоточность (multithreading)** — использование нескольких потоков внутри одного процесса для выполнения задач параллельно.

Как понятно из описаний выше, параллельное программирование ускоряет выполнение программ, позволяя значительно сократить время выполнения задач, в частности если они могут быть разбиты на независимые подзадачи.

Это особенно полезно в эпоху больших объемов данных, когда количество информации растет экспоненциально, параллельные вычисления позволяют обрабатывать огромные массивы данных за приемлемое время. Кроме того, это позволяет задействовать все ядра процессора, что повышает общую производительность системы.

В Пайтон есть несколько способов реализации параллельного программирования:

- Модуль **multiprocessing** — предоставляет инструменты для создания и управления процессами. Он позволяет обойти ограничение GIL (Global Interpreter Lock / Глобальная интерпретаторная блокировка), которое препятствует истинному параллельному выполнению потоков в CPython (стандартная реализация Python). Multiprocessing подходит для задач, требующих высокой производительности и полной загрузки всех ядер процессора.
- Модуль **threading** — предоставляет инструменты для создания и управления потоками. Однако из-за GIL потоки в CPython не могут выполняться параллельно на нескольких ядрах процессора. Threading больше подходит для задач, связанных с вводом-выводом (например, сетевых запросов), где время ожидания ответа можно использовать для выполнения других задач.
- Модуль **asyncio** — предоставляет инструменты для асинхронного программирования, которое часто путают с параллельным. Асинхронное программирование позволяет выполнять несколько задач, кажущихся параллельными, переключаясь между ними в моменты ожидания. Asyncio подходит для задач, связанных с большим количеством операций ввода-вывода, где важно эффективно использовать время ожидания.

Рассмотрим упрощенный пример. Предположим, у нас есть список URL-адресов, с которых нужно скачать данные. В последовательном варианте мы бы скачивали данные с каждого адреса по очереди. В параллельном варианте мы можем создать несколько процессов, каждый из которых скачивает данные со своего адреса.

Листинг 20.1

```
import multiprocessing
import requests

def download_data(url):
    response = requests.get(url)
    # Обработка полученных данных
    return response.content

if __name__ == '__main__':
    # Список URL-адресов
    urls = ['url1', 'url2', 'url3', 'url4']
    # Создаем пул из 4 процессов
    with multiprocessing.Pool(processes=4) as pool:
    # Запускаем функцию download_data для каждого URL в отдельном
процессе
        results = pool.map(download_data, urls)

    # Обработка результатов
    print(results)
```

Стоит иметь в виду, что параллельное программирование не всегда приводит к ускорению. Если задача слишком простая, затраты на создание и управление процессами/потоками могут нивелировать выигрыш в производительности. Параллелизм в Python — это способность системы выполнять несколько задач одновременно. Важно понимать разницу между параллелизмом и конкурентностью.

Конкурентность (Concurrency) — это способность системы создавать впечатление, что она выполняет несколько задач одновременно, быстро переключаясь между ними. Представьте себе официанта в кафе, который обслуживает несколько столиков. Он не может быть одновременно у всех столиков, но он быстро перемещается между ними, создавая иллюзию одновременного обслуживания.

Параллелизм (Parallelism) — это фактическое одновременное выполнение нескольких задач. То есть это несколько официантов в том же кафе, каждый из которых обслуживает свой столик. Задачи выполняются действительно параллельно.

Параллелизм требует наличия нескольких вычислительных ресурсов (например, нескольких ядер процессора), в то время как конкурентность может быть достигнута и на одном ядре.

> **Параллелизм** — это концепция одновременного выполнения. Параллельное программирование — это способ реализации параллелизма с помощью программных средств (библиотек, языковых конструкций).

Благодаря этому можно повысить отзывчивость системы. Например, когда программа выполняет длительную операцию, параллелизм позволяет сохранить взаимодействие с интерфейсом, не "замораживая" его, как будто программа зависла или не отвечает.

20.2. Управление жизненным циклом потоков

Жизненный цикл потока — это последовательность состояний, через которые проходит поток с момента создания до своего завершения. Поток, как и живой организм, рождается, живет и умирает (завершается). Понимание этих этапов критически важно для эффективного использования многопоточности.

Основные этапы жизненного цикла следующие:

1. Создание (New) — на этом этапе поток создается как экземпляр класса threading.Thread, но еще не начинает свое выполнение. Он находится в состоянии "новорожденного".

Листинг 20.2

```
import threading

def my_function():
    print("Поток выполняется")

# Создание потока
my_thread = threading.Thread(target=my_function)
print(f"Состояние потока: {my_thread.is_alive()}")
# False - поток еще не запущен
```

2. Запуск (Runnable/Running) — после вызова метода start(), поток переходит в состояние готовности к выполнению (Runnable). Операционная система решает, когда именно поток получит процессорное время и начнет выполняться (Running). Важно понимать, что в CPython из-за GIL только один поток может выполняться в каждый момент времени на одном ядре процессора. Однако для задач, связанных с вводом-выводом (I/O), потоки могут эффективно использовать время ожидания, переключаясь между собой.

Листинг 20.3

```
my_thread.start() # Запуск потока
print(f"Состояние потока: {my_thread.is_alive()}")
# True — поток запущен
# или находится в состоянии ожидания получения процессора
```

3. Блокировка (Blocked/Waiting) — поток может перейти в состояние блокировки, если ему нужно дождаться какого-то события, например:

- завершения операции ввода-вывода (чтение файла, сетевой запрос);
- освобождения мьютекса (блокировки), который захвачен другим потоком;
- завершения другого потока (при использовании join());
- приостановки на определенное время с помощью time.sleep().

4. Завершение (Terminated/Dead) — поток завершается, когда функция, переданная в target, заканчивает свое выполнение. После этого поток "умирает", и его состояние становится "завершенным".

Листинг 20.4

```
my_thread.join() # Ожидаем завершения потока
print(f"Состояние потока: {my_thread.is_alive()}")
# False - поток завершен
```

- Метод **is_alive()** — возвращает True, если поток запущен и еще не завершил свое выполнение, False в противном случае.
- Метод **start()** — отвечает за запуск потока.

- Метод **join()** — блокирует выполнение текущего потока до тех пор, пока не завершится поток, для которого был вызван join(). Этот метод очень важен для синхронизации и предотвращения ситуации, когда основная программа завершается раньше, чем дочерние потоки.
- У метода join() есть необязательный параметр *timeout*. Если поток не завершится за указанное время, join() продолжит выполнение.

Листинг 20.5

```
import threading
import time

def worker():
    time.sleep(2) # Имитация долгой работы
    print("Поток завершил работу")

t = threading.Thread(target=worker)
t.start() # Запустили поток
t.join(1) # Ждем максимум 1 секунду
# Выведется раньше, чем сообщение из worker(), так как таймаут истек
print("Программа продолжила выполнение")
t.join() # Дождаемся окончания работы потока
print("Программа окончательно завершена")
```

- Параметр *name* — позволяет присвоить имя потоку, это может быть полезно для отладки.

Листинг 20.6

```
t = threading.Thread(target=worker, name="Поток_1")
print(t.name) # Выведет "Поток_1"
```

Пример, демонстрирующий этапы жизненного цикла:

Листинг 20.7

```
import threading
import time

def worker():
    print(f"Поток {threading.current_thread().name}: запущен")
    time.sleep(1)
    print(f"Поток {threading.current_thread().name}: выполняет
работу...")
```

```
    time.sleep(1)
    print(f"Поток {threading.current_thread().name}: завершил работу")

if __name__ == "__main__":
    thread1 = threading.Thread(target=worker, name="Поток_1")
    thread2 = threading.Thread(target=worker, name="Поток_2")

    print(f"Состояние Поток_1 (до запуска): {thread1.is_alive()}")
    thread1.start()
    print(f"Состояние Поток_1 (после запуска): {thread1.is_alive()}")
    thread2.start()

    thread1.join()
    print(f"Состояние Поток_1 (после join): {thread1.is_alive()}")
    thread2.join()

    print("Основная программа завершена")
```

Обратите внимание, что нельзя запустить поток дважды, вызвав start() повторно. Это приведет к ошибке *RuntimeError: threads can only be started once*. Также следует корректно обрабатывать исключения внутри потоков, чтобы они не приводили к падению всей программы.

При работе с общими ресурсами необходимо использовать механизмы синхронизации (Lock, RLock, Condition, Semaphore, Event, Barrier), чтобы избежать гонок данных и других проблем.

20.3. Синхронизация потоков

Когда несколько потоков работают с общими данными, возникает проблема **гонки данных** (race condition). Представьте ситуацию, когда два потока пытаются одновременно увеличить значение одной и той же переменной. Из-за непредсказуемого порядка выполнения операций результат может быть некорректным.

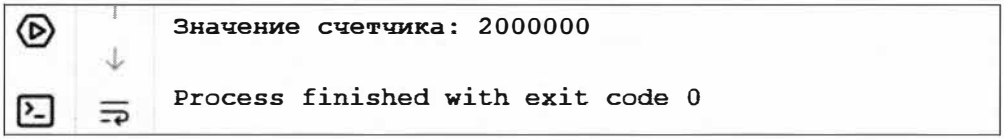
Синхронизация потоков — это набор механизмов, которые позволяют упорядочить доступ потоков к общим ресурсам, предотвращая гонку данных и обеспечивая корректное выполнение многопоточных программ.

Основные инструменты синхронизации модуля **threading**:

- **Блокировка (Lock)** — самый простой механизм синхронизации. Он позволяет защитить критическую секцию кода, доступ к которой должен быть эксклюзивным (только один поток может находиться в этой секции одновременно).
- Метод **acquire()** — захватывает блокировку. Если блокировка уже захвачена другим потоком, текущий поток блокируется и ждет, пока блокировка не будет освобождена.
- Метод **release()** — освобождает блокировку.

Листинг 20.8

```
import threading
import time

counter = 0
lock = threading.Lock() # Создаем объект блокировки

def increment():
    global counter
    for i in range(1000000):
        # Захватываем блокировку перед изменением счетчика
        lock.acquire()
        counter += 1
        # Освобождаем блокировку после изменения счетчика
        lock.release()

threads = []
for i in range(2):
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Значение счетчика: {counter}")
```

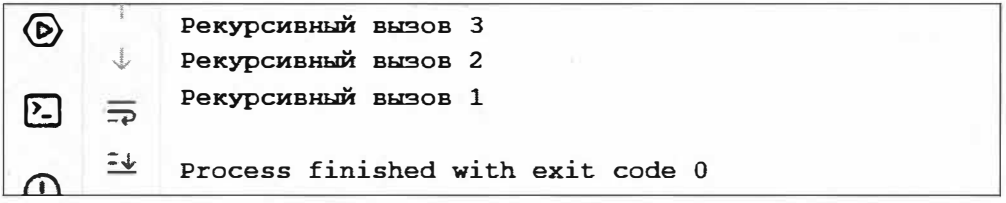


Изображение 20.1.

- **Рекурсивная блокировка RLock (Reentrant Lock)** — позволяет одному и тому же потоку захватывать блокировку несколько раз без взаимной блокировки (deadlock). Это полезно в рекурсивных функциях или когда одна функция вызывает другую, которая также использует ту же блокировку. Методы аналогичны Lock.

Листинг 20.9

```
import threading

rlock = threading.RLock()

def recursive_function(n):
    rlock.acquire()
    if n > 0:
        print(f"Рекурсивный вызов {n}")
        recursive_function(n - 1)
    rlock.release()

recursive_function(3) # Запуск функции
```



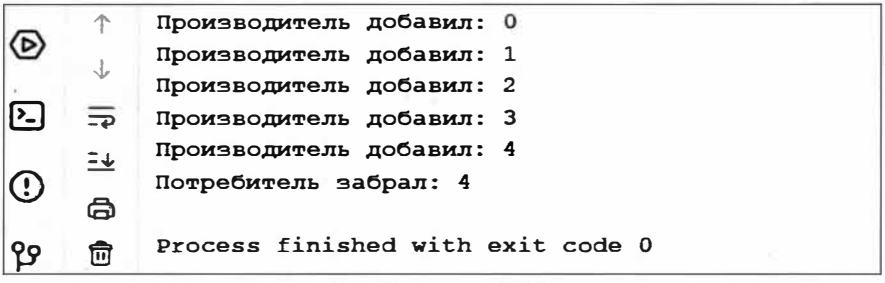
Изображение 20.2.

- **Condition** (ожидание состояния) — позволяет потокам ждать наступления определенного условия. Используется совместно с блокировкой (Lock или RLock).
- Метод **wait()** — освобождает блокировку и блокирует текущий поток до тех пор, пока другой поток не вызовет notify() или notify_all().

- Метод **notify()** — будит один из потоков, ожидающих условия.
- Метод **notify_all()** — будит все потоки, ожидающие условия.

Листинг 20.10

```
import threading
import time

condition = threading.Condition()
items = []

def producer():
    condition.acquire()
    for i in range(5):
        items.append(i)
        print(f"Производитель добавил: {i}")
        # Уведомляем потребителя о появлении нового элемента
        condition.notify()
    condition.release()

def consumer():
    condition.acquire()
    while not items: # Проверяем, есть ли элементы
        print("Потребитель ждет...")
        condition.wait() # Ждем, пока производитель добавит элемент
    item = items.pop()
    print(f"Потребитель забрал: {item}")
    condition.release()

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)

t1.start()
t2.start()

t1.join()
t2.join()
```



Изображение 20.3.

- **Semaphore** — счетчик, который ограничивает количество потоков, способные одновременно получить доступ к ресурсу. Инициализируется с максимальным количеством разрешений.
- Метод **acquire()** — уменьшает счетчик. Если счетчик равен 0, поток блокируется.
- Метод **release()** — увеличивает счетчик.

Листинг 20.11

```
import threading
import time

# Максимум 2 потока могут получить доступ одновременно
semaphore = threading.Semaphore(2)

def worker():
    semaphore.acquire()
    print(f"Поток {threading.current_thread().name} получил доступ")
    time.sleep(2)
    print(f"Поток {threading.current_thread().name} освободил доступ")
    semaphore.release()

threads = []
for i in range(5):
    t = threading.Thread(target=worker, name=f"Поток-{i}")
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

```
Поток Поток-0 получил доступ
Поток Поток-1 получил доступ
Поток Поток-1 освободил доступ
Поток Поток-0 освободил доступ
Поток Поток-3 получил доступ
Поток Поток-2 получил доступ
Поток Поток-3 освободил доступ
Поток Поток-2 освободил доступПоток Поток-4 получил доступ

Поток Поток-4 освободил доступ
```

Изображение 20.4.

- **Event** — простейший механизм синхронизации, основанный на внутреннем флаге.

- Метод **set()** — устанавливает флаг в True.
- Метод **clear()** — сбрасывает флаг в False.
- Метод **wait()** — блокирует поток до тех пор, пока флаг не будет установлен в True.

Листинг 20.12

```
import threading
import time

event = threading.Event()

def worker():
    print("Поток ждет события...")
    event.wait() # Блокируется, пока не будет вызван event.set()
    print("Поток получил событие и продолжает работу")

def main_thread():
    time.sleep(2)
    print("Устанавливаем событие...")
    event.set() # Разблокируем ждущий поток

t = threading.Thread(target=worker)
main_t = threading.Thread(target=main_thread)
t.start()
main_t.start()
t.join()
main_t.join()
```

```
Поток ждет события...
Устанавливаем событие...
Поток получил событие и продолжает работу

Process finished with exit code 0
```

Изображение 20.5.

- **Barrier** — синхронизирует заданное количество потоков, блокируя их до тех пор, пока все они не достигнут барьера. Инициализируется с количеством потоков, которые должны достичь барьера.
- Метод **wait()** — блокирует поток, пока необходимое количество потоков не вызовет этот метод. После этого все потоки одновременно продолжают выполнение.

Листинг 20.13

```
import threading
import time

# Барьер для 3 потоков
barrier = threading.Barrier(3)

def worker():
    print(f"Поток {threading.current_thread().name} достиг барьера")
    try:
        # Ждем, пока остальные потоки достигнут барьера
        barrier.wait()
    # Обработка ситуации, когда один из потоков прервал ожидание
    except threading.BrokenBarrierError:
        print(f"Barrier broken in thread {threading.current_thread().name}")
    print(f"Поток {threading.current_thread().name} продолжает работу")

threads = []
for i in range(3):
    t = threading.Thread(target=worker, name=f"Thread-{i}")
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

```
Поток Thread-0 достиг барьера
Поток Thread-1 достиг барьера
Поток Thread-2 достиг барьера
Поток Thread-2 продолжает работу
Поток Thread-1 продолжает работу
Поток Thread-0 продолжает работу

Process finished with exit code 0
```

Изображение 20.6.

Для полного понимания рекомендуется запустить все примеры, и посмотреть результат их работы. Каждый из инструментов синхронизации подходит для разных ситуаций.

20.4. Глобальная блокировка интерпретатора (GIL)

GIL (Global Interpreter Lock) — это глобальная блокировка интерпретатора Python, которая позволяет только одному потоку выполнять байт-код в любой момент времени внутри одного процесса. То есть даже на многоядерном процессоре только одно ядро будет фактически выполнять код в данный момент. Все остальные потоки будут ждать своей очереди.

Это можно представить как однополосную дорогу с регулировщиком. Даже если на дороге много машин (потоков), регулировщик (GIL) пропускает только одну машину за раз. Остальные машины вынуждены ждать, пока дорога не освободится.

GIL был введен по нескольким причинам, связанным с внутренней реализацией CPython (стандартной реализации Python). Во-первых, это упрощает управление памятью. Без GIL потребовались бы более сложные и ресурсоемкие механизмы синхронизации для защиты объектов в памяти от одновременного доступа из разных потоков.

Кроме того, многие библиотеки Python написаны на языке C (например, NumPy, SciPy). GIL обеспечивает совместимость с этими расширениями, так как они часто не являются потокобезопасными.

Дело в том, что Python, будучи интерпретируемым языком, может быть медленным для выполнения вычислительно-интенсивных задач. Чтобы преодолеть это ограничение и обеспечить высокую производительность, разработчики часто используют язык C для написания критически важных частей библиотек.

GIL оказывает существенное влияние на производительность многопоточных программ. Для задач, интенсивно использующих процессор (например, сложные вычисления, обработка изображений), многопоточность практически не дает прироста производительности, а иногда даже может привести к ее снижению из-за накладных расходов на переключение между потоками.

Листинг 20.14

```
# Пример влияния GIL
import threading
import time

# Функция вычитает из заданного числа единицу, пока не дойдет до 0
def count(n):
    while n > 0:
        n -= 1

# Отсечка начала последовательного выполнения
start_time = time.time()

# Последовательное выполнение
count(20000000) # Передаем в функцию сначала первое число
count(20000000) # Затем второе

# Отсечка завершения последовательного выполнения
end_time = time.time()
print(f"Последовательное выполнение: {end_time - start_time} секунд")

# Отсечка начала многопоточного выполнения
start_time = time.time()

# Многопоточное выполнение (не очень эффективно из-за GIL)
t1 = threading.Thread(target=count, args=(20000000,))
t2 = threading.Thread(target=count, args=(20000000,))

t1.start()
t2.start()

t1.join()
t2.join()

# Отсечка завершения многопоточного выполнения
end_time = time.time()
print(f"Многопоточное выполнение: {end_time - start_time} секунд"
```

```
Последовательное выполнение: 2.4654009342193604 секунд
Многопоточное выполнение: 2.1860549449920654 секунд

Process finished with exit code 0
```

Изображение 20.7.

В этом примере многопоточное выполнение занимает примерно столько же времени, сколько и последовательное. В таких случаях лучше исполь-

зовать **многопроцессорность** (модуль multiprocessing), которая позволяет обойти ограничение GIL и задействовать все ядра процессора.

Как обойти GIL?

Как уже упоминалось, использование нескольких процессов позволяет обойти ограничение GIL, так как каждый процесс имеет свой собственный интерпретатор и свою собственную память.

Некоторые библиотеки, такие как NumPy, выполняют большую часть вычислений на C, за пределами интерпретатора Python. Во время выполнения этих операций GIL освобождается, что позволяет другим потокам выполняться параллельно. Также существуют альтернативные реализации Python, такие как Jython (работает на JVM) и IronPython (работает на .NET), которые не имеют GIL.

20.5. Асинхронное программирование

Асинхронное программирование — это способ организации кода, при котором выполнение программы не блокируется в ожидании завершения какой-либо операции, например чтения файла, сетевого запроса или ввода данных пользователем. Вместо этого программа может переключиться на выполнение других задач, пока операция не завершится.

Продолжая аналогию с официантом:

- **Синхронный подход** — официант принимает заказ у первого клиента, идет на кухню, ждет, пока блюдо будет готово, приносит его клиенту, затем принимает заказ у второго клиента и так далее. Официант тратит много времени в ожидании.
- **Асинхронный подход** — официант принимает заказы у нескольких клиентов, передает их на кухню и затем занимается другими делами, например, обслуживает другие столики или принимает новые заказы. Когда блюдо готово, повар сообщает официанту, и тот относит заказ клиенту. Официант использует свое время гораздо эффективнее.

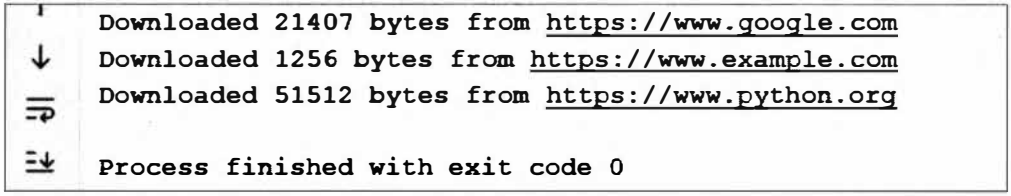
В программировании роль официанта играет программа, а роль кухни — операция обработки данных. Например, функция принимает данные, вычисляет и возвращает результат.

Асинхронное программирование состоит из двух ключевых моментов:

1. **Сопрограммы (coroutines)** — это специальные функции, которые могут приостанавливать свое выполнение и возобновлять его позже. Для определения сопрограмм используются ключевые слова **async** и **await**.
   - **async** — ключевое слово для объявления сопрограммы.
   - **await** — для приостановки выполнения сопрограммы до тех пор, пока не завершится другая асинхронная операция.
2. **Цикл событий** (event loop) — это механизм, который управляет выполнением сопрограмм, переключаясь между ними в точках ожидания.

Листинг 20.15

```
import asyncio
import aiohttp # Библиотека для асинхронных HTTP-запросов

async def download_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    urls = [
        "https://www.google.com",
        "https://www.example.com",
        "https://www.python.org"
    ]

    tasks = [download_data(url) for url in urls]
    # Запускаем все запросы "параллельно" (асинхронно)
    results = await asyncio.gather(*tasks)

    for url, result in zip(urls, results):
        print(f"Downloaded {len(result)} bytes from {url}")

if __name__ == "__main__":
    asyncio.run(main())
```



Изображение 20.8.

В этом примере функция ***download_data*** является сопрограммой. Ключевое слово *await* используется для приостановки выполнения сопрограммы до тех пор, пока не будет получен ответ от сервера.

asyncio.gather позволяет запустить несколько сопрограмм "параллельно" (асинхронно), что значительно ускоряет выполнение программы, особенно при большом количестве запросов.

Важно понимать разницу между асинхронным программированием и многопоточностью. Асинхронное — переключается между задачами во время их "простоя" (один официант, переключаясь, делает две задачи), многопоточное — выполняет задачи параллельно (два официанта делают две задачи).

Преимущество асинхронного программирования в том, что программа не тратит время на ожидание, а выполняет другие задачи. Меньшее потребление памяти и ресурсов процессора по сравнению с многопоточностью для I/O-bound задач. Программа остается отзывчивой даже во время выполнения длительных операций.

К недостаткам можно отнести то, что код может быть сложнее для понимания и отладки, особенно для начинающих.

Не подходит для CPU-bound задач. То есть задач, интенсивно использующих процессор.

В данном случае многопоточность или многопроцессорность будут более эффективными.

20.5.1. Практические задания

1. Имеется список любых чисел. Необходимо параллельно вычислить квадрат каждого числа, а затем сложить полученные результаты. Используйте модуль **multiprocessing**.
2. Несколько потоков должны уменьшать значение общего счетчика. Необходимо использовать блокировку (threading.Lock) для предотвращения гонки данных.

Глава 21.

Сериализация данных

21.1. Что такое сериализация

Сериализация (serialization) — это процесс преобразования объекта в последовательность байтов. Представим, что у нас есть сложная структура данных в памяти компьютера, например список словарей, содержащих информацию о пользователях. Чтобы сохранить эту структуру в файл или передать по сети, нужно представить ее в виде линейной последовательности байтов. Именно это и делает сериализация.

Обратный процесс, когда последовательность байтов преобразуется обратно в объект, называется **десериализацией** (deserialization).

Сериализация решает несколько важных задач. Многие программы сохраняют свое состояние между запусками. Сериализация позволяет сохранить объекты в файл, а при следующем запуске программы восстановить их из файла.

При обмене данными между компьютерами по сети объекты необходимо преобразовать в байты для передачи по каналам связи. Это также решается сериализацией. Вместе с тем она позволяет сохранять объекты в кэше для быстрого доступа. Вместо того чтобы каждый раз создавать объект заново, его можно загрузить из кэша в сериализованном виде. А в многопроцессных приложениях сериализация используется для передачи данных между процессами.

В Пайтон для сериализации используются несколько модулей, наиболее распространенные из которых **pickle** и **json**.

Модуль **pickle** является встроенным модулем Python, который позволяет сериализовать практически любые объекты, включая пользовательские классы. Он сериализует данные в бинарном формате, что делает их нечитаемыми для человека, но более компактными и быстрыми при обработке.

Основные функции модуля следующие:

- **pickle.dump(obj, file)** — сериализует объект (obj) и записывает результат в открытый файл (file).
- **pickle.dumps(obj)** — сериализует объект и возвращает результат в виде байтовой строки. **pickle.load(file)** — десериализует объект из открытого файла.
- **pickle.loads(bytes_obj)** — десериализует объект из байтовой строки.

Листинг 21.1

```
import pickle

# Словарь с данными
data = {
    "name": "Иван",
    "age": 30,
    "city": "Москва",
    "hobbies": ["программирование", "путешествия"]
}
# Сериализация в файл
# Открываем файл в бинарном режиме записи ("wb")
with open("data.pickle", "wb") as f:
    pickle.dump(data, f)

# Десериализация из файла
# Открываем файл в бинарном режиме чтения ("rb")
with open("data.pickle", "rb") as f:
    loaded_data = pickle.load(f)

print(loaded_data) # Выведет исходный словарь
print(type(loaded_data)) # Выведет тип объекта
```

Стоит помнить, что модуль **pickle** небезопасен для десериализации данных из ненадежных источников. То есть десериализация данных, полученных из ненадежного источника, может привести к выполнению произвольного кода. Поэтому не рекомендуется использовать **pickle** для обработки данных, полученных из Интернета или от ненадежных пользователей.

Модуль **json** — используется для сериализации данных в формате JSON (JavaScript Object Notation). JSON — текстовый формат, который легко читается человеком и поддерживается многими языками программирования. Данный модуль может сериализовать только простые типы данных, такие как словари, списки, строки, числа и булевы значения. Он не может сериализовать пользовательские классы или другие сложные объекты напрямую.

Основные функции модуля следующие:

- **json.dump(obj, file)** — сериализует объект (obj) в формате JSON и записывает результат в открытый файл (file).
- **json.dumps(obj)** — сериализует объект в формате JSON и возвращает результат в виде строки.
- **json.load(file)** — десериализует объект из открытого файла в формате JSON. **json.loads(str_obj)** — десериализует объект из строки в формате JSON.

Листинг 21.2

```
import json

# Словарь с данными
data = {
    "name": "Иван",
    "age": 30,
    "city": "Москва"
}
# Сериализация в JSON строку
# indent=4 пробела для красивого форматирования
json_string = json.dumps(data, indent=4)
print(json_string)
```

```
# Сериализация в файл
with open("data.json", "w") as f:
    json.dump(data, f, indent=4)

# Десериализация из JSON строки
loaded_data = json.loads(json_string)
print(loaded_data)
print(type(loaded_data))

# Десериализация из файла
with open("data.json", "r") as f:
    loaded_data_from_file = json.load(f)

print(loaded_data_from_file)
print(type(loaded_data_from_file))
```

Модуль **json** безопасен для десериализации данных из ненадежных источников, так как поддерживает только простые типы данных.

21.2. Сериализация пользовательских классов

Сериализация и десериализация пользовательских классов происходит аналогична примеру из предыдущего раздела. Поэтому здесь рассмотрим стандартный вариант для понимания общей картины.

Листинг 21.3

```
import pickle

# Пользовательский класс с данными о человеке
class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address

    def info(self):
        print(f"{self.name}, {self.age}, {self.address}")

# Создаём объект класса Person
person = Person("Иван", 30, "Москва, ул. Ленина, д. 10")
```

```
# Сериализация объекта в файл
with open("person.pickle", "wb") as f:
    pickle.dump(person, f)

# Десериализация объекта из файла
with open("person.pickle", "rb") as f:
    loaded_person = pickle.load(f)

# Проверяем, что объекты восстановлены корректно
loaded_person.info()
print(type(loaded_person))
```

Практический пример сериализации и десериализации

В качестве практического задания попробуйте сериализовать и десериализовать этот пользовательский класс с помощью модуля **json**.

Листинг 21.4

```
import json

# Пользовательский класс с данными о человеке
class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address

    def info(self):
        print(f"{self.name}, {self.age}, {self.address}")

# Функция для преобразования объекта Person в словарь
def person_to_dict(person):
    return {
        "name": person.name,
        "age": person.age,
        "address": person.address
    }
# Функция для создания объекта Person из словаря
def dict_to_person(person_dict):
    return Person(person_dict["name"],
                  person_dict["age"],
                  person_dict["address"])
```

```
# Объект
person = Person("Иван", 30, "Москва, ул. Ленина, д. 10")

# Сериализация в JSON
with open("person.json", "w") as f:
    json.dump(person_to_dict(person), f, indent=4)

# Десериализация из JSON
with open("person.json", "r") as f:
    person_dict = json.load(f)
loaded_person = dict_to_person(person_dict)

# Проверяем результат
print(loaded_person.info())
print(type(loaded_person))
```

21.3. Проблемы при сериализации

При сериализации/десериализации может возникать ряд проблем. Давайте рассмотрим самые частые из них, с которыми приходится сталкиваться.

Несериализуемые типы данных

Модуль **json** поддерживает только ограниченный набор типов данных: словари, списки, строки, числа, булевы значения и None. Он не может напрямую сериализовать объекты пользовательских классов, множества, даты, времени и другие сложные типы. Для пользовательских классов необходимо преобразование объекта в словарь и обратно (смотрите решение практического задания из раздела 19.2).

Для дат и времени можно использовать строковое представление в формате ISO 8601 (например, с помощью метода isoformat()), а при десериализации преобразовывать строку обратно в объект даты/времени.

Листинг 21.5

```
import json
import datetime

data = {
```

```
    "event": "Meeting",
    # Год, месяц, день, часы, минуты
    "date": datetime.datetime(2025, 1, 15, 10, 0).isoformat()
}
json_string = json.dumps(data)
loaded_data = json.loads(json_string)

loaded_date = datetime.datetime.fromisoformat(loaded_data["date"])
print(loaded_date) # Результат: 2025-01-15 10:00:00
```

Совместимость между версиями Python

Формат, используемый **pickle**, может меняться между разными версиями Пайтона. То есть данные, сериализованные в одной версии Python, могут не десериализоваться в другой. Для решения этой проблемы можно использовать протокол **pickle** версии 0 (самый старый и наиболее совместимый), но это может привести к увеличению размера сериализованных данных.

Листинг 21.6

```
pickle.dump(obj, file, protocol=0)
```

Циклические ссылки

Циклические ссылки возникают, когда объект ссылается сам на себя напрямую или косвенно через другие объекты. Модуль **pickle** может обрабатывать циклические ссылки, а **json** нет.

Листинг 21.7

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

# Объекты
a = Node(1)
b = Node(2)
a.next = b
b.next = a # Циклическая ссылка
```

```
# Попытка сериализации с помощью json приведет к ошибке
# json.dumps(a.__dict__)
# TypeError: Object of type 'Node' is not JSON serializable

# pickle справится с этой ситуацией
import pickle
pickle.dumps(a) #Сериализация пройдет успешно
```

При использовании модуля **json** необходимо избегать циклических ссылок. Если они неизбежны, придется реализовать специальную логику для обработки таких ситуаций, например, сохранять только идентификаторы объектов вместо самих объектов.

Большие объемы данных

Сериализация больших объемов данных может потребовать значительных ресурсов памяти и времени. Поэтому рекомендуется использовать более эффективные форматы, такие как Protocol Buffers. Также можно рассмотреть потоковую сериализацию, когда данные записываются или читаются частями, а не целиком.

Неожиданные ошибки при десериализации

В процессе разработки код классов может меняться (добавляться или удаляться атрибуты, меняться методы). Если вы попытаетесь десериализовать объект, сохраненный с использованием старой версии класса, это может привести к ошибкам.

Поэтому при сериализации стоит сохранять информацию о версии класса. При десериализации проверять версию и выполнять необходимые преобразования, если версии не совпадают.

Общие рекомендации при сериализации

- Обдуманно выбирайте формат сериализации в зависимости от ваших потребностей (pickle, json или другие).

- Обращайте внимание на безопасность при использовании **pickle** с данными из ненадежных источников.
- Предусматривайте обработку возможных ошибок при десериализации.

Глава 22.

Работа с базами данных

22.1. Что такое база данных

База данных (БД) — это организованная структура, предназначенная для управления и доступа к данным. Она обеспечивает удобный способ хранения и извлечения информации, а также гарантирует ее целостность и безопасность.

Базу данных можно представить как большую библиотеку. В ней хранится огромное количество книг, каждая из которых имеет свой уникальный идентификатор, автора, название, год издания и другие характеристики. Чтобы быстро найти нужную книгу, библиотекари используют каталоги и системы классификации. База данных — это, по сути, электронный аналог такой библиотеки.

Существует множество различных типов БД, но наиболее распространены два основных:

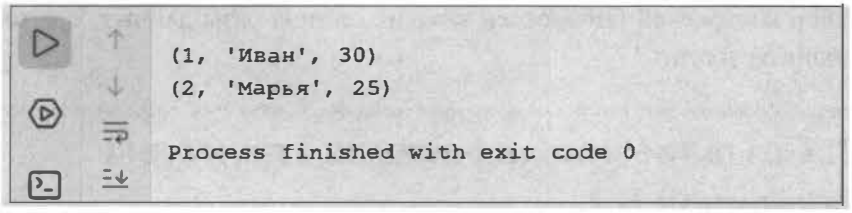
Реляционные базы данных (SQL) — данные хранятся в виде таблиц, связанных между собой отношениями. Для работы с реляционными базами используется язык SQL (Structured Query Language).

Нереляционные базы данных (NoSQL) — данные хранятся в различных форматах, отличных от таблиц, например, в виде документов, графов или пар ключ-значение. NoSQL-базы данных часто используются для хранения больших объемов неструктурированной информации.

Python предлагает различные библиотеки для работы с базами данных. Одна из них SQLite — это легкая файловая база данных, которая не требует отдельного сервера. Она встроена в Пайтон, благодаря чему удобна для небольших проектов и обучения.

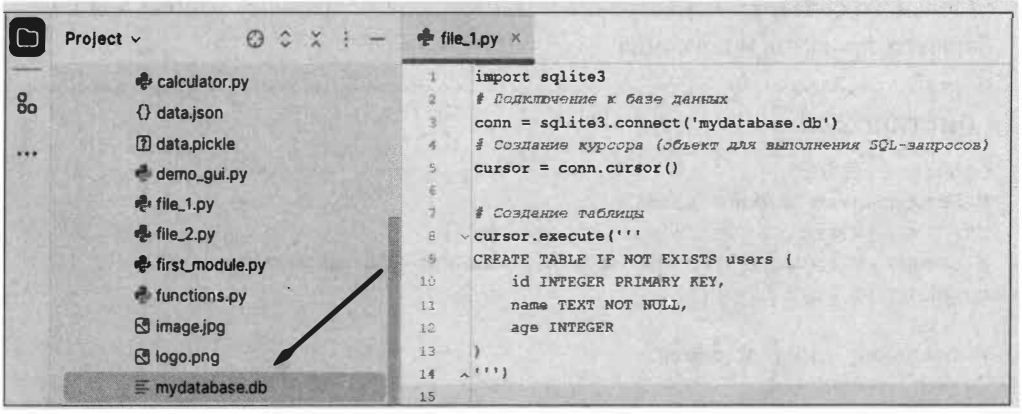
Листинг 22.1

```
import sqlite3
# Подключение к базе данных
conn = sqlite3.connect('mydatabase.db')
# Создание курсора (объект для выполнения SQL-запросов)
cursor = conn.cursor()

# Создание таблицы users
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER )
''')

# Вставка данных
cursor.execute("INSERT INTO users (name, age) VALUES ('Иван', 30)")
cursor.execute("INSERT INTO users (name, age) VALUES ('Марья', 25)")
conn.commit() # Сохранение изменений

# Выполнение запроса SELECT
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
    print(row)

# Закрытие соединения
conn.close()
```



Изображение 22.1.

В примере sqlite3.connect('mydatabase.db') подключается к базе данных mydatabase.db. Если файл не существует, он будет создан:



Изображение 22.2.

- cursor = conn.cursor() — создает объект курсора, который используется для выполнения SQL-запросов.
- cursor.execute(...) — выполняет SQL-запросы.
- conn.commit() — сохраняет изменения в базе данных.
- cursor.fetchall() — получает все результаты запроса.
- conn.close() — закрывает соединение с базой данных.

Помимо SQLite, распространены другие базы данных:

- **psycopg2** — для работы с PostgreSQL.
- **mysql-connector-python** — для работы с MySQL.
- **pymongo** — для работы с MongoDB.

Выбор конкретной библиотеки зависит от типа базы данных, с которой необходимо работать.

22.2. Создание и модификация структуры реляционной БД

Основные операции, которые мы можем выполнять с базой данных, это создание таблиц, их модификация и удаление. Для этого используются команды CREATE, ALTER и DROP. Создание и изменение структуры является основой управления данными. Эти операции позволяют организовать информацию в удобные таблицы.

CREATE TABLE (Создание таблицы)

Команда CREATE используется для создания новых таблиц в базе данных.

В предыдущем разделе с помощью этой команды мы создали таблицу **users**, которая будет содержать информацию о пользователях.

Листинг 22.2

```
import sqlite3
# Подключаемся к базе данных
conn = sqlite3.connect('my_database.db')
cursor = conn.cursor()

# Создание таблицы users
cursor.execute('''
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        age INTEGER
    )
    ''')
# Сохраняем изменения
conn.commit()
conn.close()
```

В данном примере мы определяем таблицу с тремя колонками: *id*, *name* и *age*. Поле *id* является первичным ключом, то есть оно должно быть уникальным для каждой записи.

ALTER TABLE (Изменение таблицы)

Команда ALTER используется для изменения существующей таблицы.

Мы можем добавлять, изменять или удалять столбцы. Давайте добавим новый столбец *email* в нашу таблицу:

Листинг 22.3

```
import sqlite3
# Подключаемся к базе данных
conn = sqlite3.connect('my_database.db')
cursor = conn.cursor()

# Добавляем новый столбец email
cursor.execute('''
    ALTER TABLE users
    ADD COLUMN email TEXT ''')
# Сохраняем изменения
conn.commit()
conn.close()
```

После выполнения данного кода в таблице **users** появится новый столбец, который будет хранить адреса электронной почты пользователей.

DROP TABLE (Удаление таблицы)

Команда DROP используется для удаления таблиц (или других объектов) из базы данных.

Стоит быть осторожным, так как при удалении таблицы все данные в ней тоже будут потеряны. Например, если мы хотим удалить таблицу **users**, это можно сделать следующим образом:

Листинг 22.4

```
import sqlite3
# Подключаемся к базе данных
conn = sqlite3.connect('my_database.db')
cursor = conn.cursor()

# Удаляем таблицу users
cursor.execute('DROP TABLE IF EXISTS users')

# Сохраняем изменения
conn.commit()
conn.close()
```

Здесь мы добавили условие IF EXISTS, которое предотвращает возникновение ошибок, если таблицы не существует.

22.3. SQL: углубленное изучение

Работа с реляционными базами данных требует понимания ряда ключевых концепций и навыков.

Основные понятия, связанные с базами данных

- **Схема базы данных** — описание структуры базы данных, включающее таблицы, столбцы, типы данных и связи между ними.
- **Данные** — это информация, представленная в определенной форме.
- **Таблица** — основная единица хранения данных в реляционных БД. Представляет собой набор строк (записей) и столбцов (полей).
- **Строка** (запись) — набор значений, представляющих отдельный элемент данных в таблице.
- **Столбец** (поле, атрибут) — характеристика данных в таблице. Каждый столбец имеет имя и тип данных (например, текст, число, дата).
- **Ключ** — столбец или набор столбцов, однозначно идентифицирующих каждую строку в таблице.

- **Запрос** — операция извлечения, добавления, изменения или удаления данных в базе данных.

Изображение 22.3.

Основные операции

- SELECT — извлечение данных.
- INSERT — добавление данных.
- UPDATE — изменение данных.
- DELETE — удаление данных.

Основные операторы

Для понимания работы операторов давайте создадим таблицу, которая будет содержать информацию о сотрудниках компании:

Листинг 22.5

```
import sqlite3
# Подключение к базе данных
conn = sqlite3.connect('newDB.db')
cursor = conn.cursor()

# Создание таблицы employees
cursor.execute('''
CREATE TABLE IF NOT EXISTS employees (
```

```
    id INTEGER PRIMARY KEY,
    name TEXT,
    department TEXT,
    salary REAL,
    )
''')

# Заполнение таблицы данными (если она пустая)
cursor.execute("INSERT OR IGNORE INTO employees (name, department,
salary) VALUES ('Иван', 'IT', 50000)")
cursor.execute("INSERT OR IGNORE INTO employees (name, department,
salary) VALUES ('Марья', 'Sales', 60000)")
cursor.execute("INSERT OR IGNORE INTO employees (name, department,
salary) VALUES ('Петр', 'IT', 70000)")
cursor.execute("INSERT OR IGNORE INTO employees (name, department,
salary) VALUES ('Анна', 'Marketing', 55000)")
conn.commit() # Сохранение изменений
```

Оператор **WHERE** — используется для фильтрации строк на основе заданного условия.

Например, мы можем отобразить только сотрудников определенного отдела или с определенным уровнем зарплаты:

Листинг 22.6

```
# Выбор всех сотрудников из отдела 'IT'
cursor.execute("SELECT * FROM employees WHERE department = 'IT'")
it_employees = cursor.fetchall()
print("Сотрудники IT отдела:", it_employees)

# Выбор сотрудников с зарплатой больше 60000
cursor.execute("SELECT name, salary FROM employees WHERE salary >
60000")
high_salary_employees = cursor.fetchall()
print("Сотрудники с высокой зарплатой:", high_salary_employees)
```

```
Сотрудники IT отдела: [(1, 'Иван', 'IT', 50000.0), (3, 'Петр', 'IT', 70000.0)]
Сотрудники с высокой зарплатой: [('Петр', 70000.0)]

Process finished with exit code 0
```

Изображение 22.4.

Оператор **JOIN** — используется для объединения данных из двух или более таблиц на основе общего столбца.

Существует несколько типов JOIN.

- **INNER JOIN** (или просто JOIN) — возвращает строки только тогда, когда есть совпадение в обеих таблицах.
- **LEFT JOIN** — возвращает все строки из левой таблицы и соответствующие строки из правой таблицы. Если совпадений нет, возвращаются значения NULL для столбцов правой таблицы.
- **RIGHT JOIN** — аналогично LEFT JOIN.
- **FULL OUTER JOIN** — возвращает все строки из обеих таблиц. Если совпадений нет, возвращаются значения NULL для соответствующих столбцов.

Для демонстрации JOIN создадим еще одну таблицу **departments**, в которой будет содержаться информация, в каких городах находится каждый из отделов:

Листинг 22.7

```
# Новая таблица departments
cursor.execute('''
CREATE TABLE IF NOT EXISTS departments (
    id INTEGER PRIMARY KEY,
    name TEXT,
    location TEXT )
''')

cursor.execute("INSERT OR IGNORE INTO departments (name, location)
VALUES ('IT', 'Москва')")
cursor.execute("INSERT OR IGNORE INTO departments (name, location)
VALUES ('Sales', 'Санкт-Петербург')")
cursor.execute("INSERT OR IGNORE INTO departments (name, location)
VALUES ('Marketing', 'Новосибирск')")
conn.commit()
```

Теперь выполним INNER JOIN:

Листинг 22.8

```
cursor.execute('''
SELECT e.name, d.location
FROM employees AS e
INNER JOIN departments AS d ON e.department = d.name
''')
employee_locations = cursor.fetchall()
print("Расположение сотрудников:", employee_locations)
```

В данном примере сотрудники из разных отделов были присоединены к определенному городу. То есть, в зависимости от того, в каком отделе работает сотрудник, ему была проставлена соответствующая локация.

Оператор **GROUP BY** — используется для группировки строк с одинаковыми значениями в одном или нескольких столбцах. Обычно используется вместе с агрегатными функциями, такими как COUNT(), SUM(), AVG(), MIN(), MAX().

Листинг 22.9

```
# Подсчет количества сотрудников в каждом отделе
cursor.execute("SELECT department, COUNT(*) FROM employees GROUP
BY department")
employees_per_department = cursor.fetchall()
print("Количество сотрудников в каждом отделе:", employees_per_
department)

# Средняя зарплата в каждом отделе
cursor.execute("SELECT department, AVG(salary) FROM employees
GROUP BY department")
average_salary_per_department = cursor.fetchall()
print("Средняя зарплата в каждом отделе:", average_salary_per_
department)
```

```
Количество сотрудников в каждом отделе: [('IT', 2), ('Marketing', 1), ('Sales', 1)]
Средняя зарплата в каждом отделе: [('IT', 60000.0), ('Marketing', 55000.0), ('Sales', 60000.0)]

Process finished with exit code 0
```

Изображение 22.5.

Оператор ORDER BY используется для сортировки результатов запроса по одному или нескольким столбцам. По умолчанию сортировка выполняется по возрастанию (ASC). Для сортировки по убыванию используется ключевое слово DESC.

Листинг 22.10

```
# Сортировка сотрудников по зарплате по убыванию
cursor.execute("SELECT * FROM employees ORDER BY salary DESC")
sorted_employees = cursor.fetchall()
print("Сортировка по зарплате (по убыванию):", sorted_
employees)

# Сортировка сотрудников по отделу
cursor.execute("SELECT * FROM employees ORDER BY department,
name") sorted_employees_by_department = cursor.fetchall()
print("Сортировка по отделу:", sorted_employees_by_department)
```

Оператор **LIMIT** — устанавливает лимит на количество возвращаемых строк.

Оператор **OFFSET** — устанавливает смещение относительно начала результата.

Листинг 22.11

```
# Выбор 2 сотрудников, начиная со второго
cursor.execute("SELECT * FROM employees LIMIT 2 OFFSET 1")
limited_employees = cursor.fetchall()
print("Ограниченный вывод сотрудников:", limited_employees)
```

Оператор **LIKE** — осуществляет поиск по шаблону с использованием символов % и _.

Листинг 22.12

```
# Поиск сотрудников, чье имя начинается на 'И'
cursor.execute("SELECT * FROM employees WHERE name LIKE 'И%'")
employees_starting_with_i = cursor.fetchall()
print("Сотрудники, чье имя начинается на 'И':", employees_
starting_with_i)
```

Оператор **IN** — производит проверку наличия определенного значения в списке.

Листинг 22.13

```
# Выбор сотрудников из отделов 'IT' или 'Sales'
cursor.execute("SELECT * FROM employees WHERE department IN
('IT', 'Sales')")
it_or_sales_employees = cursor.fetchall()
print("Сотрудники из IT или Sales:", it_or_sales_employees)
```

Оператор **BETWEEN** — проверяет наличие определенного значения в указанном диапазоне.

Листинг 22.14

```
# Выбор сотрудников с зарплатой от 50000 до 60000
cursor.execute("SELECT * FROM employees WHERE salary BETWEEN
50000 AND 60000")
employees_in_salary_range = cursor.fetchall()
print("Сотрудники с зарплатой от 50000 до 60000:", employees_in_
salary_range)
```

Агрегатные функции

Агрегатные функции позволяют выполнять вычисления над набором значений и возвращать одно результирующее значение.

Наиболее часто используемые функции:

- COUNT() — возвращает количество строк.
- SUM() — возвращает сумму значений.
- AVG() — возвращает среднее значение.
- MIN() — возвращает минимальное значение.
- MAX() — возвращает максимальное значение.

Пример использования функций мы могли увидеть при знакомстве с оператором GROUP BY.

22.4. NoSQL базы данных

NoSQL — это тип баз данных, который отходит от традиционной табличной модели и предлагает более гибкие подходы к хранению данных. NoSQL расшифровывается как Not Only SQL, то есть такие базы не ограничиваются только языком SQL, но могут его использовать.

Ключевые особенности NoSQL в том, что такие базы могут использовать данные без строгой структуры. Данный тип хорошо подходят для обработки больших объемов данных. Поддерживает различные форматы, такие как документы, графы, пары "ключ-значение" и пр. Кроме того, NoSQL базы часто быстрее реляционных в определенных задачах.

Существует несколько основных типов NoSQL баз данных:

1. Документно-ориентированные — данные хранятся в виде документов (обычно JSON или BSON). Это удобный способ для хранения объектов с разными свойствами. Применяются в интернет-магазинах, блогах, системах управления контентом. Основные базы MongoDB, CouchDB.

Для работы с MongoDB в Пайтон используется библиотека **pymongo**, она не является встроенной, поэтому ее необходимо установить дополнительно. Подробнее об этом можно освежить в памяти в главе 16.

Листинг 22.15

```
from pymongo import MongoClient
# Подключение к MongoDB
client = MongoClient("mongodb://127.0.0.1:27017/")
# Создаём или подключаемся к базе данных
db = client["new_db_2"]
# Создаём или подключаемся к коллекции
collection = db["users"]

# Добавление документа
user = {"name": "Иван",
        "age": 30,
        "hobbies": ["reading", "cycling"]}
collection.insert_one(user)

# Чтение данных
for user in collection.find():
    print(user)
# Фильтрация данных
result = collection.find_one({"name": "Иван"})
print(result)
# Закрытие подключения
client.close()
```

- В данном примере мы подключаемся к локальной базе MongoDB, добавляем документ в коллекцию **users**, затем извлекаем все документы и выполняем фильтрацию по имени. Для тестирования кода вы можете установить сервер MongoDB, скачав бесплатную версию с официального сайта: https://www.mongodb.com/try/download/community.
- Затем запустите скачанный установщик и следуйте инструкциям мастера установки. По умолчанию MongoDB устанавливается в C:\Program Files\MongoDB\Server\.
- MongoDB хранит данные в определенном каталоге. По умолчанию в Windows это C:\data\db. Если этот каталог не существует, MongoDB не запустится. В таком случае создайте папку **data**, а внутри нее папку **db** на диске C.
- Далее откройте командную строку и перейдите в каталог **bin** установленной MongoDB (например, cd C:\Program Files\MongoDB\Server\[версия]\bin). Затем запустите сервер командой *mongod*.

- Если вы создали каталог для данных в другом месте, укажите путь к нему с помощью параметра *--dbpath*:

Листинг 22.16

```
mongod --dbpath D:\my_mongo_data
```

- Для работы с MongoDB вам понадобится клиент. Существует несколько вариантов:
  - **MongoDB Shell (mongosh)** — это командная строка для взаимодействия с MongoDB. Она устанавливается вместе с сервером. В Windows, находясь в каталоге **bin**, запустите ее командой *mongosh*.
  - **MongoDB Compass** — это графический интерфейс пользователя (GUI) для MongoDB. Он предоставляет удобный способ для просмотра и управления данными. Скачать его можно с сайта MongoDB. После установки MongoDB Compass при запуске предложит подключиться к серверу. По умолчанию адрес локального сервера mongodb://127.0.0.1:27017.
- Для остановки работы сервера в командной строке, где запущен mongod, нажмите Ctrl + C.

2. Базы данных "ключ-значение" — хранят данные в виде пары "ключ-значение". Они похожи на словари. Применяются для кэширования данных, хранения сессий и быстрого доступа к небольшим данным. Основные базы Redis и DynamoDB.

Листинг 22.17

```
import redis

# Подключение к Redis
r = redis.Redis(host='localhost', port=6379, decode_responses=True)
# Установка значения
r.set("name", "Демид")
# Получение значения
print(r.get("name")) # Результат: Демид
# Счетчик
r.incr("counter")
print(r.get("counter")) # 1
```

- Библиотека **redis** также устанавливается дополнительно. Здесь мы подключаемся к локальному серверу Redis. Сохраняем и извлекаем значение, используя ключ *name*. Затем увеличиваем значение счетчика с помощью встроенной функции.
- Стоит иметь в виду, что официально Redis не поддерживает Windows напрямую. Чаще всего он используется на Linux.

3. Графовые базы данных — ориентированы на хранение и обработку графовых структур. Используются в социальных сетях, анализе связей и построении рекомендаций. Основные базы Neo4j, ArangoDB.

Листинг 22.18

```
from neo4j import GraphDatabase
# Подключение к базе данных
driver = GraphDatabase.driver("bolt://localhost:7687",
                              auth=("neo4j", "password"))

def create_friend(tx, name1, name2):
    tx.run("CREATE (:Person {name: $name1})-[:FRIENDS_WITH]->(:Person
{name: $name2})",
           name1=name1, name2=name2)

def find_friends(tx, name):
    result = tx.run("MATCH (p:Person {name: $name})-[:FRIENDS_WITH]-
>(friend) RETURN friend.name",
                    name=name)
    for record in result:
        print(record["friend.name"])

with driver.session() as session:
    session.write_transaction(create_friend, "Андрей", "Демид")
    session.read_transaction(find_friends, "Андрей")

driver.close()
```

Здесь мы создаем связь между двумя пользователями, а затем извлекаем список друзей. Как и в предыдущих примерах, необходимо установить соответствующую библиотеку и скачать Neo4j с официального сайта:

https://neo4j.com/deployment-center/

- Для начинающих рекомендуется скачать и установить версию Neo4j Desktop. После ее установки в приложении можно создавать и запускать локальные экземпляры Neo4j Server с удобным графическим интерфейсом.
- Для начала нажмите кнопку **Add Database**, выберите **Local DBMS**, укажите имя базы данных и пароль (по умолчанию *neo4j*), затем нажмите **Create**. После создания базы данных нажмите кнопку **Start**.
- После запуска сервера откройте браузер и перейдите по адресу:

  http://localhost:7474/

  Вы увидите Neo4j Browser — это веб-интерфейс для работы с базой данных. Введите логин *neo4j* и пароль, который вы указали при создании базы данных (или *neo4j* по умолчанию, если не меняли). Также вам будет предложено сменить пароль при первом входе.
- Для работы с Neo4j используется язык запросов Cypher. Вы можете вводить Cypher-запросы и просматривать результаты в виде графов, таблиц или текста.
- Пример Cypher-запроса:

Листинг 22.19

```
CREATE (person:Person {name: 'Иван', age: 30})
CREATE (movie:Movie {title: 'Матрица', year: 1999})
CREATE (person)-[:WATCHED]->(movie)
RETURN person, movie
```

- Этот запрос создает узел типа Person с именем "Иван" и возрастом 30, а также узел типа Movie с названием "Матрица" и годом 1999, далее создается связь WATCHED между ними.

4. Столбцовые базы данных — ориентированы на хранение данных в виде столбцов. Применяются в системах аналитики, при обработке больших данных. Основные базы Apache Cassandra, HBase, NumPy.

Листинг 22.20

```
from cassandra.cluster import Cluster

# Подключение к кластеру Cassandra
cluster = Cluster(['127.0.0.1'])
```

```
session = cluster.connect()

# Создание ключевого пространства (аналог базы данных)
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS my_keyspace
    WITH replication = {'class': 'SimpleStrategy',
                        'replication_factor': '1'}
""")
# Используем ключевое пространство
session.set_keyspace("my_keyspace")

# Создание таблицы
session.execute("""
    CREATE TABLE IF NOT EXISTS users (
        user_id UUID PRIMARY KEY,
        name TEXT,
        age INT,
        email TEXT )
""")

# Добавление данных
import uuid
session.execute("""
    INSERT INTO users (user_id, name, age, email)
    VALUES (%s, %s, %s, %s)
""", (uuid.uuid4(), "Иван", 30, "ivan@example.com"))

session.execute("""
    INSERT INTO users (user_id, name, age, email)
    VALUES (%s, %s, %s, %s)
""", (uuid.uuid4(), "Демид", 25, "demid@example.com"))

# Чтение данных
rows = session.execute("SELECT * FROM users")
for row in rows:
    print(f"ID: {row.user_id}, Name: {row.name}, Age: {row.age},
Email: {row.email}")
# Закрытие подключения
cluster.shutdown()
```

Для взаимодействия с Cassandra используется библиотека cassandra-driver. В данном примере мы создаем соединение с кластером Cassandra, указав IP-адрес. Если вы работаете локально, используйте 127.0.0.1. Таблица **users** содержит четыре столбца: *user_id* (уникальный идентификатор), *name*, *age* и *email*.

- Далее, используя INSERT INTO, мы добавляем новые записи, а uuid.uuid4() генерирует уникальный идентификатор для каждого пользователя. С помощью SELECT * FROM users извлекаем все записи и результаты обрабатываем в цикле.

22.5. Транзакции

Транзакция (transaction) — это последовательность операций с базой данных, которые выполняются как единое целое. То есть либо все операции в транзакции будут успешно выполнены, либо ни одна из них не будет выполнена.

Например, мы переводим деньги с одного банковского счета на другой. Этот процесс состоит из нескольких шагов:

1. Списание суммы с нашего счета.
2. Зачисление этой же суммы на счет получателя.

Что случится, если после первого шага произойдет сбой в системе? Деньги будут списаны с нашего счета, но не поступят на счет получателя. Это недопустимая ситуация. Транзакция в базе данных является тем механизмом, который гарантирует, что такие ситуации не произойдут.

Обе операции (списание и зачисление) должны быть объединены в одну транзакцию. Если произойдет сбой после списания, система должна откатить транзакцию, то есть вернуть состояние базы данных к исходному, как будто списания и не было.

Транзакции в базах данных должны удовлетворять свойствам ACID:

- **Atomicity (Атомарность)** — транзакция является неделимой операцией. Либо все ее части выполняются успешно, либо ни одна. Как в примере с банковским переводом: либо деньги списываются и зачисляются, либо ничего не происходит.
- **Consistency (Согласованность)** — транзакция переводит базу данных из одного согласованного состояния в другое. Согласованное состояние — это состояние, в котором выполняются все правила и ограничения

целостности базы данных (например, ограничения на значения, связи между таблицами).

- **Isolation (Изолированность)** — параллельно выполняющиеся транзакции не должны влиять друг на друга. Каждая транзакция работает так, как будто она выполняется одна. Это предотвращает ситуации, когда одна транзакция читает незавершенные изменения другой транзакции.
- **Durability (Надежность)** — после успешного завершения транзакции, изменения, внесенные ею в базу данных, сохраняются навсегда и не могут быть потеряны из-за сбоев в системе (например, отключение питания).

Листинг 22.21

```
import sqlite3
conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()

try:
    # Начало транзакции
    conn.execute("BEGIN TRANSACTION")

    # Операции в рамках транзакции
    # Списание со счёта 1
    cursor.execute("UPDATE accounts SET balance = balance - 100 WHERE
account_id = 1")
    # Зачисление на счёт 2
    cursor.execute("UPDATE accounts SET balance = balance + 100 WHERE
account_id = 2")

    # Подтверждение успешной транзакции
    conn.commit()
    print("Транзакция выполнена успешно.")

except sqlite3.Error as e:
    # Откат транзакции в случае ошибки
    conn.rollback()
    print(f"Ошибка при выполнении транзакции: {e}")

conn.close()
```

- В примере строка *conn.commit* подтверждает транзакцию. Все изменения, внесенные в рамках транзакции, сохраняются в базе данных.

- Строка *conn.rollback* откатывает транзакцию, то есть все изменения, внесенные в рамках транзакции, отменяются.
- Блок **try...except** используется для обработки возможных ошибок. Если во время выполнения транзакции что-то пойдет не так, будет выполнен откат.

Таким образом, транзакции критически важны для сохранения целостности данных во многих случаях. Например, для финансовых операций, покупки чего-либо в Интернете, бронирования билетов или гостиниц, для обновления связанных данных в нескольких таблицах. А также когда при удалении пользователя необходимо удалить все связанные с ним записи в других таблицах.

Существуют различные уровни изоляции транзакций, которые определяют, насколько сильно транзакции изолированы друг от друга. Более высокие уровни изоляции обеспечивают большую надежность, но могут снижать производительность.

- **Read Uncommitted** — самый низкий уровень изоляции. Транзакция может читать незавершенные изменения других транзакций ("грязное чтение").
- **Read Committed** — транзакция может читать только те изменения, которые были подтверждены другими транзакциями.
- **Repeatable Read** — транзакция гарантирует, что при повторном чтении одних и тех же данных она получит те же самые значения.
- **Serializable** — самый высокий уровень изоляции. Транзакции выполняются последовательно, одна за другой.

Выбор уровня изоляции зависит от требований конкретного приложения.

22.6. Безопасность баз данных

Безопасность баз данных — это комплекс мер, направленных на защиту данных, хранящихся в базе данных, от несанкционированного доступа, изменения, удаления или повреждения.

Существует несколько типов внутренних и внешних угроз:

- **Несанкционированный доступ** — доступ к данным лиц, не имеющих на это прав.
- **SQL-инъекции** — внедрение вредоносного SQL-кода в запросы к базе данных.
- **Утечка данных** — случайное или специальное раскрытие конфиденциальной информации.
- **DoS-атаки (Denial of Service)** — атаки, направленные на вывод базы данных из строя, делающие ее недоступной для пользователей.
- **Внутренние угрозы** — угрозы, исходящие от сотрудников, имеющих доступ к базе данных.

Очевидно, что безопасность важна, многие современные приложения зависят от баз данных. Их компрометация может привести к серьезным последствиям, поэтому существует несколько методов обеспечения безопасности.

Параметризованные запросы (Prepared Statements)

Это самый эффективный способ защиты от SQL-инъекций. Вместо непосредственного конструирования SQL-запроса из пользовательского ввода используются параметры, которые передаются в запрос отдельно.

Плохой пример, уязвимый для SQL-инъекций:

Листинг 22.22

```
username = input("Введите имя пользователя: ")
query = "SELECT * FROM users WHERE username = '" + username + "'"
cursor.execute(query)
```

Если пользователь введет **'; DROP TABLE users; --**, то будет выполнен запрос SELECT * FROM users WHERE username = ''; DROP TABLE users; --, что приведет к удалению таблицы users.

Хороший пример, с использованием параметризованных запросов:

Листинг 22.23

```
username = input("Введите имя пользователя: ")
query = "SELECT * FROM users WHERE username = ?"
cursor.execute(query, (username,))
```

Теперь введенное пользователем значение будет интерпретировано как параметр, а не как часть SQL-кода, что предотвратит SQL-инъекцию.

Использование ORM (Object-Relational Mapping)

ORM — это подход в программировании, который позволяет взаимодействовать с реляционными базами данных, используя объекты из объектно-ориентированных языков программирования. Этот метод позволяет работать с данными в виде объектов, упрощая процесс управления базами данных и снижая необходимость в написании сложных SQL-запросов.

Листинг 22.24

```
# Пример с SQLAlchemy (упрощенно)
user = session.query(User).filter_by(username=username).first()
```

Валидация и очистка пользовательского ввода

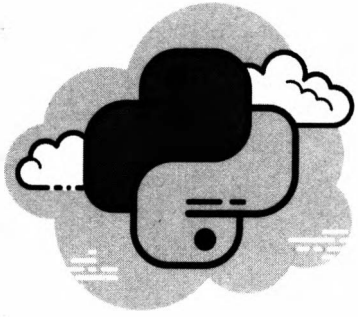
Перед использованием пользовательского ввода в запросах к базе данных необходимо проверять его на корректность и очищать от потенциально опасных символов. То есть проверки типа: является ли введенное значение числом (определенным типом данных), какова длина строк и аналогичные. Также следует экранировать специальные символы.

Контроль доступа (Authentication и Authorization)

Еще один вариант проверки пользователя, имеет ли он права доступа (админ, модератор) или является простым пользователем с ограниченными правами.

22.6.1. Практические задания

1. Создайте базу данных SQLite с таблицей **students**, содержащей поля *id*, *name* и *age*. Заполните таблицу данными о трех студентах. Затем выведите все записи из таблицы.
2. Запросите имя студента у пользователя. Найдите запись в таблице **students**, соответствующую этому имени, и выведите ее.



Ответы к заданиям

3.3.1. Ответы:

1.

```
>>> abs(-5.27)
5.27
```

2.

```
>>> round(3.14159, 2)
3.14
```

3.

```
>>> int('684')
684
```

4.

```
>>> max(3, 7, 2, 9, 5)
9
>>> min(3, 7, 2, 9, 5)
2
```

5.

```
>>> sum([5, 4, 32, 18, 841])
900
```

6.

```
>>> pow(2,5)
32
```

3.5.1. Ответы:

1.

Листинг 3.3

```
>>>name = 'Иван'
'Иван'
```

2.

Листинг 3.4

```
>>> x=5
>>> y=7
>>> z=9
>>> sum([x,y,z])
21
```

3.

Листинг 3.5

```
>>> min(x,y,z)
5
>>> max(x,y,z)
9
```

4.

Вариант 1. Замена с помощью третьей переменной:

Листинг 3.6

```
>>> z=x   # z сохраняет значение переменной x
>>> x=y   # x получает значение переменной y
>>> y=z   # y получает значение переменной z
>>> x     # Проверяем результаты
7
>>> y
5
```

Вариант 2. С помощью математических операций:

Листинг 3.7

```
>>> x=5   # Значения x и y по умолчанию
>>> y=7
>>> x=x+2  # Переменная x сохраняет новое значение, равное x+2 (5+2)
>>> y=y-2 # Аналогично для y
>>> x     # Проверяем результаты
7
>>> y
5
```

Вариант 3. Множественное присвоение:

Листинг 3.8

```
>>> x=5          # Значения x и y по умолчанию
>>> y=7
>>> x,y = y,x   # Множественное присвоение
>>> x           # Проверяем результаты
7
>>> y
5
```

4.3.1. Ответы:

1.

Листинг 4.8

```
age=int(input('Сколько вам полных лет? '))
login=input('Придумайте логин ')
password=input('Придумайте пароль ')
print(f"Здравствуйте {login}, вам {age} лет, ваш пароль {password}")
```

2.

Преобразуем вводимые данные в вещественные числа, так как стороны четырехугольника могут иметь разную длину, в том числе с дробной частью.

Листинг 4.9

```
side_1 = float(input('Укажите длину 1-ой стороны: '))
side_2 = float(input('Укажите длину 2-ой стороны: '))
side_3 = float(input('Укажите длину 3-ей стороны: '))
side_4 = float(input('Укажите длину 4-ой стороны: '))
perimeter = side_1 + side_2 + side_3 + side_4
print(f"Периметр четырехугольника равен: {perimeter}")
```

Вместе с тем, подсчет можно проводить при выводе:

Листинг 4.10

```
print(f"Периметр равен: {side_1 + side_2 + side_3 + side_4}")
```

5.1.1. Ответы:

1.

Листинг 5.7

```
apple = int(input("Введите количество яблок: "))
box = int(input("Введите количество коробок: "))
result = apple // box # Делим без остатка яблоки на коробки
print(result)
```

2.

Листинг 5.8

```
apple = int(input("Введите количество яблок: "))
box = int(input("Введите количество коробок: "))
result = apple % box # Остаток от деления
print(result)
```

3.

Листинг 5.9

```
number = int(input("Введите целое число: "))
print(number % 10) # Узнаем остаток от числа, являющийся последней цифрой
```

4.

Листинг 5.10

```
number = int(input("Введите целое число: "))
print(number % 100 // 10)
# Сначала получили остатком от деления две последние цифры (%100)
# Затем разделили без остатка, отбросив последнюю (//10)
```

5.

Листинг 5.11

```
number = int(input("Введите трехзначное число: "))
x = number // 100 # Узнаем первое число, отбрасывая два последних
y = number // 10 % 10 # Узнали вторую цифру, отбросив третью и первую
```

```
z = number % 10 # Узнали третью цифру
print (x + y + z) # Суммировали и вывели результат
```

5.2.5. Ответы:

1.

Листинг 5.29

```
import math # Импортировали модуль
total_apples = int(input("Введите количество яблок: "))
box_capacity = 5 # Вместимость каждой коробки
boxes_needed = math.ceil(total_apples / box_capacity)
print(boxes_needed)
```

В четвертой строке, в скобках мы делим общее количество яблок (например, 23) на вместимость коробок (5), то есть 23 / 5 = 4,6. Далее функция ***ceil*** округляет это число в большую сторону. Таким образом, нам понадобится 5 коробок, 4 из которых будут полными, а последняя содержит 3 яблока.

2.

Листинг 5.30

```
import math
total_apples = int(input("Введите количество яблок: "))
box_capacity = 5 # Вместимость каждой коробки
full_boxes = math.floor(total_apples / box_capacity)
print(full_boxes)
```

В четвертой строке в скобках делим общее количество яблок (например, 32) на вместимость коробок (5), то есть 32 / 5 = 6,4. Затем функция ***floor*** округляет это число в меньшую сторону.

3.

Листинг 5.31

```
import math
shop_1 = int(input("Сколько заказал 1-й магазин?: "))
shop_2 = int(input("Сколько заказал 2-й магазин?: "))
shop_3 = int(input("Сколько заказал 3-й магазин?: "))
```

```
box_1 = math.ceil(shop_1 / 5) # Заказ магазина делим на
box_2 = math.ceil(shop_2 / 5) # вместимость коробки
box_3 = math.ceil(shop_3 / 5) # и округляем в большую сторону
print(box_1 + box_2 + box_3) # Сложили и вывели результат
```

Данная задача решается аналогично первой, только на этот раз пользователь вводит три значения вместо одного. В строках 5–7 мы делим заказ каждого магазина на количество коробок, а затем округляем в большую сторону, чтобы узнать, сколько коробок нужно для каждого отдельного магазина. Затем мы подсчитываем общее количество коробок.

Обратите внимание, что если сразу сложить все заказы вместе result = (box_1 + box_2 + box_3) и высчитать общее количество коробок math. ceil(result / 5), количество коробок будет меньше, но заказы магазинов смешаются.

5.3.6. Ответы:

1.

Листинг 5.40

```
number = int(input("Введите целое число: "))
print(number % 5 == 0)
```

Все числа, разделенные на 5 без остатка, являются кратными пяти и дают ответ True.

2.

Листинг 5.41

```
number = int(input("Введите целое число: "))
print(number > 9 and number < 100)
```

В диапазон двузначных чисел входят числа от 10 до 99. Если введенное число больше девяти и меньше ста, оно является двузначным. Также правильным решением будет сравнение number >= 10 and number <= 99.

6.5.1. Ответы:

1.

Листинг 6.14

```
characters = int(input("Сколько персонажей тянуло репку в сказке
про репку?: "))
if characters == 7:
    print("Совершенно верно!")
else:
    print("Ответ неверный, попробуйте еще раз")
```

Правильный ответ: 7 персонажей (Дедка, Бабка, Внучка, Жучка, Кошка, Мышка).

2.

Листинг 6.15

```
number = int(input("Введите число: "))
result = "Это четное число" if number % 2 == 0 else "Это нечетное
число"
print(result)
```

3.

Листинг 6.16

```
points = int(input("Введите количество баллов: "))
# Определяем оценку в зависимости от баллов
if points >= 91:
    grade = 5
elif points >= 81:
    grade = 4
elif points >= 71:
    grade = 3
elif points >= 61:
    grade = 2
else:
    grade = 1
# Выводим результат на экран
print(f"Ваша оценка: {grade}")
```

В переменную *grade* сохраняем полученную оценку, чтобы потом можно было использовать эту информацию в других частях кода.

4.

Листинг 6.17

```
# Логин и пароль от профиля
correct_login = "admin"
correct_password = "password123"

# Предлагаем ввести логин
login = input("Введите логин: ")

# Проверка ввода
if login == correct_login: # Сравниваем введенный логин со старым
    password = input("Введите пароль: ") # Если логин верный, вводим
пароль
    if password == correct_password: # Если совпадают — успешный вход
        print("Вы успешно авторизованы!")
    else:
        print("Неверный пароль.")
else:
    print("Неверный логин.")
```

7.2.1. Ответы:

1.

Листинг 7.24

```
text = input("Введите любой текст: ")
print("Вы ввели", len(text), "символов")
```

2.

Листинг 7.25

```
login = input("Введите логин (минимум 3, максимум 10 символов): ")
# Если длина логина не менее 3 символов и не более 10:
if 3 <= len(login) <= 10:
    print("Ваш логин сохранен")
else:
    print("Неверная длина строки. Попробуйте снова.")
```

7.4.1. Ответы:

1.

Листинг 7.74

```
# Запрашиваем у пользователя текст
user_input = input("Введите текст: ")
# Проверяем, состоит ли текст только из букв
if user_input.isalpha(): # Если True
    print("Строка состоит только из букв.")
else: # Иначе
    print("Строка содержит не только буквы.")
```

2.

Листинг 7.75

```
# Запрашиваем у пользователя число
user_input = input("Введите число: ")
# Проверяем, состоит ли значение только из цифр
if user_input.isdigit(): # Если True
    print("Строка состоит только из цифр.")
else: # Иначе
    print("Строка содержит не только цифры.")
```

3.

Листинг 7.76

```
# Запрашиваем у пользователя данные
login = input("Придумайте логин, состоящий только из букв и цифр: ")
# Проверяем, состоит ли логин только из букв и цифр
if login.isalnum(): # Если True
    print("Ваш логин сохранен.")
else: # Иначе
    print("Логин содержит запрещенные символы.")
```

4.

Листинг 7.77

```
text = "Мой кот очень любит играть с клубком. Но иногда кот спит весь день."
new_text = text.replace("кот", "пес") # Заменяем старое слово на новое
print(new_text)
```

5.

Листинг 7.78

```
email = input("Введите email адрес: ")
if email.find("@") >= 0:
    if email.endswith(".com", 4) or email.endswith(".ru", 4):
        print("Это корректный email адрес")
    else:
        print("Email адрес не корректен")
else:
    print("Email адрес не корректен")
```

После ввода email-адреса происходит проверка на наличие символа "@". Если символ найден, метод ***find*** возвращает его индекс (положение в строке), который всегда будет больше или равен 0. Если символ не найден, возвращается -1, что меньше нуля, а значит проверка будет провалена (False).

Затем методом ***endswith*** проверяется, заканчивается ли строка на .com или .ru, начиная с четвертого символа. Аргумент 4 указывает, с какой позиции начинать проверку. Это делается для того, чтобы исключить случаи, когда точка перед .com или .ru является частью локальной части или введен слишком короткий адрес. Например, a@a.com.

6.

Листинг 7.79

```
text = input("Введите любой текст: ")
words = text.split() # Разбиваем текст на слова по пробелам
count_words = len(words) # Считаем количество слов
print("Количество слов в тексте:", count_words)
```

7.5.3. Ответы:

1.

Листинг 7.106

```
import re
text = "Иванов Иван Иванович: 89123456789"
pattern = r"\d+" # Ищем одну или более цифр подряд
matches = re.findall(pattern, text)
print(matches) # Результат: ['89123456789']
```

2.

Листинг 7.107

```
import re
text = "Это событие произошло: 17-07-1777"
# Ищем последовательность в формате ДД-ММ-ГГГГ
date_pattern = r"\d{2}-\d{2}-\d{4}"
match = re.search(date_pattern, text)
if match:
    print("Найденная дата:", match.group())
else:
    print("Дата не найдена")
```

3.

Листинг 7.108

```
import re
text = "Наш кот любит сметану. Иногда кот спит на печи."
new_text = re.sub("кот", "пес", text)
print(new_text)
# Результат: Наш пес любит сметану. Иногда пес спит на печи.
```

4.

Листинг 7.109

```
import re
email = input("Введите ваш email: ")
pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z]+$"
if re.match(pattern, email):
    print("Email-адрес корректен")
else:
    print("Email-адрес некорректен")
```

- **^** — проверяем на начало строки.
- **[a-zA-Z0-9_.+-]+** — указываем диапазоны букв, цифр, точек, подчеркиваний, плюсов и минусов. Это соответствует локальной части email-адреса (например, "user").
- **@** — символ разделяет локальную часть и домен.
- **[a-zA-Z0-9-]+** — одна или более букв, или цифр, или дефисов. Это соответствует имени домена (например, "example").

- \. — точка (экранирована обратным слешем, так как точка имеет специальное значение в регулярных выражениях).
- [a-zA-Z]+ — указываем диапазоны букв, которые должны совпадать с доменной зоной (например, "com").
- $ — проверяем на окончание строки (далее не должно быть символов).

7.6.1. Ответ:

Листинг 7.124

```
with open(r"C:\Users\ZET\Test_Project\new_file.txt", 'a',
encoding="utf-8") as new_file:
    new_file.write(f"\nФайл был отредактирован")
```

8.2.1. Ответы:

1.

Листинг 8.26

```
spisok = [3, 7, 2, 9, 1]
max_number = max(spisok)
print(f"Максимальное число: {max_number}")
min_number = min(spisok)
print(f"Минимальное число: {min_number}")
```

2.

Листинг 8.27

```
numbers = [3, 7, 2, 9, 1, 3, 7, 8, 9, 11, 15, 7]
count_7 = numbers.count(7)
print(f"Число 7 встречается {count_7} раз(а)")
```

3.

Листинг 8.28

```
numbers = [54, 13, 24, 58, 61, 7, 1, 73, 8]
numbers.sort(reverse=True)
print(f"Отсортированный список: {numbers}")
# Ответ: [73, 61, 58, 54, 24, 13, 8, 7, 1]
```

4.

Листинг 8.29

```
spisok_1 = ['стол', 'стул', 'диван']
spisok_2 = ['ковер', 'шторы', 'торшер']
combined_spisok = spisok_1 + spisok_2
print(f"Объединенный список: {combined_spisok}")
```

5.

Листинг 8.30

```
users = ['admin', 'moderator', 'user'] # Список с разрешенным доступом
your_status = 'guest' # Статус пользователя
if your_status in users: # Если статус пользователя в списке
    print('Вы можете оставлять комментарии')
else:
    print('Вам запрещено оставлять комментарии')
```

6.

Листинг 8.31

```
spisok = ["переменная", "строка", "число", "функция",
"аргумент", "оператор"]
srez = spisok[2:5]
print(srez) # ['число', 'функция', 'аргумент']
```

7.

Листинг 8.32

```
numbers = [18, 39, 24, 16, 8, 2]
numbers.insert(2,7)
print(numbers) # [18, 39, 7, 24, 16, 8, 2]
```

8.4.1. Ответы:

1.

Листинг 8.53

```
spisok = ["ананас", "манго", "банан", "манго", "ананас"]
spisok = list(set(spisok))
print(spisok) # Результат: ['ананас', 'банан', 'манго']
```

Во второй строке список преобразуется во множество set(spisok). Затем результат множества преобразуется обратно в список list(set(...)). Далее полученный результат сохраняется в переменной spisok = ...

2.

Листинг 8.54

```
nabor = {"ананас", "манго", "банан"} # Первоначальное множество
nabor.update(["апельсин", "абрикос"]) # Добавляем элементы
nabor.discard("ананас") # Удаляем элемент
print(nabor) # Результат: {'манго', 'банан', 'апельсин', 'абрикос'}
```

8.7.1.Ответы:

1.

Листинг 8.79

```
slovar = dict.fromkeys(["apple", "orange", "apricot"], 70)
del slovar["apricot"] # Удалили ключ и его значение
slovar["apple"] = 50 # Изменили первые два элемента
slovar["orange"] +=20
print(slovar) # Результат: {'apple': 50, 'orange': 90}
```

2.

Листинг 8.80

```
slovar = {"apple": 22, "orange": 24, "apricot": 26}
if "orange" in slovar: # Если ключ есть в словаре:
    print(slovar["orange"]) # Показали его значение
else: # Если ключа нет в словаре:
    slovar["orange"] = 28 # Присваиваем новое значение
    print(slovar["orange"])
```

9.1.1. Ответы:

1.

Листинг 9.4

```
number = 1 # Переменная со значением 1
while number < 11: # Пока переменная меньше 11
```

```
    print(number) # Выводим на экран текущее значение переменной
    number += 2 # Увеличиваем значение на 2 после каждого прохода
print("Цикл завершен") # Код, который выполнится по завершении цикла
```

2.

Листинг 9.5

```
number = 15 # Переменная со значением 15
while number > 4: # Пока переменная больше 4
    print(number) # Выводим на экран текущее значение переменной
    number -= 1 # Уменьшаем значение на 1
print("Цикл завершен")
```

3.

Листинг 9.6

```
spisok = ["кот", "пес", "кот", "кит", "кот", "ерш", "кот"]
while "кот" in spisok: # Пока кот в списке
    spisok.remove("кот") # Удаляем первого встреченного кота
print(spisok)
```

4.

Листинг 9.7

```
user_password = "admin123" # Старый пароль
password = input("Введите пароль: ") # Переменная, хранящая введенный
пароль
while password != user_password: # Пока пароли не совпадают
    print("Пароль неверный, попробуйте еще раз")
    password = input()
print("Это правильный пароль. Вход выполнен успешно.")
```

9.4.1. Ответы:

1.

Листинг 9.29

```
for i in range(10,100):
  print(i)
```

Вывели двузначные числа от 10 до 99 (100 не включается в ранжирование).

2.

Листинг 9.30

```
number_1 = int(input("Введите число: "))
number_2 = int(input("Введите число большее, чем предыдущее: "))
while number_1 >= number_2: # Пока первое число больше второго или
равно ему
    number_2 = int(input("Введите число большее, чем предыдущее: "))
for i in range(number_1, number_2): # Выводим числа в диапазоне
    print(i)
```

3.

Листинг 9.31

```
spisok = ["кот", "пес", "кот", "кит", "пес", "ерш", "кит"]
new_spisok = [] # Новый список без дублей
for i in spisok: # Проходимся по каждому элементу списка
    if i not in new_spisok: # Если элемент не в новом списке,
        new_spisok.append(i) # добавляем элемент в список
print(new_spisok) # Результат: ['кот', 'пес', 'кит', 'ерш']
```

4.

Листинг 9.32

```
z = "x" # Переменная, хранящая значение "x"
for i in range(4): # Внешний цикл, формирующий новые строки
    for j in range(5): # Вложенный цикл, формирующий столбцы
        print(z, end=" ") # Каждый столбец завершается пробелом
    print() # Переход на новую строку
# Результат:
# x x x x x
# x x x x x
# x x x x x
# x x x x x
```

10.10.1. Ответы:

1.

Листинг 10.55

```
def largest_number(x, y): # Функция находит наибольшее число
    if x > y: # Если "x" больше "y"
        print(f"{x} больше {y}")
    elif y > x: # Если "y" больше "x"
        print(f"{y} больше {x}")
    else:
        print("Числа равны")

x = int(input("Введите первое число: "))
y = int(input("Введите второе число: "))
largest_number(x, y) # Вызвали функцию сравнения
```

2.

Листинг 10.56

```
def data(password, login="Пользователь", *args):
    print(f"Ваш пароль: {password}")
    print(f"Ваш логин: {login}")
    print("Ваши увлечения:", args)

# Запрашиваем у пользователя ввод данных
password = input("Введите пароль: ")
name = input("Введите логин: ")
args = input("Напишите ваши увлечения через запятую: ").split(',')
# Вызываем функцию и передаем аргументы
data(password, name, *args)
```

3.

Листинг 10.57

```
fruits = ['апельсин', 'банан', 'манго', 'вишня']
for i, fruit in enumerate(fruits, 1):
    print(i, fruit)
```

4.

Листинг 10.58

```
numbers = [1, 2.5, 3.1, 4, 5.0]

# Проверяем каждое число
all_numbers = all(isinstance(x, int) for x in numbers)
```

```
# Выводим результат
if all_numbers:
    print("Список состоит из целых чисел")
else:
    print("В списке не все числа целые")
```

12.4.1. Ответы:

1.

Листинг 12.12

```
def divide_numbers():
    try:
        num1 = int(input("Введите первое число: "))
        num2 = int(input("Введите второе число: "))
        result = num1 / num2
        print(f"Результат деления: {result}")
    except ZeroDivisionError:
        print("Ошибка: деление на ноль невозможно.")
    except ValueError:
        print("Ошибка: введено не целое число.")

# Запуск функции
divide_numbers()
```

2.

Листинг 12.13

```
def convert_to_number():
    user_input = input("Введите число: ")
    try:
        number = float(user_input)
        print(f"Успешное преобразование! Число: {number}")
    except ValueError:
        print("Ошибка: введенное значение нельзя преобразовать в число.")

# Запуск функции
convert_to_number()
```

3.

Листинг 12.14

```
def read_file():
    file_name = input("Введите имя файла: ")
    try:
        with open(file_name, 'r') as file:
            content = file.read()
            print("Содержимое файла:")
            print(content)
    except FileNotFoundError:
        print("Ошибка: файл не найден.")
    except Exception as e:
        print(f"Произошла непредвиденная ошибка: {e}")

# Запуск функции
read_file()
```

13.5.1. Ответы:

1.

Листинг 13.18

```
class Character:
    def __init__(self, name, age, profession):
        self.name = name # Имя персонажа
        self.age = age # Возраст
        self.profession = profession # Профессия

    def get_info(self):
        # Метод для получения информации о персонаже
        return f"Имя: {self.name}, Возраст: {self.age}, Профессия:
{self.profession}"

# Создание объектов класса Character
vakula = Character("Вакула", 22, "Кузнец")
oksana = Character("Оксана", 20, "Домохозяйка")

# Вывод информации о персонажах
print(vakula.get_info()) # Имя: Вакула, Возраст: 22, Профессия:
Кузнец
print(oksana.get_info()) # Имя: Оксана, Возраст: 20, Профессия:
Домохозяйка
```

2.

Листинг 13.19

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width # Ширина
        self.height = height # Высота

    def area(self):
        # Метод для расчета площади
        return self.width * self.height

    def perimeter(self):
        # Метод для расчета периметра
        return 2 * (self.width + self.height)

# Создание объектов класса Rectangle
rect1 = Rectangle(10, 5)
rect2 = Rectangle(21, 17)

# Вывод площади и периметра
print(f"Площадь 1-го прямоугольника: {rect1.area()}") # 50
print(f"Периметр 1-го прямоугольника: {rect1.perimeter()}") # 30
print(f"Площадь 2-го прямоугольника: {rect2.area()}") # 357
print(f"Периметр 2-го прямоугольника: {rect2.perimeter()}") # 76
```

14.5.1. Ответы:

1.

Листинг 14.20

```
# Декоратор для логирования
def log_decorator(func):
    def obertka(*args, **kwargs):
        print(f"Вызов функции '{func.__name__}' с аргументами: {args},
{kwargs}")
        result = func(*args, **kwargs) # Выполняем функцию
        print(f"Результат функции '{func.__name__}': {result}")
        return result
    return obertka
```

```
@log_decorator
def add_numbers(x, y):
    # Функция складывает два числа
    return x + y

# Вызываем функцию для сложения значений
add_numbers(33, 55)
```

2.

Листинг 14.21

```
# Декоратор для проверки прав доступа
def access_control(status):
    def decorator(func):
        def obertka(user_status, *args, **kwargs):
            # Если права соответствуют, выполняем функцию
            if user_status == status:
                return func(*args, **kwargs)
            else:
                print("Доступ запрещен")
        return obertka
    return decorator

@access_control("admin")
def view_secret_data():
    # Функция выводит секретные данные
    print("Секретные данные: пароль - 123456")

# Вызываем функцию, где в качестве аргумента статус пользователя
view_secret_data("admin") # Секретные данные: пароль — 123456
view_secret_data("user") # Доступ запрещен
```

15.10.1. Ответы:

1.

Листинг 15.30

```
class Counter_char:
    count = 0 # Статический атрибут для подсчета

    def __new__(cls, *args, **kwargs):
        cls.count += 1
        print(f"Создан объект №{cls.count}")
```

```
        return super().__new__(cls)

    def __init__(self, name):
        self.name = name

# Создаем объекты
Ulrik = Counter_char("Ульрик")
Alvor = Counter_char("Алвор")
Lukan = Counter_char("Лукан")
# Результат:
# Создан объект №1
# Создан объект №2
# Создан объект №3
```

2.

Листинг 15.31

```
class Box:
    def __init__(self):
        # Инициализируем список предметов
        self.items = []

    def add(self, item):
        # Добавляем элемент в коробку
        # с помощью стандартного метода для списков
        self.items.append(item)

    def __getitem__(self, index):
        # Доступ к элементу по индексу
        return self.items[index]

    def __setitem__(self, index, value):
        # Изменяем элемент по индексу
        self.items[index] = value

    def __len__(self):
        # Возвращаем количество элементов
        return len(self.items)

pack = Box() # Создали экземпляр класса
pack.add("Вишня") # Добавили три предмета
pack.add("Груша")
pack.add("Апельсин")

print(len(pack)) # Посчитали: 3
```

```
print(pack[1]) # Отобразили элемент с индексом 1
pack[1] = "Яблоко" # Изменили элемент с индексом 1
print(pack[1]) # Проверяем изменения
```

16.6.1. Ответы:

1.

Листинг 16.22

```
class Base: # Базовый класс
    def method(self, x, y):
        return x + y # Метод складывает два числа

class Subclass_1(Base): # Подкласс_1
    def method(self, x, y):
        return x - y # Метод вычитает

class Subclass_2(Base): # Подкласс_2
    def method(self, x, y):
        return x * y # Метод перемножает

# Создаем объекты
object_1 = Subclass_1()
object_2 = Subclass_2()

# Вызываем метод
print(object_1.method(2,3)) # -1
print(object_2.method(2,3)) # 6
```

2.

Листинг 16.23

```
class Employee: # Базовый класс
    def __init__(self, name, salary):
        self.name = name # Имя
        self.salary = salary # ЗП

    def info(self): # Метод для вывода информации
        print(f"Имя: {self.name}, Зарплата: {self.salary}")

class Manager(Employee): # Подкласс
    def __init__(self, name, salary, office):
        super().__init__(name, salary) # Вызываем конструктор базового
```

```
класса
        self.office = office

    def info(self):
        super().info() # Вызываем метод info из базового класса
        print(f"Офис: {self.office}")

# Создаем сотрудника
manager_1 = Manager("Иван", 170000, "Отдел разработки")
# Проверяем
manager_1.info()
# Имя: Иван, Зарплата: 170000
# Офис: Отдел разработки
```

17.5.1. Ответы:

1.

Листинг 17.22

```
# Файл shapes.py
import math # Импортируем модуль

def circle_area(radius):
    """Возвращает площадь круга"""
    return math.pi * radius ** 2

def rectangle_area(width, height):
    """Возвращает площадь прямоугольника"""
    return width * height
```

В основной скрипт импортируем обе функции через запятую:

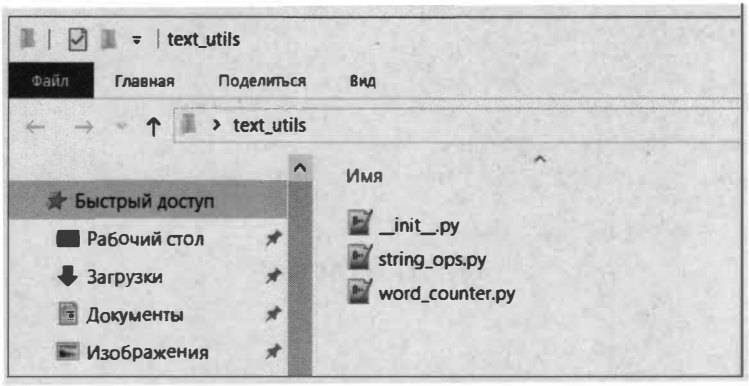
Листинг 17.23

```
# Файл main.py
# Импортируем функции из модуля geometry.shapes
from geometry.shapes import circle_area, rectangle_area

# Вычисляем площадь круга
radius = float(input("Введите радиус круга: "))
print(f"Площадь круга: {circle_area(radius):.2f}")

# Вычисляем площадь прямоугольника
width = float(input("Введите ширину прямоугольника: "))
```

```
height = float(input("Введите высоту прямоугольника: "))
print(f"Площадь прямоугольника: {rectangle_area(width, height):.2f}")
```

2.

Структура пакета:



Изображение 17.16.

Содержимое файла string_ops.py:

Листинг 17.24

```
# Файл string_ops.py
def reverse_string(s):
    """Возвращает строку в развернутом виде"""
    return s[::-1]
```

Содержимое файла word_counter.py:

Листинг 17.25

```
# Файл word_counter.py
def count_words(text):
    """Возвращает количество слов в строке"""
    return len(text.split())
```

Файл __init__.py (инициализация пакета):

Листинг 17.26

```
# Файл __init__.py
from .string_ops import reverse_string
from .word_counter import count_words
```

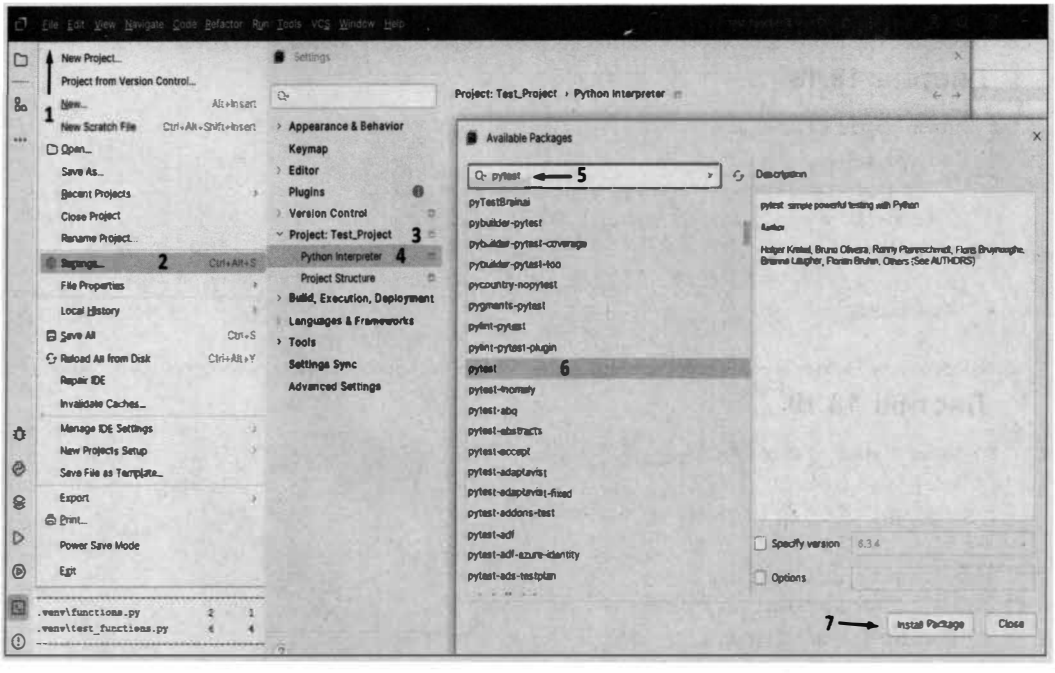
Основной скрипт:

Листинг 17.27

```
# Файл main.py
# Импортируем весь пакет text_utils
import text_utils

# Используем функцию reverse_string
text = input("Введите текст: ")
reversed_text = text_utils.reverse_string(text)
print(f"Текст в обратном порядке: {reversed_text}")

# Используем функцию count_words
word_count = text_utils.count_words(text)
print(f"Количество слов в тексте: {word_count}")
```

18.6.1. Ответы:

1.



Изображение 18.7.

2.

Листинг 18.16

```
# Файл functions.py
def multipl(a, b):
    """Перемножаем два числа"""
    return a * b
```

Листинг 18.17

```
# Файл test_functions.py
from functions import multipl

def test_multipl():
    '''Проверяем корректность перемножения'''
    assert multipl(2, 2) == 4
    assert multipl(-1, 1) == -1
    assert multipl(0, 0) == 0
    assert multipl(2.5, 2.5) == 6.25
```

3.

Листинг 18.18

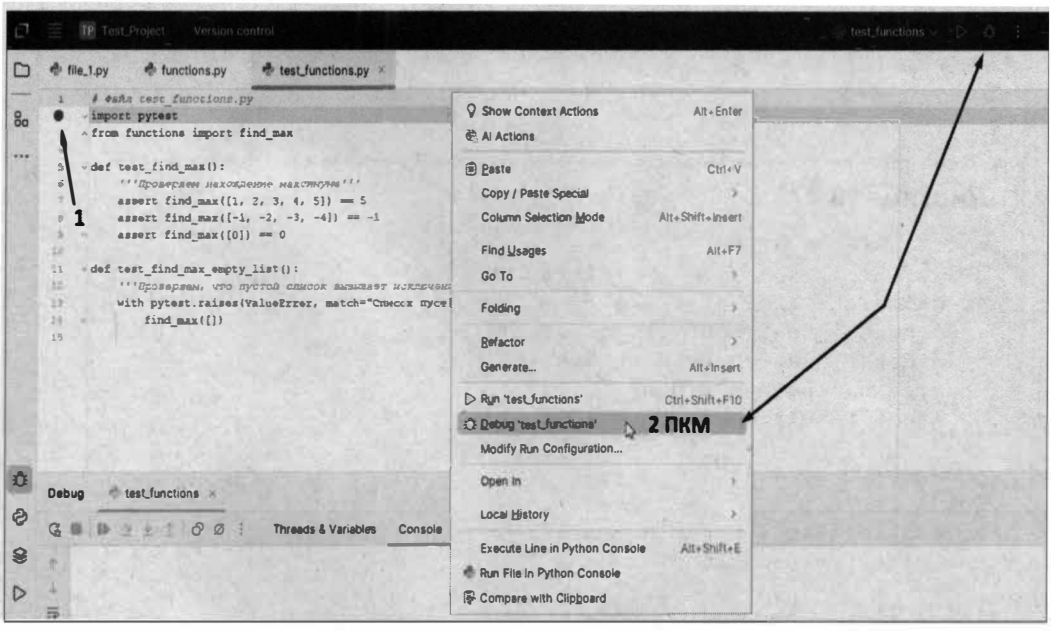
```
# Файл functions.py
def divide(a, b):
    """Возвращает результат деления a на b.
    Если b == 0, выбрасывает ValueError."""
    if b == 0:
        raise ValueError("Деление на ноль!")
    return a / b
```

Листинг 18.19

```
# Файл test_functions.py
import pytest
from functions import divide

def test_divide():
    '''Проверяем корректное деление'''
    assert divide(6, 3) == 2
    assert divide(10, 2) == 5
```

```
    assert divide(-10, 5) == -2

def test_divide_by_zero():
    '''Проверяем, что деление на ноль вызывает исключение'''
    with pytest.raises(ValueError, match="Деление на ноль!"):
        divide(5, 0)
```

4.



Изображение 18.8.

19.8.1. Ответы:

1.

Листинг 19.46

```
import tkinter as tk

def change_color(color):
    root.config(bg=color)

# Главное окно
root = tk.Tk()
```

```
root.title("Изменение цвета фона")
root.geometry("400x300")

# Кнопки для изменения цвета
colors = {"Красный": "red", "Зеленый": "green", "Синий": "blue"}

for name, color in colors.items():
    button = tk.Button(root, text=name, command=lambda c=color:
change_color(c))
    button.pack(pady=10)

root.mainloop()
```

2.

Листинг 19.47

```
import tkinter as tk

def calculate_area():
    length = float(entry_length.get())
    width = float(entry_width.get())
    area = length * width
    label_result.config(text=f"Площадь: {area}")

# Главное окно
root = tk.Tk()
root.title("Калькулятор площади")
root.geometry("300x200")

# Поле ввода длины
label_length = tk.Label(root, text="Длина:")
label_length.pack()
entry_length = tk.Entry(root)
entry_length.pack(pady=5)

# Поле ввода ширины
label_width = tk.Label(root, text="Ширина:")
label_width.pack()
entry_width = tk.Entry(root)
entry_width.pack(pady=5)

# Кнопка
button_calculate = tk.Button(root, text="Вычислить", command=calculate_
area)
button_calculate.pack(pady=5)

# Метка с результатом
```

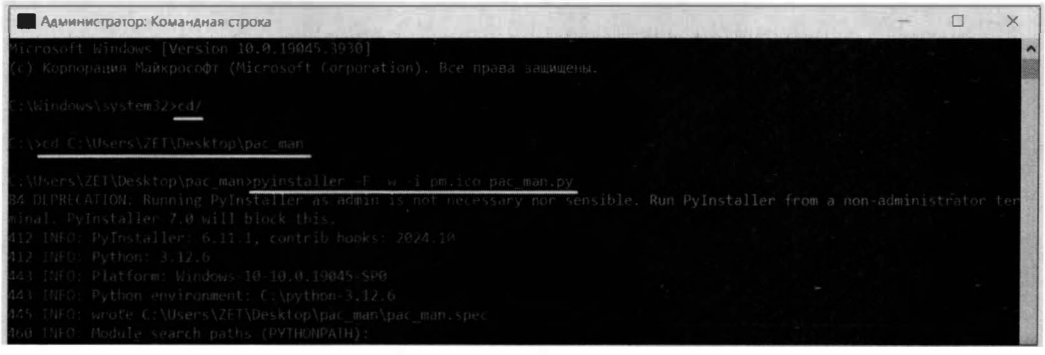
```
label_result = tk.Label(root, text="")
label_result.pack(pady=10)

root.mainloop()
```

3.

- Создаем папку с файлами:

Изображение 19.45.

- Открываем командную строку от имени администратора. Вводим команду *cd /*. Затем: *cd путь к папке с файлами*.
- Далее прописываем: *pyinstaller -F -w -i картинка.ico скрипт.py*.



Изображение 19.46.

20.5.1. Ответы:

1.

Листинг 20.16

```
import multiprocessing
import time

# Функция вычисляет квадрат числа
def square(x):
    time.sleep(1.0) # Имитация долгой работы
    return x * x

# Функция параллельно вычисляет сумму квадратов чисел
def parallel_sum_of_squares(numbers):
    # Создаем пул процессов, количество равно количеству ядер ЦП
    with multiprocessing.Pool(processes=multiprocessing.cpu_count())
as pool:
# Применяем функцию square к каждому элементу списка numbers
параллельно
        results = pool.map(square, numbers)
        return sum(results) # Суммируем результаты

if __name__ == "__main__":
    numbers = [5, 45, 13, 267] # Создаем список чисел
    start_time = time.time() # Отсечка старта параллельного вычисления
    result = parallel_sum_of_squares(numbers)
    end_time = time.time() # Отсечка завершения
    print(f"Сумма квадратов: {result}")
    print(f"Время выполнения: {end_time - start_time} секунд")

    # Для сравнения посчитаем последовательно
    start_time = time.time()
    result_seq = sum([square(x) for x in numbers])
    end_time = time.time()
    print(f"Сумма квадратов (последовательно): {result_seq}")
    print(f"Время выполнения (последовательно): {end_time - start_
time} секунд")
```

```
Сумма квадратов: 73508
Время выполнения: 1.324608564376831 секунд
Сумма квадратов (последовательно): 73508
Время выполнения (последовательно): 4.0042405128479 секунд

Process finished with exit code 0
```

Изображение 20.9.

- multiprocessing.Pool создает пул процессов, что позволяет эффективно использовать несколько ядер процессора.
- pool.map() применяет функцию square() к каждому элементу списка numbers параллельно.
- multiprocessing.cpu_count() возвращает количество ядер процессора, что позволяет автоматически определить оптимальное количество процессов.

2.

Листинг 20.17

```
import threading
import time

counter = 0 # Общий счетчик
lock = threading.Lock() # Создаем блокировку

def increment():
    global counter
    for i in range(500000):
        with lock:
            counter -= 1

if __name__ == "__main__":
    threads = []
    for i in range(1):
        t = threading.Thread(target=increment)
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

    print(f"Значение счетчика: {counter}") # Результат: -500000
```

- **with lock** — контекстный менеджер автоматически захватывает блокировку при входе в блок и освобождает ее при выходе, даже если произошло исключение. Это упрощает работу с блокировками и предотвращает забывание освободить блокировку.

22.6.1. Ответы:

1.

Листинг 22.25

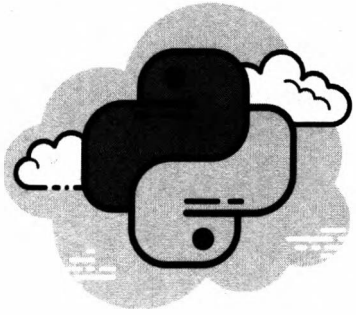
```
import sqlite3
# Подключаемся к базе данных (создаётся файл students.db)
conn = sqlite3.connect("students.db")
cursor = conn.cursor()

# Создаём таблицу students
cursor.execute("""
    CREATE TABLE IF NOT EXISTS students (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        age INTEGER NOT NULL )
""")
# Вставляем данные о студентах
students = [("Иван", 30), ("Марья", 25), ("Демид", 20)]
cursor.executemany("INSERT INTO students (name, age) VALUES
(?, ?)", students)
# Сохраняем изменения
conn.commit()

# Выводим все записи из таблицы
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()
for row in rows:
    print(row)
# Закрываем соединение
conn.close()
```

2.

Листинг 22.26

```
import sqlite3
# Подключаемся к базе данных
conn = sqlite3.connect("students.db")
cursor = conn.cursor()

# Запрашиваем имя у пользователя
search_name = input("Введите имя студента для поиска: ")
```

```
# Выполняем запрос
cursor.execute("SELECT * FROM students WHERE name = ?",
(search_name,))
student = cursor.fetchone()

# Выводим результат
if student:
    print(f"Найден студент: ID={student[0]}, Имя={student[1]},
Возраст={student[2]}")
else:
    print("Студент не найден.")

# Закрываем соединение
conn.close()
```



Заключение

Конечно, в одной книге нельзя охватить весь язык Python.

В этой книге вы познакомились с основными темами и получили знания, которых будет более чем достаточно для устройства на работу программистом на позицию Junior Python-разработчик (со средней заработной платой 1000–1500 долларов США).

Кроме того, вы можете работать на себя — искать заказы на фриланс биржах или создавать свои собственные приложения и игры, а затем выпускать их на таких площадках, как Google Play, Steam и подобных.

В зависимости от того, чем бы вы хотели заниматься дальше, вы можете изучать одно из следующих направлений:

1. Веб-разработка.

Backend — создание серверной части веб-приложений. Популярные фреймворки: Django, Flask.

Frontend — разработка клиентской части веб-приложений. Пайтон можно использовать с помощью таких фреймворков, как Brython, но чаще он применяется для создания бэкенда, а интерфейс разрабатывается на JavaScript.

2. Data Science и машинное обучение.

Анализ данных — обработка, очистка и анализ больших объемов данных. Библиотеки: Pandas, NumPy.

Визуализация данных — создание интерактивных графиков и диаграмм. Библиотеки: Matplotlib, Seaborn.

Моделирование — создание математических моделей и проведение симуляций.

Машинное обучение — разработка моделей для прогнозирования, классификации и других задач. Библиотеки: Scikit-learn, TensorFlow, PyTorch.

3. Автоматизация.

Системная администрация — автоматизация рутинных задач, управление конфигурацией.

Тестирование — автоматизация тестирования программного обеспечения. Библиотеки: pytest, unittest.

Скрапинг данных — извлечение данных с сайтов. Библиотеки: BeautifulSoup, Scrapy.

4. Разработка программ и приложений.

Разработка десктопных приложений — использование библиотек Tkinter, PyQt, wxPython и др.

Создание утилит командной строки — автоматизация задач операционной системы.

Разработка 2D- и 3D-игр — с использованием библиотек Pygame, Kivy и таких движков, как Renpy и Godot.

Чтобы закрепить материал, рекомендуется перечитать книгу еще раз. Снова прорешайте все задачи. При повторном прочтении многие вещи станут намного понятнее и закрепятся в долговременной памяти.

Также стоит иметь в виду, что навыки, которыми вы не пользуетесь на постоянной основе, постепенно «выветриваются». Поэтому продолжайте изучение языка, практикуйте программирование, создавайте небольшие программы для своих нужд.

Ну, а на этом всё, надеюсь вам понравилась книга (или, я надеюсь, что прочитанный материал был вам как минимум полезен).

С вами был Анатолий Адонин, мой профиль в сети вы можете найти на этой странице: *vk.com/anzheri*.

Успехов в работе и творчестве!

Издательство «Наука и Техника» выпускает книги более 25 лет!

Уважаемые авторы!

Приглашаем к сотрудничеству по созданию книг
по **IT-технологиям, электронике, электротехнике, медицине, педагогике.**

Наши преимущества:

- являемся одним из ведущих технических издательств страны;
- выпускаем книги большими тиражами, что положительно влияет на гонорар авторов;
- регулярно переиздаем тиражи, автоматически выплачивая гонорар за *каждый* тираж;
- применяем индивидуальный подход в работе с каждым автором;
- работаем профессионально: от корректуры до авторских дизайн-проектов;
- проводим политику доступной цены;
- имеем собственные каналы сбыта: от федеральных сетей, крупнейших книжных магазинов РФ, ведущих маркетплейсов ОЗОН, Wildberries, Яндекс-Маркет и др. до ведущих библиотек вузов, ссузов.

Ждем Ваши предложения:

- **тел. (812) 412-70-26**
- **эл. почта: nitmail@nit.com.ru**

Будем рады сотрудничеству!

Для заказа книг:

- **интернет-магазин: nit.com.ru**
  - 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 18.00
  - справки и заказ: тел. (812) 412-70-26

- **книжные сети и магазины**
  - «Читай-город» - сеть магазинов тел. +7 (495) 424-84-44
  - «Буквоед» - сеть магазинов тел. +7 (812) 601-0-601
  - Московский дом книги – сеть магазинов тел. +7 (495) 789-35-91
  - ТД «БиблиоГлобус» тел. +7 (495) 781-19-12
  - «Амиталь» — сеть магазинов тел. +7 (473) 223-00-02
  - Дом книги, г. Екатеринбург тел. +7 (343) 289-40-45
  - Дом книги, г. Нижний Новгород тел. +7 (831) 246-22-92
  - Приморский торговый Дом книги тел. +7 (423) 263-10-54

- **маркетплейсы ОЗОН, Wildberries, Яндекс-Маркет, Myshop и др.**

Адонин А. М.

Профессия:
Python-разработчик

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *Е.В. Финков*

Редактор: *О.С. Петрунич, Н. В. Жерлов*

Корректор: *А.В. Громова*

Изображения на обложке и в книге использованы с ресурсов freepik.com и vecteezy.com

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 24.03.2025. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 32 п.л.

Тираж 2000. Заказ 12886.

Отпечатано с готового оригинал-макета

ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1

ПРОФЕССИЯ
Python-разработчик

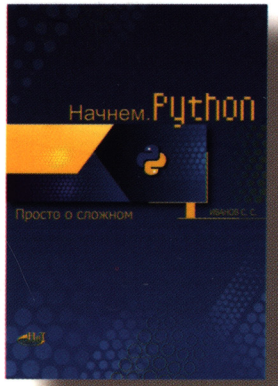
А. Адонин

Книга позволит вам получить знания, достаточные для профессии Python-разработчик (начиная с уровня Junior Developer). Это даст вам возможность устройства на работу программистом, или возможность работать на себя – искать заказы на фриланс-биржах, создавать свои собственные приложения и игры, а затем выпускать их на таких площадках как Google Play, Steam и подобных.

Шаг за шагом вы познакомитесь с основными принципами Python, научитесь писать первые программы, работать с данными, создавать графические интерфейсы, разрабатывать свои приложения, писать рабочие скрипты по созданию страниц регистрации, банковских приложений и других направлений разработки. Вы также изучите принципы тестирования и отладки кода, что сделает ваши программы более надёжными. В заключительной части книги вы освоите продвинутые темы: многопоточность, асинхронное программирование, работу с модулями и пакетами, а также основы взаимодействия с реляционными, SQL и NoSQL базами данных.

В книге большое внимание уделяется практическим навыкам – каждая глава содержит примеры с разбором кода и различные задачи, которые помогут закрепить материал на практике.

издательство НАУКА и ТЕХНИКА рекомендует



Наши книги — ваши инвестиции в будущее

ISBN 978-5-907592-73-5

«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