Интерактивная цифровая книга

Анатолий Постолит

Самоучитель Python с интерактивными примерами



Постолит А.В.

Самоучитель Python с интерактивными примерами. – М.: 2025. -470 с.: ил.

Интерактивный самоучитель по основам языка программирования Python предназначен для начинающих программистов, которые решили ознакомиться с основами Python с нуля. Эта книга написана простым языком, сопровождается большим количеством примеров и заданий для самоконтроля. Кроме того, для лучшего закрепления теории к пособию прилагаются еще и тесты для контроля знаний. Все это позволит новичкам самостоятельно изучить основы Python с нуля за максимально короткие сроки.

Интерактивный раздел книги включает в себя текстовые материалы, рисунки, листинги программных модулей, интерпретатор языка Python, редактор программного кода и набор дополнительных библиотек для запуска программных модулей. Пользователю нет необходимости устанавливать на свой компьютер, какие либо инструментальные средства. Изучать листинги программ, вносить в них изменения и запускать на выполнение можно в рамках самой книги в интернет браузере.

Книга предназначена как для начинающих программистов, так и для читателей, имеющих опыт работы с языками программирования, отличными от Python. Кроме того, данная книга может быть использована как лабораторный практикум для школьников старших классов, студентов ВУЗов и слушателей компьютерных курсов при изучении практических приемов программирования приложений на Python. С использованием данного пособия можно не только изучать предложенные материалы, но и осуществлять, как самостоятельное тестирование полученных знаний, так и прием зачетов и экзаменов.

Предисловие

Можно поздравить всех поклонников языка программирования Python, уже не один год данный язык является самым популярным языком программирования в мире! Для оценки популярности языков программирования в международном сообществе используется показатель - «Индекс TIOBE Programming Community», который обновляется раз в месяц. Рейтинги основаны на количестве квалифицированных инженеров по всему миру, курсах и сторонних поставщиках. Для расчета рейтингов используются популярные поисковые системы, такие как Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube и Baidu. Рейтинг языков программирования на ноябрь 2025 года представлен на рис. 1.

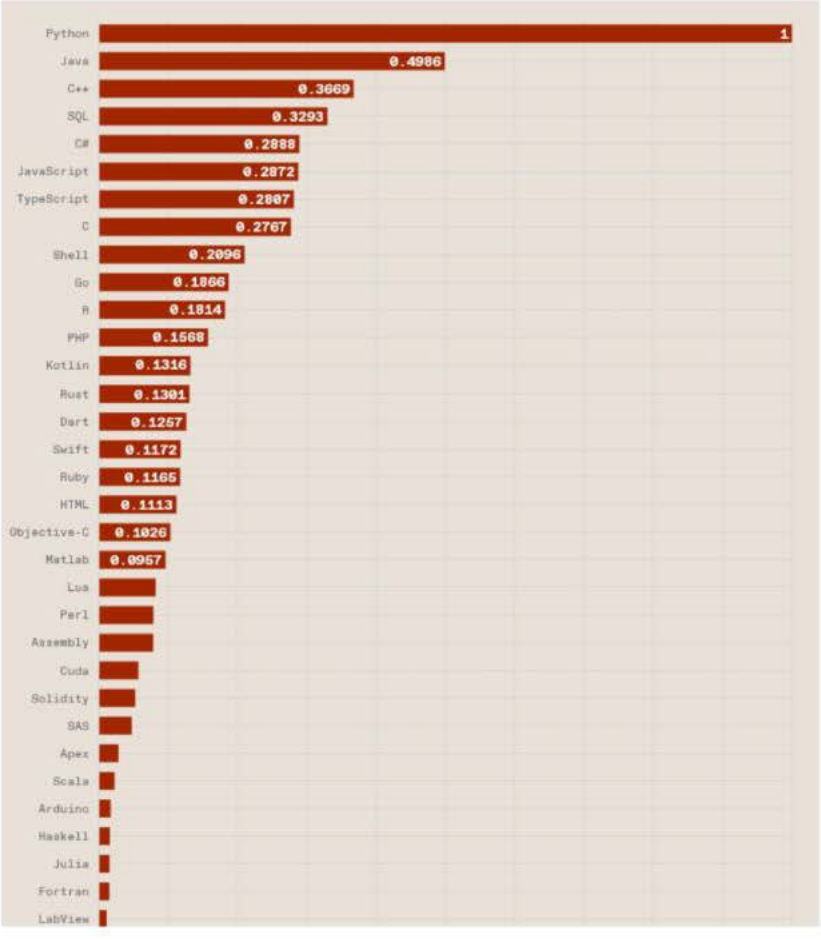


Рис.1. Рейтинг популярности языков программирования по состоянию на ноябрь 2025 г.

Python это один из лучших языков для начинающих. Он гибкий, почти «всепрощающий», и у него простой и понятный синтаксис. В работе на Python не нужно беспокоиться о тонкостях работы с памятью как, например, в C++, и можно сразу приступить к

программированию в дружелюбной среде. А ещё Python подробно пишет, где и какая ошибка в программном коде (правда на английском), но все равно это очень удобно.

Данная интерактивная электронная книга предназначена как для начинающих программистов (школьники и студенты), так и для специалистов с опытом, которые планируют заниматься или уже занимаются разработкой приложений с использованием Python. Для того чтобы начать работать с книгой, нужен только компьютер, при этом нет необходимости на компьютер устанавливать какие либо дополнительные программы. Книга работает в режиме «все включено». В нее, кроме текста и рисунков, встроен интерпретатор Python, листинги программных модулей, упрощенный редактор программного кода. Работу с интерактивной книгой в веб браузере обеспечивают фреймворки Django и Streamlit. **Можно изучать код, запускать его на выполнение, вносить в него изменения, и все это не выходя из самой книги и вашего интернет браузера.** Мало того, код программ можно экспортировать и эти фрагменты использовать в своих разработках в качестве мини-шаблонов. В книге очень удобное оглавление: можно моментально попасть в любой раздел любой главы. Не обязательно изучать материалы книги строго последовательно, можно быстро переключаться между разными разделами.

Кроме того, практически в каждом разделе книги имеются контрольные вопросы. Эти вопросы включены в систему тестирования знаний, которая также встроена в интерактивную электронную книгу, а тестирование можно пройти на сайте **rumnik.ru**. Таким образом, она может использовать в учебных заведениях и как учебное пособие, и как средство контроля знаний. Еще в книгу встроены средства перевода ее содержания на различные языки, и даже система озвучивания текста на этих языках. Эти сервисы работают на основе элементов искусственного интеллекта.

Не смотря на то, что книга предназначена для начинающих программистов, она содержит довольно много материала, касающегося структуры языка программирования Python, его внутренних механизмов и тонкостей. С первого прочтения этот материал может показаться не очень понятным (особенно для начинающих), и чтобы разобраться с ним, нужно детально изучать приведенные в книге примеры. Если такой материал достаточно сложен для понимания, а возможности таких тонкостей не особо важны на текущий момент, данный материал можно просто пропустить. По мере приобретения новых сведений из других разделов, к этому материалу всегда можно вернуться.

Это первая интерактивная книга из серии «Академия Python». Далее будут выпущены аналогичные интерактивные книги по программированию на Python пользовательского интерфейса, разработки web приложений, кроссплатформенных мобильных и настольных приложений, систем на основе компьютерного зрения, искусственного интеллекта, распознавания и синтезирования голоса, создания ботов и т.п.

Структура книги

В *первой главе* книги приводятся интересные факты об истории возникновения данного языка программирования, о базовых структурах и особенностях синтаксиса.

Вторая глава посвящена типам данных, которые можно обрабатывать в Pytnon. Это числа, строки, списки, кортежи, словари и множества. Рассказывается о том, как эти данные хранятся в компьютере, и как можно одни типы данных преобразовывать в другие. Это, пожалуй, самая длинная глава, но материалы этой главы нужно изучить основательно, так как она является базой для создания программных модулей

Третья глава посвящена инструкциям, выражениям и операторам. Это основные элементы, которые формируют строки программного кода. Так же рассматриваются инструкции для формирования блоков программного кода, которые состоят из этих строк. К таким инструкциям относятся ключевые слова, которые организуют разветвления в программах, организацию циклов, итерацию и генерацию данных.

Четвертая глава посвящена функциям. Приводятся общие сведения о функциях, о параметрах и аргументах, об областях видимости данных. Рассматриваются так называемые неименованные и рекурсивные функции, аннотированные функции и особые функции декораторы

В пятой главе рассмотрены вопросы объектно-ориентированного программирования. Приводятся сведения о классах и объектах, примеры их создания и использования.

Шестая глава посвящена модулям, пакетами и библиотекам. Это основные «кирпичики», из которых строятся приложения. Разбираются особенности импорта программных модулей, импорта и использования богатейших библиотек Python, которые можно найти на просторах интернета, и использовать в своих приложениях.

В седьмой главе рассмотрены инструментальные средства, которые используются для разработки приложений на Python. Это сам интерпретатор Python и инструментарий для написания программного кода PyCharm и GigaIDE. Эти программные средства не нужны в процессе работы с книгой, но далее, когда программист начнет работать самостоятельно, их нужно обязательно устанавливать на свой компьютер. В материалах данной главы как раз и рассказывается о том, как выполнить установку этого инструментария на компьютеры, работающие под операционными системами Windows и Linux.

В материалах книги некоторые фрагменты текста выделены особым образом. Это сделано для того, чтобы по оформлению текста сразу было видно, какая особенность у выделенного фрагмента.

Все листинги программ в книге имеют черный цвет фона. Примеры программ разрабатывались с использованием IDE PyCharm. В текстовом редакторе этого инструментария имеется темная и светлая «тема» для задания фона экрана. У программистов большей популярностью пользуется именно темная «тема», которая имеет название «Dracula». Это связано с тем, что темный цвет экрана меньше раздражает глаза при длительной работе за монитором. Ниже приведен пример текста с листингом программы:

```
def my_fun_1(x, y):
    print('Тип переменной y ->', type(y))
    rez = x + y
    return rez
```

Листинги программ приводятся в оригинальном виде еще и для того, чтобы сохранить цвета, которые используются для написания ключевых слов и функций. В текстовых редакторах специфичные фрагменты программного кода выделяются разными цветами.

Результаты работы программных модулей представлены в виде изображений, выведенных в интернет браузер с использованием интерактивного раздела книги. Так как некоторые подобные изображения являются достаточно большими, то результаты работы таких программных модулей представлены текстом с выделением серым фоном, например:

```
a = 'Стоимость товара - '
b = 100
```

```
c = ' руб.'
d = a + b + c
print(d)
```

Если текст содержит примечание или иную важную информацию, то он выделяется другим шрифтом, например:

> ***Примечание.***
> Здесь присутствует ключевое слово «def», которое объявляет пользовательскую функцию. О пользовательских функциях будет подробно рассказано в следующих главах.

На протяжении всей книги раскрываемые вопросы сопровождаются достаточно упрощенными, но полностью законченными примерами. Это в значительной степени облегчает восприятие приведенных в книге материалов. Наилучший способ обучения – это практика. Интерактивная электронная книга - это прекрасная возможность познакомится с языком программирования Python, и понять, насколько он прост и удобен в использовании. Итак, если Вас заинтересовало изучение основ программирования на Python, то самое время перейти к изучению материалов этой интерактивной электронной книги.

О примерах программного кода и интерактивности

На протяжении всей книги раскрываемые вопросы сопровождаются по возможности упрощенными, но полностью законченными примерами. Ход решения той или иной задачи показан на большом количестве иллюстративного материала. Некоторые фрагменты программного кода могут показаться сложными, но такие примеры содержат поясняющие комментарии. В программном коде интерактивного раздела книги гарантированно нет ошибок. Абсолютно все примеры интерактивного раздела можно запустить на выполнение в рамках самой книги, так как она содержит весь необходимый инструментарий. Когда вы увидите результаты работы программного кода, появится ясность — как работает тот или иной элемент изучаемого примера. Мало того, зарегистрированные пользователи платформы могут скачать листинги ее программных кодов и использовать их в своих проектах.

> ***Интерактивная часть цифровой книги***
> Работать с интерактивной частью книги можно на сайте **rumnik.ru**. Демо-версию книги можно найти в главном меню в опции **Каталог**. Полная версия книги будет доступна в разделе **Мои книги** личного кабинета зарегистрированного пользователя после ее приобретения в магазине сайта (цифровая PDF версия книги и интерактивный раздел цифровой книги являются самостоятельными продуктами и продаются раздельно).
>
> Использование полной версии интерактивного раздела книги избавляет вас от необходимости приобретения «бумажной» версии книги — если вам это покажется более удобным…

Наилучший способ обучения — это практика. Все листинги программ приведены на языке Python. Любой пример, приведенный в интерактивном разделе книги, можно запустить на выполнение в интернет браузере и посмотреть, как выглядит конечный результат.

Итак, если вас заинтересовали вопросы создания эффективных приложений с использованием Python, то самое время перейти к изучению материалов этой книги и ее цифровой интерактивной версии.

Глава 1. Введение в Python



Программирование (от англ. programming) – это процесс создания компьютерных программ. Для того чтобы иметь возможность общаться с компьютером и создавать программы, нужно знать один или несколько языков программирования. Благодаря языкам программирования человек получает возможность писать программы и ставить перед компьютером конкретные задачи, которые тот впоследствии решает в соответствии с заданными условиями.

В настоящее время число языков программирования перевалило за сотню. Однако пугаться этого не стоит, поскольку, несмотря на то, что языки могут сильно отличаться областью применения, все они имеют одну и ту же логическую основу и очень похожий синтаксис. Поэтому, хорошо изучив какой-нибудь один современный язык программирования, вы в дальнейшем сможете без особых проблем освоить и любой другой необходимый вам язык.

Вместе с тем следует помнить, что изучение любого предмета подразумевает не только желание учиться, но и некоторую предрасположенность к изучаемой области. Что касается программирования, то его изучение будет даваться значительно проще людям, у которых есть склонность к математике и другим точным наукам, поскольку в программировании очень важно уметь хорошо мыслить и логически думать. Кроме того, приветствуется знание английского языка, а также терпеливость, упорство и усидчивость.

У всех людей данные качества и возможности развиты по-разному. Поэтому однозначно ответить на вопрос, сложно ли стать хорошим программистом, довольно затруднительно. Кто-то сможет освоить программирование самостоятельно, не выходя из дома, а кто-то без посторонней помощи вообще не сможет начать процесс обучения даже при наличии учебников и наставников. В любом случае нужно помнить, что хорошими программистами не рождаются - ими становятся. При этом не нужно думать, что стать лучшим программистом - это значит с отличием окончить соответствующий ВУЗ. Многие представители IT-сферы либо совсем не учились в университете, либо покинули его еще до получения диплома. Вот лишь несколько примеров: Билл Гейтс – основатель компании Microsoft, Марк Цукерберг – создатель социальной сети Facebook, Стив Джобс – основатель компании Apple, Есть довольно много примеров, когда целеустремленные самоучки во многом превосходят дипломированных специалистов.

Языков программирования много и не все они достаточно легки для изучения. В интернете можно найти немало различных рейтингов сложности языков и их популярности. В любом случае желательно начинать с достаточно распространенного и востребованного языка программирования, который по многочисленным отзывам не сложен в изучении, а также имеет много русскоязычных книг и учебных пособий с примерами кода и программ. Одним из таких языков является Python.

Следует отметить, что простого чтения книг и учебников не достаточно. Большую часть времени нужно посвящать именно набору кода и практической реализации различных задач. Нужно писать как можно больше программного кода, начиная с самых простых задач, посвящая этому 70-80% учебного времени.

Из материалов данной главы вы узнаете:

- кто же создал этот замечательный язык программирования, который за относительно небольшой срок стал самым популярным и востребованным языком в мире;
- какого типа программы могут быть писаны на Python;
- правила базового синтаксиса при написании программного кода на Python;
- где можно найти дистрибутив Python, чтобы установить его на свой компьютер.

1.1.Кем и когда был создан Python

Датой рождения языка программирования Python можно считать февраль 1991 года, когда его создатель Гвидо ван Россум опубликовал в группе новостей исходный текст Python версии 0.9.0. Голландский инженер и программист Гвидо ван Россум может служить примером для современной молодежи, поскольку он обладает такими качествами, как упорство, усидчивость и целеустремленность.

Ван Россум создал работающий прототип Python всего за три месяца. «В те дни я редко выходил из дома, так как в основном смотрел телевизор и писал код», — рассказывает Гвидо. А по телевизору в то время шло популярное юмористическое шоу «Monty Python Flying Circus» (Летающий цирк Monty Python). Название языка не имеет ничего общего с известным всем змеем питон, поскольку язык назван так в честь популярной британской комедийной труппы Монти Пайтона. Не смотря на это, на обложках большинства книг по программированию на Python присутствует изображение именно змеи, в результате чего у большинства людей название языка ассоциируется с этой рептилией.

Гвидо ван Россум учился в Амстердамском университете, который окончил со степенью магистра математики и компьютерных наук. Гвидо учился на математическом факультете. Но среди прочих предметов было и программирование. Юноша быстро понял, что работа с компьютерами была для него не просто частью учебной программы, а по-настоящему приятным времяпрепровождением. Со временем любознательный студент забросил математику с электроникой и полностью погрузился в премудрости программирования, так как эта сфера показалось ему более интересной. Ван Россум во время учебы устроился на подработку в дата-центр университета. Работа настолько доставляла ему удовольствие, что он просто забывал про учебу. Так подающий надежды студент оказался на грани отчисления. К счастью начальник центра, и по совместительству профессор университета, вовремя уберег юношу от опрометчивых поступков. Он убедил Гвидо продолжить обучение, объяснив, что без диплома он не сможет остаться на любимой работе.

В декабре 1989-го года в качестве хобби он начал дорабатывать учебный язык программирования «ABC», и даже не подозревал, что из этого может получиться что-то серьезное. Язык программирования, создаваемый им на основе упрощенного английского языка, отличался от других языков простотой и открытым кодом, что обеспечивало любому пользователю возможность вносить в язык свои коррективы. Несмотря на то, что команды отличались от популярных на тот момент языков, программисты легко осваивали язык и уже через несколько дней могли свободно использовать его в своей работе. Коллеги ван Россума сразу же стали использовать первую версию Python.

Гвидо ван Россум представил Python в новостных группах alt.sources в 1991 году. Это фактически был opensource - проект, хотя термин «open source» впервые использовали только через 6 лет. В то время пользователям приходилось целую ночь загружать файл с интерпретатором Python через сеть. Но это было эффективнее походов от двери к двери с

кассетами, на которых записывались программы. Ван Россум был одним из первых разработчиков и программистов, выступавших за развитие свободного программного обеспечения и популяризацию открытого кода.

С 1995 года Гвидо Ван Россум продолжил работу над Python-ом в корпорации национальных исследовательских инициатив (CNRI) в городе Рестон, штат Вирджиния, где было выпущено несколько версий языка. С тех пор Python стал очень популярен среди разработчиков, которых привлекает простой синтаксис языка и удобство читаемости кода.

Вместе с новыми друзьями Гвидо ван Россум открыл сайт python.org. На этом ресурсе хранилось ядро Python, и по сегодняшний день на этом сайте можно найти все актуальные версии языка. Также сайт помогал координировать работу специалистов, которые участвовали в развитии Python. Уже во второй половине 90-х вокруг Python сформировалось многочисленное сообщество. Чтобы формализовать управление языком, Гвидо ван Россум и его друзья создали сообщество Python Software Foundation (это некоммерческая корпорация, которая владеет правами интеллектуальной собственности на язык программирования Python). Окончательно эта организация оформилась в 2001 году, и далее Python развивался преимущественно благодаря именно этому сообществу. При этом Гвидо продолжал играть ключевую роль в развитии языка.

В следующие годы Python быстро развивался. Тектонические изменения произошли в 2008 году, когда был представлен Python версии 3.0. В 2018 году Гвидо ван Россум оставил пост неформального руководителя разработкой языка, но остался в наблюдательном совете, который следит за его развитием.

Популярность Python растет, и по состоянию на сентябрь 2022 года он стал самым востребованным языком программирования в мире, а по данным рейтинга языков программирования TIOBE за апрель 2025 года, Python занимает первое место. Но опытные представители сообщества, развивающие Python, уже говорят о вызовах, с которыми столкнется язык в ближайшее время. Python должен преодолеть эти вызовы, чтобы оставаться популярным. На одной из конференций сообщества разработчиков было высказано мнение, что у Python могут возникнуть серьезные проблемы, если в ближайшее время не улучшится поддержка мобильных приложений и современных веб-платформ. Смартфонами и планшетами пользуется практически каждый человек, поэтому сообществу разработчиков Python надо задуматься об использовании языка на девайсах этого типа. В этом направлении уже сделаны определенные шаги – появился фреймворк Kivy и библиотека KivyMD, которые обеспечивают создание кроссплатформенных приложений на Python, работающих на устройствах любого типа и под любыми операционными системами. Гвидо ван Россум тоже уверен, что сообщество будет главным двигателем дальнейшего развития Python. «У нас очень сильное сообщество и новая система управления, поэтому мы готовы к вызовам, с которыми столкнется язык», — говорит ван Россум.

1.2.Сильные стороны языка программирования Python

Язык программирования Python (Питон или Пайтон) это высокоуровневый, интерпретируемый, объектно-ориентированный язык общего назначения с динамической типизацией и автоматическим управлением памятью, который распространяется бесплатно. Для высокоуровневых языков программирования используются специальные программы, которые переводят код программ на исходном языке в машинный код, то есть понятный для

компьютеров двоичный язык «нулей и единиц». Существует два типа таких программ: интерпретаторы и компиляторы.

Интерпретатор – это специальная программа, которая выполняет интерпретацию, то есть последовательно анализирует поступающий на ее вход исходный код программы и тут же оператор за оператором, строчка за строчкой выполняет ее, предварительно преобразуя каждый кусок исходного кода в машинный код. Если в ходе интерпретации будет обнаружена ошибка, интерпретатор прекратит дальнейшее выполнение программы. Так вот, Python является интерпретируемым языком программирования. Программы на нем могут выполняться на любом устройстве, в котором встроен интерпретатор языка. А поскольку Python имеет официальный сайт и распространяется бесплатно, то скачать и установить интерпретатор на свой компьютер не составит особого труда даже для обычных школьников. Помимо Python интерпретируемыми являются, например, языки JavaScript и PHP.

Компилятор – это специальная программа, которая выполняет компиляцию, то есть еще до начала использования готовой программы полностью преобразует исходный код какого-либо языка программирования в понятный для компьютера машинный код. Компиляция программы выполняется один раз, и если процесс компиляции завершается без ошибок, программу можно будет запустить на компьютере без повторного компилирования даже при отсутствии на нем компилятора.

Как видим, главной задачей, как интерпретатора, так и компилятора является преобразование кода языка высокого уровня в машинный код, хотя и делают они это по-разному.

Есть множество причин для успеха Python. Python имеет гораздо более простой и понятный синтаксис, чем другие популярные языки, например, Java, C и C++, что упрощает изучение. Многие считают, что один из главных факторов успеха Python это наличие дружелюбного и отзывчивого сообщества пользователей, наряду с дизайнерской интуицией его создателя Гвидо Ван Россума.

Программы, написанные на Python, уступают по скорости программам, написанным на C или C++, однако на современных аппаратных средствах эта разница в скорости практически не заметна. При этом программисты экономят время на написание программного кода. Имеется возможность пробовать короткие фрагменты программ в интерактивной среде без предварительного компилирования.

Python сочетает в себе многие из лучших принципов и идей из разных языков программирования. В отличие, например, от девиза Perl «есть несколько способов сделать это, и пользуйся любым», Python придерживается философии «должен существовать один и, желательно, наилучший способ сделать это».

Python часто изучается и рекомендуется как первый язык программирования. Простота синтаксиса Python обеспечивает высокую производительность при написании кода. Использование интерпретатора упрощает тестирование и отладку программ. Python также помогает понять основные концепции объектно-ориентированного программирования.

Python - это интерпретируемый язык программирования высокого уровня с динамической типизацией. Язык Python поддерживает различные подходы и виды программирования:

- императивное (исходный код, основан на командах, которые выполняются последовательно);
- процедурное (деление программ на подпрограммы);

- структурное (представление программы в виде блоков);
- объектно-ориентированное (основанное на классах и объектах);
- мета программирование (написание программ, которые управляют другими программами);
- функциональное (процессы оформляются в виде функций).

Python - очень компактный язык программирования, краткий синтаксис прост, но при этом очень продуктивен. Программы на Python намного короче кода, написанного на других современных языках программирования. Меньше кода - меньше ошибок.

Что касается динамической типизации, используемой в языке Python, то она избавляет программиста от необходимости объявлять тип переменной при ее создании, так как типы данных определяются интерпретатором автоматически во время выполнения программы. Поэтому переменной можно сначала присвоить число (один тип данных), а затем, в случае необходимости, строку или какой-нибудь другой объект или другой тип данных. При этом Python самостоятельно распределяет память под объекты программы и автоматически освобождает ее, когда объекты становятся ненужными.

Также следует добавить, что с самого начала Python проектировался как объектно-ориентированный язык программирования (ОПП) с поддержкой таких понятий, как полиморфизм и наследование. Однако использование ООП никогда не являлось обязательным, поскольку Python поддерживает сразу несколько стилей программирования.

Необычной особенностью языка является выделение блоков кода пробельными отступами. Гвидо ван Россум считает, что использование отступов для группировки инструкций делает программный код чрезвычайно элегантным, такой код не перегружен лишними знаками и легко читается.

Стандартная библиотека, распространяемая вместе с Python, очень обширна и предлагает широкий спектр возможностей. Библиотека содержит встроенные модули (написанные на C), которые обеспечивают доступ к функциям операционной системы, таким как файловый ввод-вывод. В ней также содержатся модули, написанные на Python, в которых предоставлены многие решения для часто повторяющихся повседневных задач, для создания графического оконного интерфейса взаимодействия с пользователями.

Python пользуется популярностью среди разработчиков еще и потому, что в дополнение к стандартной библиотеке можно использовать множество дополнительных библиотек, пакетов и фреймворков для разработки различных типов приложений. К ним можно отнести следующие библиотеки:

NumPy - фундаментальный пакет для научных вычислений, для быстрых операций с массивами и матрицами, основные статистические операции, моделирование и многое другое.

TensorFlow - библиотека с открытым исходным кодом для численных вычислений, а так же ускорения и упрощения машинного обучения. Для новичков и экспертов TensorFlow позволяет легко создавать модели машинного обучения для настольных компьютеров, мобильных устройств, Интернета и облака.

Pandas - библиотека машинного обучения на Python и обработки данных. Одна из замечательных особенностей этой библиотеки - возможность производить сложные операции с данными с помощью одной или двух команд.

Matplotlib - комплексная библиотека для создания статических визуализаций в Python. Визуальные эффекты полностью настраиваются и могут быть легко экспортированы в другие приложения.

Plotly - библиотека для создания анимированных и интерактивных визуализаций в Python.

PyQt - это набор виджетов графического интерфейса для доступа к базам данных с помощью SQL (ODBC, MySQL, PostgreSQL, Oracle), в этот набор также входит Qt Designer для создания графического интерфейса взаимодействия с пользователями.

Pygame - «игровая библиотека», набор инструментов, помогающих программистам создавать видеоигры и мультимедийные программы на языке Python.

Pillow – библиотека предназначена для работы с растровой графикой и имеет довольно мощные возможности обработки изображений.

Django - фреймворк для разработки web-приложений и сайтов, работающих с базами данных.

Streamlit - фреймворк для разработки web-приложений, связанных с обработкой данных, машинным обучением и системами искусственного интеллекта..

Kivy + KivyMD – фреймворк и библиотека, позволяющие создавать мобильные приложения на Python под Adroid и iOs, кроссплатформенные настольные приложения под Windows, Linux и MacOS.

Scrapy - фреймворк для сканирования веб-сайтов и извлечения из них необходимых данных.

Python, который начинался как простой язык сценариев, стал зрелым и сделал большой шаг вперед. Простота обучения, огромное количество библиотек и широкое использование во всех областях сделали его самым популярным языком программирования в мире. С использованием данного языка многие компании стали мировыми лидерами. Так в 1998 году небольшая компания из Калифорнии с помощью языка программирования Python реализовала способ поиска актуальной информации в Интернете. Эта малоизвестная компания называлась Google. Двенадцать лет спустя это имя оценивалось в 111 миллиардов долларов США, что сделало его одним из самых ценных брендов в мире. Кроме Google с использованием Python созданы такие известные сайты как YouTube, Instagram, Yahoo Maps, Mozilla и многие другие.

Безусловно, технический прогресс набирает обороты, и языки программирования стремительно развиваются. Появляются новые развивающиеся языки, которые потенциально могут составить серьезную конкуренцию Python. Но в настоящее время Python пользуется популярностью благодаря тому, что:

- Python - отличный стартовый язык для начинающих, не владеющих языками программирования;
- Python - лучший способ начать работу любому, особенно тому, кто планирует пойти дальше в области науки о данных, искусственного интеллекта и машинного обучения;
- разработчикам на Python не нужно тратить слишком много времени на понимание синтаксиса дружелюбного языка программирования;
- Python распространяется под лицензией с открытым исходным кодом. (это делает его бесплатным для использования и распространения);
- хорошо переносим, и надежно работает практически на всех основных платформах (т.е. обладает кроссплатформенностью);

- легко соединяется с компонентами, которые написаны на других языках программирования, например, C++, что дает возможность использовать программный код Python в чужеродных языковых средах и системах.

А ещё у Python есть огромное потрясающее сообщество, наполненное замечательными людьми, которые постоянно обновляют и совершенствуют этот язык программирования. Благодаря такому огромному набору преимуществ Python по праву стал самым популярным и любимым языком программирования в мире, заслуженно занимая первую строчку мировых рейтингов.

1.3.Сферы применения языка программирования Python

Как было сказано в предыдущем разделе, Python является языком программирования общего назначения, что предполагает наличие у него возможностей для решения широкого круга самых разнообразных задач. Перечислим наиболее популярные из них.

Python довольно успешно используется в веб-разработке. Для расширения возможностей языка в этой области используются фреймворки Django, Pyramid, Flask, Streamlit.

В машинном обучении (Machine Learning) и в приложениях обработки данных (Data Science) на Python пишут системы искусственного интеллекта и анализа данных. В этом помогают библиотеки Keras (открытая нейросетевая библиотека для глубокого обучения), OpenCV (компьютерное зрение) или Scikit-Learn (дата-майнинг и анализ данных).

Это язык популярен и в научных исследованиях. Здесь не обойтись без библиотек NumPy (поддержка математических функций для работы с матрицами и многомерными массивами), Matplotlib и Plotly (визуализация 2D и 3D графики), SciPy (целый арсенал для выполнения инженерных и научных расчетов).

На серверных площадках и дата центрах Python широко применяется для автоматизации задач системного администрирования (он по умолчанию установлен на все серверы с ОС Linux).

В компьютерных играх на Python пишут скрипты для взаимодействия персонажей, запуска сцен, обработки событий, реализации пользовательского интерфейса. Примерами могут служить многопользовательские игры вроде World of Tanks, EVE Online или Battlefield 2.

Также на Python разрабатывают практически любые настольные, а также мобильные кроссплатформенные приложения.

Как видим, Python действительно очень востребован в сфере разработки программного обеспечения. Не удивительно, что его используют такие всемирно известные гиганты как Nasa, Google, Facebook, Intel, IBM, iRobot, Hewlett-Packard и др.

1.4.Синтаксис языка программирования Python

Чтобы начать писать даже простейшие программы на языке Python (и не только на нем), необходимо изучить хотя бы первичные основы синтаксиса языка. В противном случае компьютер просто не сможет понять и правильно выполнить программу. Например, если строку кода print('Привет, Мир!') записать в виде print ('Привет, Мир!'), интерпретатор выдаст ошибку, т.к. в программном коде мы поставили лишний пробел. При написании программного кода нужно строго соблюдать правила синтаксиса языка Python.

Синтаксис языка программирования – это набор правил, которые определяют структуру и порядок написания программного кода на этом языке.

Говоря о синтаксисе, мы подразумеваем правила написания идентификаторов и литералов, использования ключевых слов, разделения и оформления инструкций, комментариев и т.д. Уверенное знание всех этих правил, позволит писать программы любой сложности, допуская при этом минимум ошибок. А учитывая простоту и лаконичность синтаксиса Python, сделать это будет совсем не трудно. Но прежде давайте познакомимся с понятиями лексемы и литерала.

1.4.1. Лексемы и литералы в Python

Лексемы – это символы, или слова, которые не могут быть разбиты на более мелкие части. Лексемы - это те «кирпичики», из которых строятся фрагменты программного кода. Лексемами являются знаки арифметических операций: сложение «+», вычитание «-», умножение «*», деление «/», возведение в степень «**». Лексемами, например, являются ключевые слова: begin, end, if, while, for и т.п.

Литералы можно определить как данные, заданные в переменной или константе. Можно выделить следующие типы литералов:

- строковые;
- числовые;
- логические;
- специальные;
- литеральные коллекции.

Строковые литералы предназначены для формирования текстовой информации, их можно создать, заключив текст в кавычки. Можно использовать как одинарные, так и двойные кавычки для создания строки. Например, ‘Текстовая строка’, “Текстовая строка”.

В Python поддерживаются два типа строк:

- однострочная строка;
- многострочная строка.

Однострочные строки находятся в пределах одной строки программного кода. Например:

```
text1 = ‘Это пример однострочной строки программы’
text2 = “Это то же пример однострочной строки программы”
```

Есть два способа разбить фрагмент программного кода на несколько строк:

- добавление косой черты «\» в конце каждой строки;
- использование тройных кавычек.

Например:

```
text1 = ‘Это пример\
разбиения строки программного кода\
на несколько строк’
text2 = ‘‘‘Это то же пример
        разбиения строки программного кода
на несколько строк ‘‘‘
```

Числовые литералы предназначены для задания чисел. Числовыми литералами можно задать следующие четыре типа чисел:

- целые числа (int);
- длинные целые числа (long);
- действительные числа с дробной частью (float);
- сложные комплексные числа (complex).

Целые числа могут быть как положительными, так и отрицательными, У них нет дробной части, например, 100, 1500, -200, -1.

Длинные целые числа представляют собой последовательность цифр, за которыми следует строчная или прописная буква L, например: 870328454567893456L. Предпочтительно использовать прописную букву L, не строчную, поскольку строчная буква похожа и на единицу (1), и на прописную букву i.

У действительных чисел целая и дробная части отделяются друг от друга десятичной точкой, например: -26.2, 150.8, -34,25, 1500,15.

Сложные комплексные числа записываются в форме a + bj, где a – действительная часть, a b – мнимая часть комплексного числа. Ниже приведен пример создания двух комплексных чисел:

```
a1 = 2 + 2j
a2 = 3 + 4j
```

Логические литералы. Логический литерал может иметь только одно из двух значений: True (истина) или False (лож). Ниже приведен пример использования логических литералов:

```
x = True
y = False
if x == True:
```

Специальные литералы. Python имеет один специальный литерал – None (никто, ничто, ничего). None используется для указания некого пустого значения. Он также используется для конца списков в Python. Ниже приведен пример использования специального литерала:

```
a1 = None
a2 = None
```

Литеральные коллекции. Python предоставляет четыре типа коллекции литералов:

- List (список);
- Tuple (кортеж);
- Dict (словарь);
- Set (набор).

Список (List) – это набор либо однотипных элементов, либо элементов с разными типами данных. Данные, которые хранятся в списке, заключаются в квадратные скобки - [ ], а между собой разделяются запятой (,). Ниже приведен пример создания списков:

```
List1 = [100, 200, 300, 400]
List2 = [55.5, 33.8, 44.17, 105.5]
List3 = ['Иванов И.И.', 'Петров П.П.', 'Сидоров С.С.']
List4 = [1000, 55.8, "Семенов С.С."]
```

Списки можно корректировать в ходе работы программы.

Кортеж (Tuple) – это тоже набор либо однотипных элементов, либо элементов с разными типами данных. Данные, которые хранятся в кортеже, заключаются в круглые скобки ( ), а между собой разделяются запятой (,). В отличие от списков кортежи неизменяемы. Кортеж создается один раз и его нельзя изменить в процессе работы программы. Ниже приведен пример создания кортежей:

```
tup1 = (10, 20, 30, 40)
tup2 = (100.5, 20.2, 1000, 2000)
tup3 = ('Иванов И.И.', 'Петров П.П.', 'Сидоров С.С.')
tup4 = (10, 20.5, "Семенов С.С.", [2,3,4,5,6])
```

Словарь (Dict) - это набор либо однотипных элементов, либо элементов с разными типами данных. В словаре данные хранятся в виде пары значений: ключ-значение. Данные, которые хранятся в словаре, заключаются в фигурные скобки { }, а между собой разделяются знаком двоеточия (:). Ниже приведен пример создания словаря:

```
dict = {'Имя': 'Василий', 'Возраст':25, 'Образование': 'Высшее', 'Стаж работы (лет)': 3.5}
```

Набор (Set) – это неупорядоченная и неиндексированная группа элементов. Данные, которые хранятся в наборе, заключаются в фигурные скобки { }, а между собой разделяются запятой (,). Ниже приведен пример создания набора:

```
Fruits = {"Яблоки", "Бананы", "Апельсины"}
```

1.4.2. Имена в Python (идентификаторы)

Идентификаторы – это просто имена (например, переменных, функций или классов). В Python идентификаторы чувствительны к регистру, должны начинаться с любой буквы алфавита в Юникоде (в нижнем или верхнем регистре) или с символа подчеркивания (_), после чего могут следовать любые буквы, цифры или символы подчеркивания. При этом имена не должны начинаться с цифры.

Примерами идентификаторов могут служить: g (латинская буква), ж (русская буква), _g (начинается со знака подчеркивания), pyt_2 (включает цифру), _Фио (начинается со знака подчеркивания и включает русские буквы, одна из которых заглавная), S123 (начинается с заглавной буквы), set_file, setFile и т.д. Комбинации символов 15var_1, $Var2 или var 3 нельзя использоваться в качестве идентификаторов, так как первый начинаются с цифры, второй содержат недопустимый символ, третий содержит пробел. Также следует помнить про регистр используемых символов. Например, идентификаторы var_1, Var_1 и VAR_1 будут обозначать имена совершенно разных переменных или функций, так как регистр символов у них не совпадает. В Python допускается использовать символы кириллицы для задания имен. Однако это не желательно, поскольку, во-первых многие редакторы программного кода будут воспринимать такие идентификаторы как ошибку (при этом сама программа будет работать), во-вторых вполне вероятно, что в некоторых операционных системах или платформах программа действительно будет работать с ошибками. Использование символов латинского алфавита для задания имен будет гарантировать работоспособность программ в любой среде.

В Python идентификаторы (или переменные), это, по сути, имена объектов, в которых хранятся некоторые данные. Имена переменным придумывает программист, но есть несколько ограничений, связанных с именами переменных. В качестве имен переменных нельзя использовать ключевые слова, которые для Python имеют определенный смысл (в

редакторах программного кода ключевые слова выделяются определенным цветом). Набор ключевых слов, которые используются в Python, приведены на рис. 1.3.

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

Рис. 1.3. Ключевые слова в Python

Для лучшего восприятия кода следует давать именам понятные и по возможности краткие названия, которые бы соответствовали хранящимся в переменных данным или выполняемым функциями действиям (например, имя – name, файл – file, фамилия имя отчество - fio). Идентификаторы могут состоять не только из одной буквы или слова, но и из нескольких слов. Для таких случаев в программировании существуют два распространенных стиля записи идентификаторов. Можно два слова объединить в одно, написав каждое с заглавной буквы (FileName), а можно два слова разделить нижним подчеркиванием (file_name). Можно объединить эти два стиля (File_Name).

1.4.3. Отступы и точка с запятой в Python

Во многих языках программирования, например в C++, C#, конец строки обозначают символом точка с запятой (;). В отличие от них в Python каждая инструкция (или команда) не имеет символа, который обозначает конец строки. Общее правило Python гласит, что конец любой строки автоматически считается концом инструкции, находящейся в этой строке. В свою очередь это правило порождает другое, которое предписывает (за редким исключением) писать в каждой строке по одной команде.

Другим важным правилом языка Python является то, что вложенные инструкции одного уровня должны оформляться одинаковым количеством отступов от левого края. Именно по количеству отступов интерпретатор определяет, где заканчивается очередной блок инструкций и начинается новый. Общий шаблон оформления составных (вложенных) инструкций имеет вид:

```
Основная инструкция:
        Вложенный блок инструкций
        Вложенный блок инструкций
```

Как и в большинстве других языков программирования в Python присутствует условная инструкция if, которая в общем случае имеет следующий формат записи:

```
if Условие №1:
….Вложенный блок инструкций №1
 elif Условие №2:
….Вложенный блок инструкций №2
```

В данном примере вложенные блоки инструкций имеют отступы от начала строки 4 символа (здесь вместо пробелов отступы условно обозначены точками). Большинство редакторов программного кода автоматически формируют количество отступов от начала строки для вложенных инструкций. Обычно это 4 пробела, которые можно задать клавишей табуляции (Tab). Ниже приводится пример программного кода, демонстрирующий использование отступов:

```
def open_book():
   if name_db:
      sql = ''
      sql_bd = sql_meneg(name_db, sql)
      con_bd = sql_bd.connect_to_db()

      sql = '''SELECT name FROM sqlite_master WHERE type='table' AND name='Kniga';'''
      data = sql_bd.select_rec_all(con_bd, sql)
      if data == []:
         mb.showinfo("Ошибка", 'Данный файл не является интерактивной книгой')
         return
```

Примечание.

Здесь присутствует ключевое слово «def», которое объявляет пользовательскую функцию. О пользовательских функциях будет подробно рассказано в следующих главах.

Бывают ситуации, когда в программном коде имеется несколько коротких команд, которые идут одна за другой. В Python допускается записывать их в одной строке, разделяя команды точкой с запятой. Например:

```
a = 1; b = 2; c = 3
```

Однако такой стиль не приветствуется, и многие редакторы программного кода автоматически разносят такие команды в разные строки.

Бывают случаи, что инструкция очень длинная и не помещается на экране. В этом случае ее можно разнести на несколько строк, заключив в пару круглых, квадратных или фигурных скобок. Ниже приведены такие примеры.

Использование круглых скобок:

```
a = 1
d = 2
s = 3
n = (
   143.77 + 34 * 439.25 -
   23 / (a * d) + abs(-36 * s)
   )
```

Использование квадратных скобок:

```
a = 1
d = 2
s = 3
n = [
   143.77 + 34 * 439.25 -
   23 / (a * d) + abs(-36 * s)
   ]
```

Использование фигурных скобок:

```
a = 1
d = 2
s = 3
n = {
   143.77 + 34 * 439.25 -
   23 / (a * d) + abs(-36 * s)
   }
```

В официальном руководстве советуют использовать строки кода длиной не более 79 символов. Поэтому, если строка программного кода очень длинная, необходимо разбивать ее на несколько строк, используя скобки.

1.4.4. Комментарии в Python

Еще одной важной частью синтаксиса любого языка программирования является возможность вставлять в программный код комментарии и пояснения. В Python можно использовать однострочные и многострочные комментарии. Строка с однострочным комментарием начинаются с символа #. Весь код после символа # интерпретатором игнорируется. Комментарии можно вынести в отдельную строку, а можно разместить в строке с выполняемым программным кодом. Ниже приведен пример использования комментирующих строк.

```
# вывести текст из текстового файла в текстовое поле
self.Txt_print.configure(state='normal')    # разрешить редактировать
self.Txt_print.insert("1.0", f)             # вставить текст из переменной f
self.Txt_print.configure(state='disabled')  # запретить редактировать
```

Многострочные комментарии представляют собой текст, расположенный между тройными апострофами ('''строки с комментариями'''). Ниже приведен пример многострочного комментария:

```
def start_test():
   if lab_status['text'] == '':
      mb.showinfo("Нет открытой книги", 'В приложении нет открытых книг')
      return
   '''
   Обращение к модулю, который обеспечивает
   формирование тестов, используемых
   для контроля усвоения материалов пособия
   '''
   Made_Test(name_db)
   return
```

Текст комментариев интерпретатором игнорируется. Возможность вставки пояснений в программный код позволяет повысить его читабельность. Любой программист знает, что по истечении некоторого времени разобраться даже в собственном коде становится достаточно трудно. Поэтому в процессе разработки программ нужно обязательно использовать комментарии, которые позволяют:

- пояснить какую задачу решает тот или иной фрагмент программного кода;

- временно закомментировать фрагмент кода во время отладки программы;
- через некоторое время освежить в памяти программиста детали работы того или иного фрагмента программного кода;
- пояснить другим программистам назначение того или иного фрагмента программного кода (особенно это важно, когда группа программистов работает в команде).

В любом случае, правильно составленные комментарии ускоряют как разработку кода, так и его модификацию в дальнейшем.

1.5.Установка Python и запуск программ

Узнать последние новости о Python, познакомиться с официальной документацией, скачать установочный пакет Python можно на официальном сайте python.org.

Перед загрузкой интерпретатора внимательно ознакомьтесь с информацией на странице официального сайта. Убедитесь, что скачиваемая версия интерпретатора Python подходит для вашей операционной системы. После загрузки и установки интерпретатора можно приступить к работе с исходным кодом. Файлы программных модулей на Python имеют расширение «.py». При установке интерпретатора будет установлена и инструментальная среда для редактирования и запуска программного кода, так называемая IDE (от англ. *integrated development environment*). В операционной системе Windows эту инструментальную среду можно легко найти, если в строке поиска меню «Пуск» набрать ее имя - IDLE (рис. 1.4).

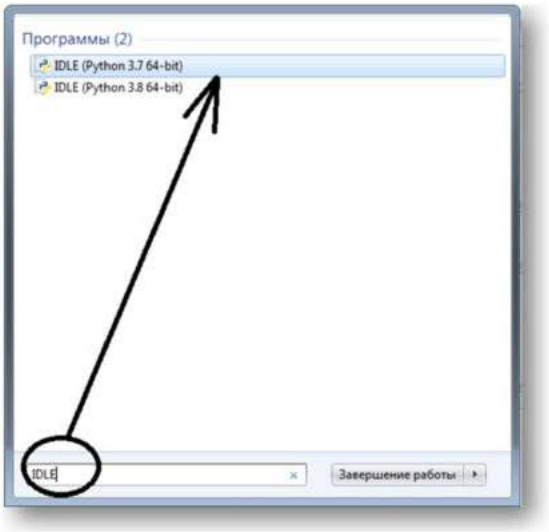


Рис. 1.4. Встроенная в Python инструментальная среда для работы с программным кодом

После запуска модуля IDLE откроется окно, в котором можно набирать программный код и выполнять программные модули (рис. 1.5) .

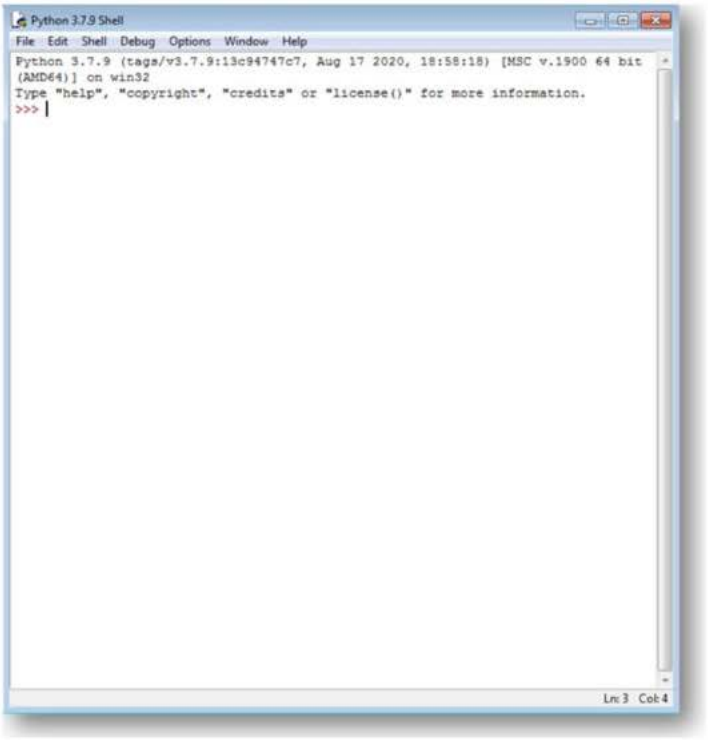


Рис. 1.5. Окно инструментальной среды IDLE для работы с программным кодом на Python

Инструментальная среда IDLE позволяет сохранить код программы во внешнем файле, который будет иметь расширение «.py», например, my_prog.py. Такую программу можно потом загрузить в IDLE и выполнить. В данном инструментарии можно работать только с короткими фрагментами программного кода. Для написания серьезных программ нужно использовать более удобные инструментальные средства. К ним можно отнести:

- PyCharm Community (the Python IDE for Developers – бесплатная или профессиональная версия);
- GIGA IDE от Сбера – альтернатива PyCharm;
- Visual Studio Code (бесплатная версия);
- Visual Studio community (бесплатная версия).

Наибольшей популярностью среди программистов пользуется IDE PyCharm, правда этот инструментарий существует только в англоязычном варианте. Редакторы кода Visual Studio имеют интерфейс на английском и русском языках.

При работе с данным цифровым электронным пособием вам ничего не нужно устанавливать на свой компьютер. Весь инструментарий уже встроен в интерактивную книгу. Когда вы изучите основы программирования на Python с использованием данного пособия, можно будет скачать и установить и интерпретатор Python, и одну из вышеуказанных IDE. В последней главе данного пособия можно найти инструкции по скачиванию и установке данного инструментария.

1.6. Первая программа на Python

В этой главе мы будем не только знакомиться с теорией, но и приступим к практическому программированию. Во всех книгах, учебниках и пособиям первой программой является программа приветствия, которая состоит из одной строки и выдает сообщение «Привет Мир!». Мы не будем отходить от этой традиции. Правда в связи с тем, что мы приступаем к освоению Python, наша программа приветствия выдаст сообщение «Привет Python!». Итак, в листинге 1.1 приведен текст такой программы.

Листинг 1.1. Первая программа на Python

```
# Листинг 1.1
print('Привет Python!')
```

В этой программе всего одна строка (не считая строки комментария). Здесь используется встроенная в Python функция print(), которая выводит на экран текстовую строку.

Теперь можно проверить работу этого программного модуля.

> **Примечание.**
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 1», и в выпадающем списке боковой панели выбрать раздел «Листинг 1.1. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 1.1).

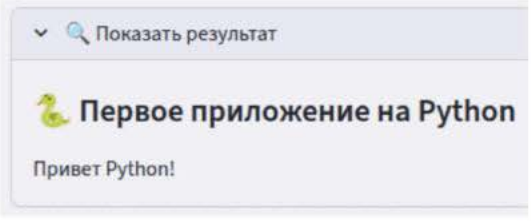


Рис. 1.1. Первое приложение на Python

Данный программный модуль, как и все последующие примеры программ, вы можете запустить на выполнение в рамках инструментария данного пособия (в программе BookReader).

1.7.Выводы к главе 1

Из материалов данной главы вы узнали: кем и когда был создан язык программирования Python, познакомились с возможностями и сильными сторонами этого языка программирования, узнали о сферах применения данного языка. Кроме того, изучили основы синтаксиса Python. По материалам данной главы нужно запомнить и усвоить следующие главные моменты.

1. В каждой строке кода следует писать только одну инструкцию (команду), не вставляя в конце строки никаких дополнительных символов.
2. После основной инструкции должно присутствовать двоеточие. При этом строки всех вложенных в основной блок инструкций должны иметь одинаковые отступы. Для этого нужно использовать по четыре пробела на каждый уровень вложенности или однократное нажатие на клавишу Tab.
3. Строки кода по возможности должны полностью помещаться в видимой области редактора программного кода. В идеале они не должны превышать длины в 79 символов. Для переноса длинных инструкций на новую строку в зависимости от ситуации следует использовать круглые, квадратные или фигурные скобки.
4. В большинстве случаев интерпретатор игнорирует подряд идущие пробелы и символы табуляции. Поэтому для визуального разделения некоторых лексем и повышения читабельности кода можно смело использовать пробелы. Однако при этом желательно придерживаться устоявшихся стандартов и, например, не ставить подряд несколько пробелов.
5. Комментарии в исходном коде можно писать как в отдельной строке, так и совмещать с командами (инструкциями). При этом принято отделять символ решетки от текста комментария пробелом.

Итак, из данной главы мы узнали об особенностях Python, написали и выполнили нашу первую программу. Теперь можно перейти в следующей главе и приступить к знакомству с типами данным, и выполнения над ними элементарных операций.

1.8. Вопросы и задания для самоконтроля



1. Найдите идентификатор, в котором есть ошибка:
   - 15var1
   - f
   - f2
   - ab_c
   - ABC).
2. Какой символ используется в Python для обозначения комментариев (#, $, &, %).
3. Какие из представленных утверждений верно для Python:
   - распространяется бесплатно;
   - компилируемый язык программирования;
   - тип переменной объявлять нужно обязательно;
   - массивы не являются объектами.

Глава 2. Типы данных в Python и элементарные действия с данными



В данной главе будет рассказано, как Python работает с переменными и какие типы данных можно использовать в рамках этого языка. Подробно рассмотрена модель данных Python, а также механизмы создания и изменения значения переменных.

Язык программирования может иметь динамическую или статическую типизацию данных. В первом случае тип переменной определяется непосредственно при выполнении программы, во втором – на этапе компиляции. Статическая типизация данных используется в таких языках программирования, как C, C#, Java. А вот Python – это динамически типизированный язык. Практически все данные, с которыми оперирует Python, это объекты различных типов: числа, строки, списки, кортежи, словари и множества. Эта глава посвящена особенностям работы с различными типами данных, в частности будут рассмотрены:

- базовые типы данных;
- работа с числовыми данными;
- обработка строковых данных;
- списки и обработка элементов списков;
- кортежи и обработка элементов кортежей;
- словари и обработка элементов словарей;
- логические типы данных и работа с ними;
- методы преобразования данных одного типа в данные других типов.

2.1. Базовые типы данных Python

Когда программист пишет программный код, то он, по сути, манипулирует различными типами данных, например, числами или строками. В Python, как и в других языках программирования, имеется как встроенный набор типов данных, так и возможность определять собственные, пользовательские типы данных. Все, чем оперирует Python, это объекты. Типы данных являются классами, а их значения - это экземпляры или объекты соответствующих классов. Каждое значение в Python имеет свой тип. В первой главе мы уже частично знакомились с типами данных, ниже приведен их перечень.

1. Числа – целые (-8, 0, 47), вещественные (-0.54 или -.54, 0.0, 2.475) и комплексные числа (-3 + 5j, 77 + 2j). Все они относятся к неизменяемым типам данных.
2. Строки – это текст произвольной длины, взятый в одинарные, двойные или тройные кавычки. Например, 'Текстовая строка', "Hello Python!", '''Строка в тройных кавычках'''. Строки также являются неизменяемым типом данных, хотя получить доступ к символу строки можно по его индексу.

3. Списки – это упорядоченные неименованные изменяемые коллекции значений (объектов), например, ['A', 'B', 'C'] или [2, [True, False], "house"]. Списки ограничиваются квадратными скобками, внутри которых через запятую перечисляются значения любых типов данных, включая и сами списки. Доступ к значениям списка осуществляется по индексам (об этом чуть позже). В Python нет таких понятий, как одномерный, двумерный или трехмерный массив. Списки – эти аналог массивов, но с более широкими возможностями. С использованием вложенных списков можно получить аналог массива любой размерности.
4. Кортежи – это упорядоченные неименованные неизменяемые коллекции значений (объектов), например, ('A', 'B', 'C') или (2, "house", True). Кортеж задается круглыми скобками, внутри которых через запятую перечисляются значения неизменяемых типов данных. Доступ к значениям элементов кортежа осуществляется по индексам. Но, так как кортежи являются неизменяемыми последовательностями, то добавлять или удалять элементы из них нельзя.
5. Словари – это именованные, изменяемые, неупорядоченные коллекции значений (объектов), например, {"Имя: 'Максим', 'Возраст': 25}. Словарь задается фигурными скобками, внутри которых через запятую перечисляются пары «ключ: значение». В качестве ключей в принципе можно использовать любые неизменяемые уникальные значения, например, числа или строки, а в качестве значений вообще любые типы объектов, включая другие словари. Доступ к значениям в словаре осуществляется по ключам.
6. Множества – это неименованные и неупорядоченные изменяемые коллекции уникальных и неизменяемых значений (объектов), например, {'Имя', False, 45}. Множество задается фигурными скобками, внутри которых через запятую перечисляются значения неизменяемых типов данных. Сами множества являются изменяемыми коллекциями значений, но вот значения в коллекции должны относиться к неизменяемым типам вроде чисел и строк.
7. Логические значения – этот тип данных содержит всего две булевых константы: True и False.
8. None – специальное значение, которое означает отсутствие значения.

Необходимо отметить, что встроенные типы данных можно классифицировать по нескольким признакам:

- изменяемые (списки, словари, множества);
- неизменяемые (числа, строки, кортежи);
- логические;
- пустые (None);
- упорядоченные (строки, списки, кортежи);
- неупорядоченные (словари, множества).

Эта классификация очень важна для понимания возможностей каждого типа данных и области их применения. Пока мы не будем углубляться в эти подробности, они будут подробно рассмотрены в следующих разделах пособия на конкретных примерах.

2.1.1. Динамическая типизация в Python

Как упоминалось ранее, Python относится к языкам программирования с динамической типизацией данных. Динамическая типизация – это прием, при котором переменная связывается с типом данных в момент присваивания значения, а не в момент объявления переменной. Это значит, что в Python программисту не нужно объявлять тип переменной в исходном коде, как это делается в других языках. Интерпретатор делает это автоматически во время присвоения переменной нового значения. При этом он самостоятельно контролирует объем памяти, распределяемый под вновь создаваемый объект.

Сами переменные в Python не имеют конкретного типа. Когда в коде программы переменной присваивается значение, интерпретатор создает объект, в котором хранятся и тип переменной, и значение переменной, и набор операций, допустимых для этого типа данных. Самой же переменной присваивается только ссылка на этот объект. Таким образом, в Python переменные не хранят сами значения, а только ссылаются на объекты присваиваемых им значений. Когда той же переменной будет присвоено другое значение, ей присваивается новая ссылка, указывающая на другой объект. А прежний объект, если он больше не используется в программе, просто удаляется. При этом программисту не нужно заботиться об удалении старого объекта, так как в Python используется механизм автоматической сборки мусора. Рассмотрим этот механизм на простом примере (листинг 2.1).

Листинг 2.1. Динамическая типизация в Python

```
# Листинг 2.1
a = 100
print(a)

a = 0.25
print(a)

a = 'Это первая текстовая строка'
print(a)

a = 'Это вторая текстовая строка'
print(a)

a = ['A', 'B', 'C']
print(a)

a = ('x', 'y', 'z')
print(a)

a = {'Имя': 'Максим', 'Возраст': 25}
print(a)

a = {'Имя', False, 45}
print(a)

a = True
print(a)

a = None
print(a)
```

В этом программном модуле переменной «a» присваиваются совершенно разные типы данных, после чего значения этих данных выводятся на экран.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 2» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.1. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.1).



Рис. 2.1. Динамическая типизация в Python

Из результатов работы программы видно, что переменная «a» в процессе работы программного модуля принимала значения совершенно разных типов данных, что подтверждает динамическую типизацию в Python.

Остановимся подробно на том, как Python работает с переменными. Здесь есть существенная особенность, которая отличает его от других языков программирования, где переменным явно задается тип (например, целое число) и он не меняется в процессе работы программы. В Python в момент выполнения присваивания, например a = 100, в памяти компьютера создается объект, расположенный по некоторому адресу (условно обозначим его как adr1). В ячейку с адресом adr1 будет занесено значение 100 целочисленного типа int. Затем создается переменная с именем «a», которая связывается с адресом объекта adr1. Переменные в Python содержат адреса объектов или можно сказать, что переменные ссылаются на объекты (рис. 2.2).

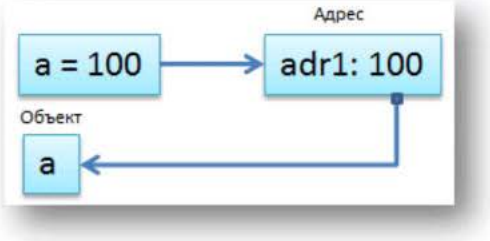


Рис. 2.2. Операция присваивания для новой переменной (создание нового объекта)

В момент присваивания той же переменной «a» другого значения, например строкового «Это первая текстовая строка», в памяти компьютера будет создан другой объект по адресу adr2 строкового типа, в который будет помещена заданная строка. После этого изменится

содержимое переменной «a», вместо адреса adr1 она получит ссылку на адрес adr2. Также Python увидит, что на объект по адресу adr1 больше никто не ссылается и поэтому удалит его из памяти. Python производит автоматическую сборку мусора (рис. 2.3).



Рис. 2.3. Операция присваивания для существующей переменной (создание нового объекта с новым адресом)

Как видно из данного рисунка, переменная осталась той же, но она теперь ссылается на другой адрес и уже имеет тип не целое число, а строка. Имея в виду эту модель, будем говорить, что переменная содержит значение.

2.1.2. О приведении типов в Python

Довольно часто в программе требуется преобразовать значение одного типа данных в другой. Например, когда пользователь вводит данные для вычислений в строковом виде, перед вычислениями возникает необходимость преобразования строки в число. Если этого не сделать, то мы получим либо ошибку, либо совсем неожиданный результат.

Рассмотрим следующий программный код:

```
a = 'Стоимость товара - '
b = 100
c = ' руб.'
d = a + b + c
print(d)
```

При выполнении этого программного кода должен быть выдан результат:

```
Стоимость товара - 100 руб.
```

Однако после запуска программы интерпретатор выдаст ошибку, так как мы пытаемся сложить две текстовые переменные с числовой переменной. Здесь число 100 нужно преобразовать в текст. Для такого преобразования используется встроенная функция str().

Изменим программный код следующим образом (листинг 2.2).

Листинг 2.2. Преобразование числа в строку

```
# Листинг 2.2
a = 'Стоимость товара - '
b = 100
b = str(b)
c = ' руб.'
d = a + b + c
print(d)
```

Здесь мы переменную b, в которой содержалось число 100, преобразовали в строку – '100'. Теперь переменной d будет присвоена сумма трех строковых переменных.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 2» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.2. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.4).

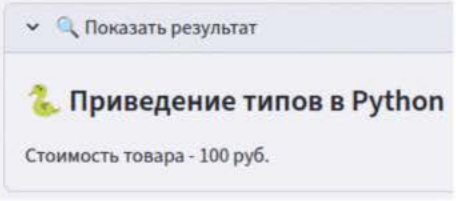


Рис. 2.4. Преобразование числа в строку

В данном примере для преобразования строки в число мы использовали встроенную функцию str(). Кроме нее в Python имеется набор других функций для преобразования типов данных: bool(), int(), complex(), tuple(), list(), dict(), set(). Каждую из них мы рассмотрим подробнее в других разделах этой главы.

Как можно было убедиться на примере, во избежание ошибок в процессе преобразования типов нужно следить за возможностью такого преобразования. В общем случае один тип числа легко преобразуется в другие типы чисел, один объект - в другие типы объектов, а логический тип можно преобразовать вообще во все что угодно.

2.1.3. Получение помощи по объектам Python

Ранее было отмечено, что язык Python основан на объектах: это данные, наборы данных и методы их обработки. Методы и данные называют атрибутами объекта. Но как можно получить доступ атрибутам? В Python доступ к атрибутам любого объекта осуществляется через точку, после которой следует имя атрибута. Чтобы узнать, какие атрибуты имеет объект, можно воспользоваться встроенной функцией dir(), которой в качестве аргумента нужно передать имя объекта. Данная функция выдаст словарь со всеми атрибутами объекта. Поскольку мы изучаем возможности Python, рассмотрим работу данной функции на простом примере. В этом примере будет использована еще одна встроенная в Python функция – len(). Эта функция позволяет получить количество элементов в какой-нибудь последовательности или коллекции (например, в строке или в словаре). Функции len()

нужно передать объект в качестве аргумента. Код программы с примером использования функций dir() и len() приведен в листинге 2.3.

Листинг 2.3. Получение доступа к атрибутам объектов Python

```
# Листинг 2.3
list1 = ['Миша', 'Саша', 'Маша']
print('--- Это исходный список ---')
print(list1)

print('--- Это атрибуты списка ---')
print(dir(list1))
print('Количество элементов в исходном списке- ', len(list1))

# Добавили в список два элемента
list1.append('Ваня')
list1.append('Таня')

print('--- Это список  с добавленными элементами ---')
print(list1)
print('Количество элементов в обновленном списке- ', len(list1))

# Очистим список
list1.clear()
print('--- Это пустой список ---')
print(list1)
print('Количество элементов в пустом списке- ', len(list1))
```

В данном программном модуле был создан список с именем list1, состоящий из трех строковых элементов, и этот список был выведен на экран. Затем были напечатаны атрибуты списка (объекта list1) с использованием функции dir(list1), и количество элементов в исходном списке - с использованием функции len(list1). Среди атрибутов объекта list1присутствуют такие методы манипулирования данными объекта как: append (добавить элемент в список), clear (очистить список). Воспользуемся этими методами. Для начала добавим в список два элемента:

```
# Добавили в список два элемента
list1.append('Ваня')
list1.append('Таня')
```

В этих командах было указано имя объекта (list1), после точки атрибут объекта (метод append – добавить элемент в список), а в качестве данных указана строка символов. После этого на печать были выведены: обновленный список, и количество элементов в обновленном списке.

Затем, с использованием метода clear была произведена очистка списка - list1.clear(). На печать вывели очищенный список и количество элементов в очищенном списке.

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 2» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел

«Листинг 2.3. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.5).

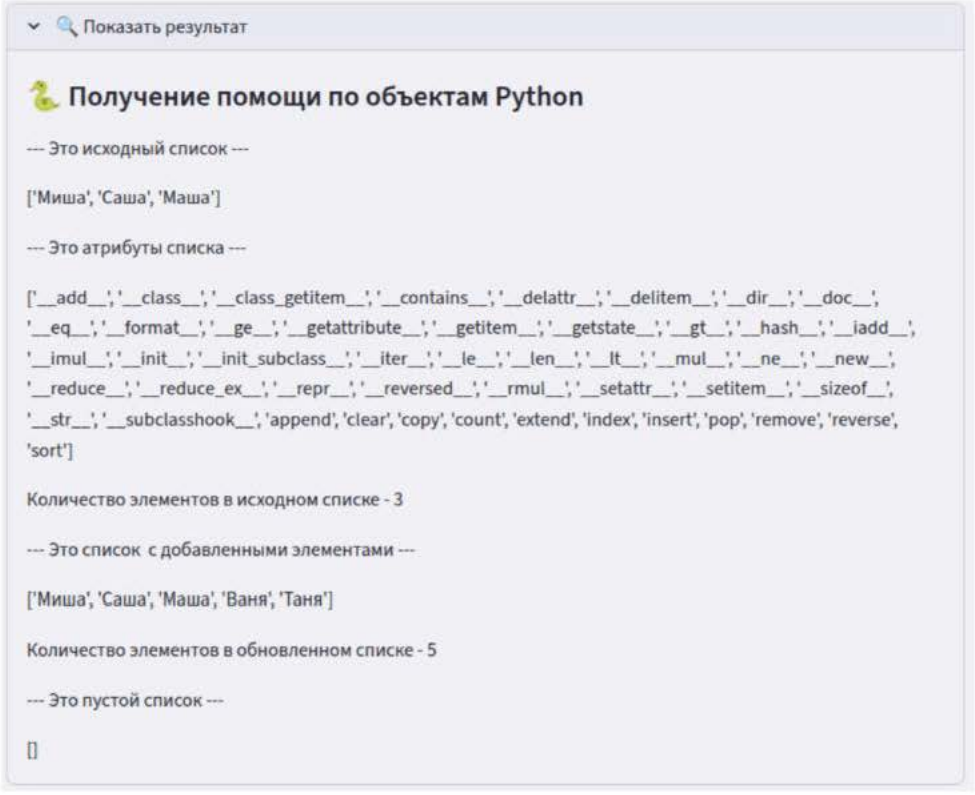


Рис. 2.5. Получение помощи по объектам Python

Функцией dir() удобно пользоваться в качестве подсказки тогда, когда вы работаете с простейшими редакторами кода. Когда программист работает с профессиональными редакторами кода, например с PyCharm, то редактор автоматический подскажет атрибуты объекта тогда, когда после имени объекта будет введена точка. При работе с такими редакторами необходимость использования функции dir() отпадает.

2.1.4. Краткие итоги раздела 2.1

1. Все данные в Python представлены в виде объектов, которые включают в себя как сами данные, таки и методы их обработки.
9. К базовым типам данных Python относятся числа (int, float, complex), строки (str), списки (list), кортежи (tuple), словари (dict), множества (set), булевы значения (bool) и специальное значение None (NoneType).
10. Python относится к языкам программирования с динамической типизацией данных, поэтому программисту не нужно объявлять тип переменной в исходном коде.

Интерпретатор делает это автоматически во время присвоения переменной нового значения. При этом он самостоятельно контролирует объем памяти, распределяемый под вновь создаваемый объект.

11. Переменные в Python хранят не само значение, а ссылки на объект этого значения.
12. Для преобразования одного типа данных в другой можно пользоваться встроенными функциями или конструкторами типов данных: bool(), int(), float(), complex(), str(), tuple(), list(), dict() или set(). Однако нужно быть внимательным и избегать ошибок, связанных с невозможностью преобразования одного типа данных в другой.
13. Для получения списка атрибутов объекта предназначена встроенная функция dir(), которой в качестве аргумента нужно передать требуемый объект. Результатом будет словарь со всеми атрибутами объекта.

На этом мы заканчиваем краткое описание типов данных Python, и переходим к более подробному рассмотрению каждого из перечисленных в этом разделе типу данных.

2.1.5. Вопросы для самоконтроля к разделу 2.1



1. Какой тип данных является числом: 123, '123', [1,2,3], {1,2,3}, {1: 1, 2: 2, 3: 3}, (1,2,3)?
2. Какой тип данных является строкой: '123', [1,2,3], {1,2,3}, 123, {1: 1, 2: 2, 3: 3}, (1,2,3)?
3. Какой тип данных является списком: [1,2,3], '123', {1,2,3}, 123, {1: 1, 2: 2, 3: 3}, (1,2,3)?
4. Какой тип данных является кортежем: (1,2,3), '123', [1,2,3], {1,2,3}, 123, {1: 1, 2: 2, 3: 3}?
5. Какой тип данных является словарем: {1: 1, 2: 2, 3: 3}, '123', [1,2,3], {1,2,3}, 123, (1,2,3).
6. Какой тип данных является множеством: {1,2,3}, '123', [1,2,3], 123, {1: 1, 2: 2, 3: 3}, (1,2,3)?

2.2.Обработка числовых данных в Python

Когда мы хотим провести какие-либо математические вычисления, обработать сведения о количестве перелетных птиц, сделать перевод денег между банковскими картами, мы оперируем с числами. В Python числа представлены несколькими типами это: целые числа (int), вещественные – с дробной частью (float), комплексные числа (complex). Если возможностей базовых типов чисел не хватает, всегда можно воспользоваться числовыми расширениями с использованием модулей decimal (десятичный) или fractions (дроби). Все перечисленные числовые типы довольно легко преобразуются между собой, а математические операции с ними интуитивно понятны и производятся по общепринятым правилам математики.

2.2.1.Целые числа

По умолчанию работы с целыми числами в Python используется десятичная система исчисления, однако разрешается записывать числа в двоичном, восьмеричном и шестнадцатеричном виде. При этом следует помнить, что вне зависимости от используемого формата представления, результат все равно будет отображаться в десятичном виде. Пример обработки целых чисел приведен в листинге 2.4.

Листинг 2.4. Операции с целыми числами

```
# Листинг 2.4
# Присвоили целое положительное число.
int_1 = 15
print('int_1=', int_1)

# Присвоили целое отрицательное число в двоичной системе
int_2 = -0b11001
print('int_2=', int_2)

# Присвоили целое отрицательное число в восьмеричной системе
int_3 = -0o34
print('int_3=', int_3)

# Присвоили целое положительное число в шестнадцатиричной системе
int_4 = 0xFC
print('int_4=', int_4)

# Результат сложения отобразится в десятичной системе
res = int_1 + int_2 + int_3 + int_4
print('Сумма=', res)
```

В этом программном модуле переменным int_1, int_2, int_3, int_4 были присвоены значения целых чисел в разных системах исчисления, а потом была подсчитана сумма этих чисел.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.4. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.6).

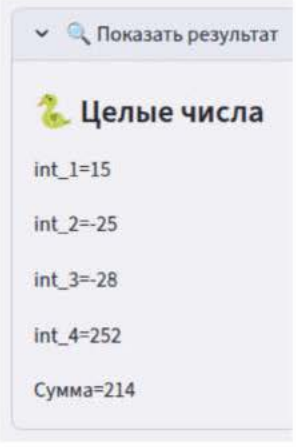


Рис. 2.6. Операции с целыми числами

Как видно из полученных результатов, на экран все числа были выведены в десятичном виде.

В Python синтаксис записи целых чисел в разрешенных системах счисления имеет следующий вид.

- в десятичной системе счисления целые числа записываются в виде привычной последовательности цифр от нуля до десяти, например, 0, -33, +127;
- в двоичной системе счисления запись целого числа должна начинаться с нуля и латинской буквы B в верхнем или нижнем регистре (0b или 0B), после чего должна идти последовательность нулей и единиц. Например, 0b01101, 0B1011, -0b1001. Если будут использованы другие цифры, интерпретатор сгенерирует сообщение об ошибке;
- в восьмеричной системе счисления целое число должно начинаться с нуля и латинской буквы O в верхнем или нижнем регистре (0o или 0O), после чего должна идти последовательность цифр от нуля до семи (01234567). Например, 0o77, 0O35, -0O174. Если будут использованы цифры не входящие в данный диапазон, интерпретатор сгенерирует сообщение об ошибке;
- в шестнадцатеричной системе счисления запись целого числа должна начинаться с нуля и латинской буквы X в верхнем или нижнем регистре (0x или 0X), после чего должны идти символы 0123456789ABCDEF, которые представляют данную систему счисления. При этом буквы также разрешается использовать в любом регистре. Например, 0x7F, 0XDD5A, -0xfF9. Если будут использованы символы не входящие в данный диапазон, интерпретатор сгенерирует сообщение об ошибке.

Во всех случаях разрешается использовать перед числом знаки плюса (+) и минуса (-).

Также можно переводить целые числа из одной системы исчисления в другую. Для этого следует использовать следующие встроенные функции:

- bin(int_num) – преобразует целое число в двоичную систему исчисления;
- oct(int_num) – преобразует целое число в восьмеричную систему исчисления;
- hex(int_num) – преобразует целое число в шестнадцатеричную систему исчисления.
- int(obj) – конструктор типа int может использоваться как с одним, так и с двумя аргументами. Если указан только один аргумент, то функция пытается преобразовать объект в целое число в десятичной системе исчисления, а в случае невозможности такого преобразования возбуждает соответствующее исключение. Если объект obj является вещественным числом, оно преобразуется в целое число путем усечения дробной части, например, int(21.302) вернет 21;
- int(num_str, base=10) – если указано два аргумента, то первый аргумент должен быть строковым представлением целого числа в системе счисления с указанным основанием base (второй аргумент). Разрешается использовать системы исчисления с основаниями от 2 до 36. При этом префикс 0b, 0o или 0x у строкового представления числа можно опускать. Если же точно известно, что в строковом представлении числа присутствует соответствующий префикс, то второму аргументу можно передать ноль, тогда интерпретатор сам определит систему исчисления переводимого числа (например, int('0xF5D', base=0)). В случае невозможности преобразования строки в целое число функция выдаст соответствующее исключение.

Пример программы конвертации целых чисел в разные системы исчисления приведен в листинге 2.5.

Листинг 2.5. Конвертации целых чисел в разные системы исчисления

```
# Листинг 2.5
# Присвоили переменной целое положительное десятичное число - 45
int_dec = 45
print('45 в десятичной системе-', int_dec)

# Преобразовали 45 в двоичную систему
int_bin_str = bin(int_dec)
print('45 в двоичной системе-', int_bin_str)

# Преобразовали 45 в восьмеричную систему
int_oct_str = oct(int_dec)
print('45 в восьмеричной системе-', int_oct_str)

# Преобразовали 45 в шестнадцатеричную систему
int_hex_str = hex(int_dec)
print('45 в шестнадцатеричной системе-', int_hex_str)

# Преобразовали 45 из шестнадцатеричной системы в десятичную
int_dec = int(int_hex_str, base=16)
```

```
print('45 в десятичной системе-', int_dec)

# Преобразовали 45 из шестнадцатеричной системы в десятичную
int_dec = int(int_hex_str, base=0)
print('45 в десятичной системе-', int_dec)
```

В этом программном модуле переменная int_dec получила значение 45, а затем это число было переведено в другие системы исчисления.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.5. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.7).

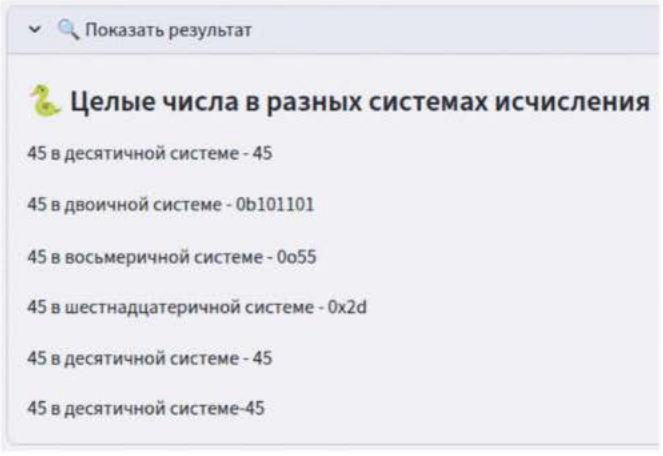


Рис.2 7. Целые числа в разных системах исчисления

Итак, функции bin(int_num), oct(int_num) и hex(int_num) принимают целые числа и возвращают строковые представления этих чисел в соответствующих системах счисления. Если же требуется обратное преобразование в десятичную систему, то следует передавать числа функции int(num_str, base=10), предварительно приведя число к строке (num_str = str(num)) и указав основание системы счисления переводимого числа.

Целые числа в Python отличаются от целых чисел других языков программирования тем, что их размер ограничивается лишь доступной памятью, т.е. они поддерживают длинную арифметику и могут быть сколь угодно большими. При этом следует помнить, что при обработке очень больших чисел приходится жертвовать скоростью их обработки.

Также следует отметить, что в отличие от других типов чисел над целыми числами в Python можно производить битовые операции:

- x | y - битовое or (ИЛИ);
- x ^ y - битовое xor (исключающее ИЛИ);

- x & y - битовое and (И);
- x << n - побитовый сдвиг влево;
- x >> y - побитовый сдвиг вправо;
- ~x - инверсия битов числа.

Подробнее вопрос битого преобразования целых чисел будет рассмотрен позже, при изучении битовых операций.

Также для работы с целыми числами типа int можно использовать несколько дополнительных методов:

- int.bit_length() – метод возвращает количество бит, которое необходимо для представления числа в памяти в двоичном виде, без учета знака и незначащих (лидирующих) нулей;
- int.to_bytes(length, byteorder, *, signed=False) – метод возвращает массив байтов, который соответствует данному числу;
- int.from_bytes(bytes, byteorder, *, signed=False) – метод класса, который возвращает целое число, соответствующее указанному массиву байтов.

Как видим, все дополнительные методы тоже касаются битов и байтов, поэтому мы их на данном этапе рассматривать не будем.

2.2.2. Вещественные числа

Что касается вещественных чисел (чисел с плавающей точкой), то в Python для них используется привычный синтаксис: сначала записывается знак числа (плюс можно не указывать), затем последовательность цифр целой части, разделительная точка и последовательность цифр дробной части. Например, 0.35, .44, 0.0, .0, -123.45, -.321. Также допустима запись вещественных чисел в экспоненциальной форме. При этом для представления степени 10n используется латинская буква «E» в верхнем или нижнем регистре («e» или «E»), затем следует необязательный знак плюс (+) или обязательный знак минус («-»), а также сам показатель степени в виде положительного целого числа или нуля. Пример программы работы с вещественными числами приведен в листинге 2.6.

Листинг 2.6. Представление вещественных чисел

```
# Листинг 2.6
# Положительное вещественное число
float_num_1 = 123.45
print('Положительное вещественное число:', float_num_1)

# Отрицательное вещественное число
float_num_2 = -123.45
print('Отрицательное вещественное число:', float_num_2)

# Отрицательное с пропущенным нулем
float_num_3 = -.45
print('Отрицательное с пропущенным нулем:', float_num_3)
```

```
# 123.45 в экспоненциальной форме
float_num_4 = 1.2345e2
print('123.45 в экспоненциальной форме:', float_num_4)

# 0.0012345 в экспоненциальной форме
float_num_5 = 1.2345E-3
print('0.0012345 в экспоненциальной форме:', float_num_5)
```

В этом программном модуле показаны возможные варианты представления вещественных чисел.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.6. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.8).

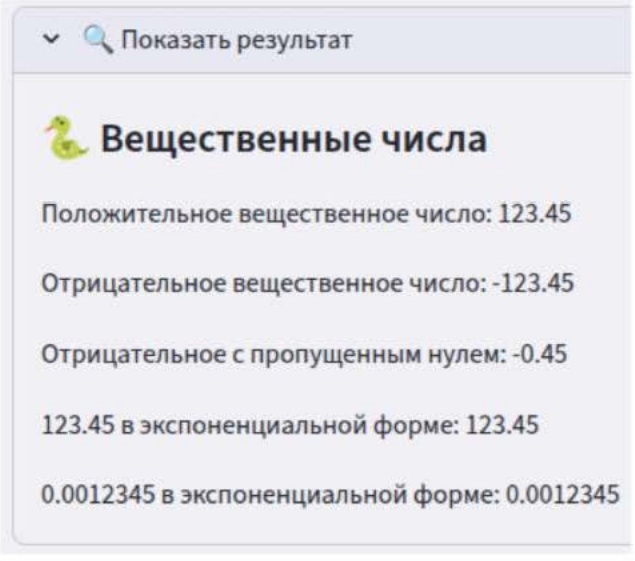


Рис. 2.8. Вещественные числа

Вещественные числа, в отличие от целых, не поддерживают длинную арифметику. В обычных условиях их точность ограничивается максимум 17 значащими цифрами. Если в расчетах необходима высокая точность представления вещественных чисел, следует использовать модуль decimal, позволяющий задавать требуемую точность результата (опять же, за счет снижения скорости обработки таких чисел). Пример программы, показывающей использования модуля decimal, приведен в листинге 2.7.

Листинг 2.7. Представление вещественных чисел с модулем decimal

```
# Листинг 2.7
# Импортируем модуль для задания фиксированной точности
import decimal

# Возведем вещественное число в 15 степень.
res = 55.25 ** 15  # Результат будет округлен до 17 значащих цифр
print('Простое возведение в степень:', res)

# Используем модуль decimal
res = decimal.Decimal('55.25') ** 15
print('Возведение в степень с использованием decimal:', res)

# Получим очень маленькое число
res = 0.0002777775793 / 995
print('Простое деление:', res)

# Установим точность в 35 значащих цифр.
decimal.getcontext().prec = 35
# Используем модуль decimal
res = decimal.Decimal('0.0002777775793') / 995
print('Деление с использованием decimal:', res)
```

Обратите внимание, что в этой программе мы впервые использовали процедуру импорта стороннего модуля. Модуль decimal – это программа, которая входит в поставку языка программирования Python. Мы из своей программы будем обращаться к методам обработки данных, которые реализованы в данном модуле. Для этого мы должны поместить текст модуля decimal в свою программу. В Python такое размещение выполняется с путем импорта программного модуля, для чего используется ключевое слово import. В начале своей программы мы написали строку: import decimal. Фактически мы дали команду - на место этой строки поместить программный код модуля decimal. Теперь в своей программе можно использовать методы, которые запрограммированы в модуле decimal. Как было отмечено ранее, для вызова того или иного метода нужно после имени модуля поставить точку (.), после которой указать нужный нам метод.

Итак, после импорта модуля decimal мы сначала число 55.25 возвели в степень 15, а потом выполнили туже операцию с использованием модуля decimal. Затем мы просто число 0.0002777775793 разделили на 995, а потом выполнили туже операцию с использованием модуля decimal. То есть в первом случае было получено очень большое число, во втором очень маленькое.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.7. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.9).

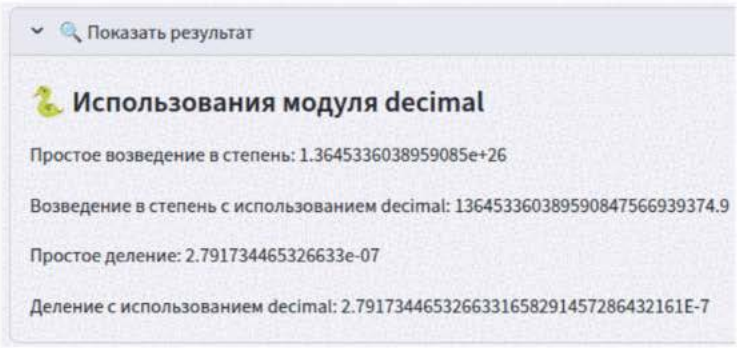


Рис.2. 9. Использование модуля decimal

Как видно из этих результатов, при использовании модуля decimal получена более высокая точность вычислений. Для традиционных вычислений вполне достаточно той точности, которая обеспечивается штатными средствами Python. Если же возникает необходимость работать с супер большими, или с супер малыми числами, то и здесь у Python есть весь необходимый инструментарий.

С вещественными числами можно выполнять следующие арифметические операции:

- сложение (x + y);
- вычитание (x - y);
- умножение (x * y);
- деление (x / y);
- возведение числа x в степень y (x ** y);
- целочисленное деление (x // y);
- получение остатка от деления (x % y);
- преобразование положительного числа в отрицательное (-x);
- преобразование отрицательного числа в положительное (+x).

Пример программы, показывающей использования арифметических операций с вещественными числами, приведен в листинге 2.8.

Листинг 2.8. Арифметические операции с вещественными числами

```
# Листинг 2.8
x = 10.5
y = 2.2

# Сложение
z = x + y
print('Сложение:', z)

# Вычитание
z = x - y
print('Вычитание:', z)
```

```
# Умножение
z = x * y
print('Умножение:', z)

# Деление
z = x / y
print('Деление:', z)

# Возведение в степень
z = x ** y
print('Возведение в степень:', z)

# Целочисленное деление
z = x // y
print('Целочисленное деление:', z)

# Получение остатка от деления
z = x % y
print('Получение остатка от деления:', z)
```

В этом программном модуле переменным «x» и «y» присвоены значения, а затем выполнены различные арифметические действия.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.8. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.10).

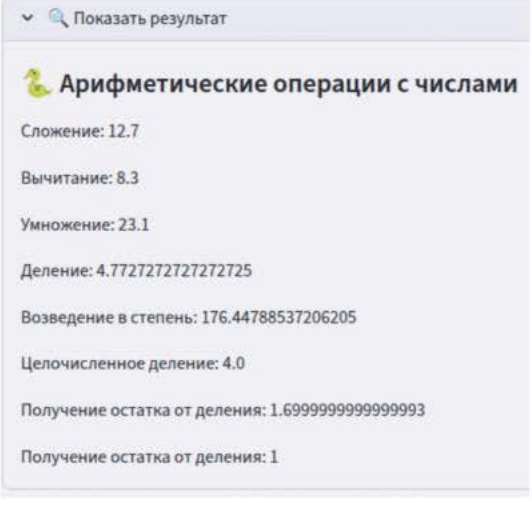


Рис. 2.10. Арифметические операции с числами

Вещественные числа можно обрабатывать встроенными функциями, например:

- abs(x) - получение абсолютного значения числа;
- eval('x') - преобразование символьного выражения в арифметические операции;
- int(x) - преобразование вещественного числа в целое;
- float(x) - преобразование числа или символьного значения X в вещественное число;
- max(x1, x2, x3, …') - нахождения «максимального» значения в последовательности чисел;
- min(x1, x2, x3, …') - нахождения «минимального» значения в последовательности чисел;
- sorted(x1, x2, x3, …') - сортировки последовательностей чисел.

Пример программы, показывающей обработку вещественных чисел встроенными функциями, приведен в листинге 2.9.

Листинг 2.9. Обработка вещественных чисел встроенными функциями

```
# Листинг 2.9
x = 10.5
# Получение абсолютного значения
z = abs(-x)
print('Получение абсолютного значения:', z)

# Преобразование строки в арифметическое выражение
z = eval('x + 20')
print('Преобразование строки в выражение:', z)

# Преобразование в целое
z = int(x)
print('Преобразование в целое:', z)

# Преобразование в вещественное
x = '25.3'
z = float(x)
print('Преобразование в вещественное:', z)

# Поиск максимального значения
z = max(10.3, 20.4, 40.5)
print('Максимальное значение:', z)

# Поиск минимального значения
z = min(10.3, 20.4, 40.5)
print('Минимальное значение:', z)

# Сортировка чисел
a = (100.1, 33.2, 20.8, 50.4)
print('Сортировка чисел:', sorted(a))
```

В этом программном модуле набор переменных получил результаты обработки чисел различными встроенными функциями.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.9. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.11).

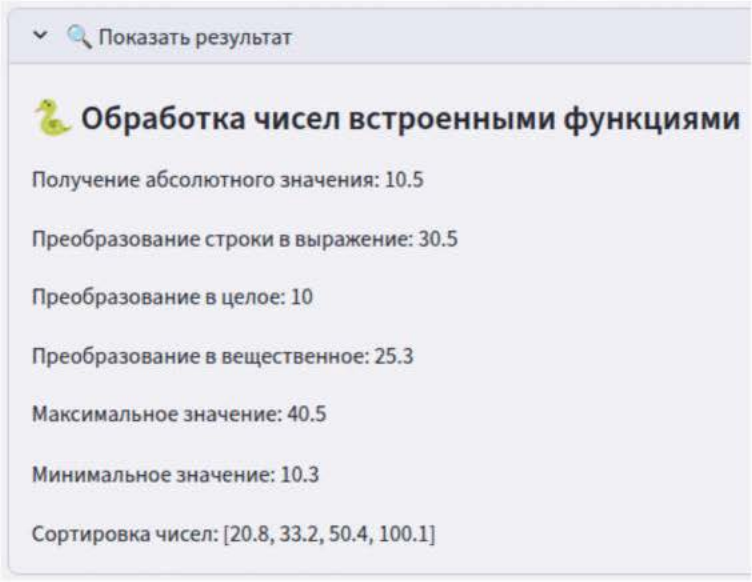


Рис.2 11. Обработка чисел встроенными функциями

В данном примере показано только несколько встроенных математических функций, с полным их перечнем можно познакомиться в документации на Python, и чуть позже мы изучим некоторые возможности модуля math стандартной библиотеки Python, который имеет расширенный набор математических функций. И не стоит забывать про целые библиотеки, которые смогут удовлетворить даже самых привередливых математиков.

Что касается математических выражений, в которых используются сразу несколько операций, то для их группировки предназначены круглые скобки. При этом внутри скобок приоритет отдается:

1. Возведению в степень (x**y).
2. Равнозначно умножению (x*y), делению (x/y), делению с округлением (x//y), получению остатка от деления (x%y)
3. Равнозначно выполняются сложение (x + y) и вычитание (x - y).

Если в выражении встречаются подряд идущие операторы с одинаковым приоритетом, то по умолчанию интерпретатор обрабатывает их слева направо. Однако, во избежание ошибок, старайтесь всегда использовать скобки, самостоятельно определяя порядок группировки, не полагаясь на приоритет или ассоциативность операторов.

Пример программы, показывающей использование группы операций с вещественными числами, приведен в листинге 2.10.

Листинг 2.10. Обработка вещественных чисел с группировкой арифметических операций

```
# Листинг 2.10
a = 100
b = 200
c = 2

# групповые операции без использования скобок
d = a + b + c + a*b + b/2 + a**2 - c
print('Вычисление без группировки d=', d)

# групповые операции с использованием скобок
d = a + b + c + a*b + b/(2 + a**2 - c)
print('Вычисление с группировкой d=', d)
```

В этом программном модуле для группировки арифметических операций использовались круглые скобки.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.10. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.12).

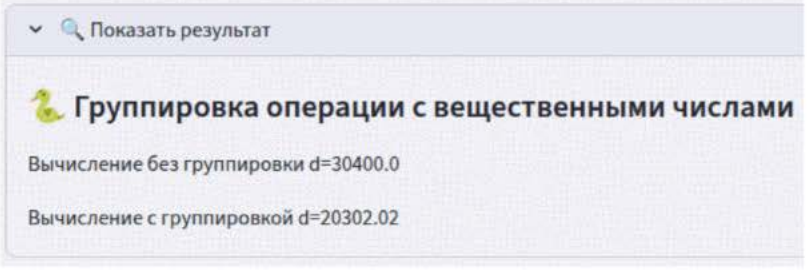


Рис. 2.12. Группировка операций с вещественными числами

Как видно из данного примера, всего две поставленные скобки привели к совершенно другому результату.

Для сравнения чисел, доступно 8 операций, причем все они имеют одинаковый приоритет:

- x < y - True если x меньше y, иначе False
- x <= y - True если x меньше или равно y, иначе False
- x > y - True если x больше y, иначе False
- x >= y - True если x больше или равно y, иначе False

- x == y - True если x равно y, иначе False
- x != y - True если x не равно y, иначе False
- x is y - True если x и y это один и тот же объект, иначе False
- x is not y - True если x и y это не один и тот же объект, иначе False

Пример программы, показывающей использование операций сравнения, приведен в листинге 2.11.

Листинг 2.11. Операции сравнения чисел

```
# Листинг 2.11
a = 10
b = 20
c = 10
print('a == b', a == b)
print('a == c', a == c)
print('a > b', a > b)
print('a < b', a < b)
print('a != b', a != b)
print('a != c', a != c)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.11. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.13).

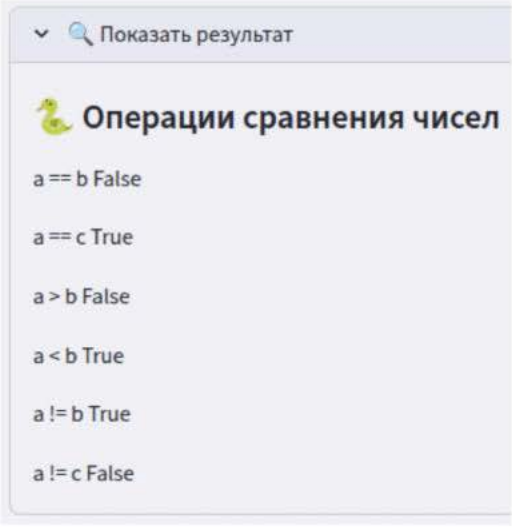


Рис. 2.13. Операции сравнения чисел

Этот пример достаточно условный, он просто показывает истинность или ложность приведенных в них выражений. Операций сравнения чисел в основном используются совместно с оператором разветвления «if», с которым мы подробно познакомимся чуть позже.

В дополнение к базовому оператору присваивания = в Python можно использовать комбинированный оператор присваивания, который объединяет операцию присваивания с другими операциями. Например, вместо выражения x = x + y можно использовать комбинированную операцию x += y. В Python можно использовать следующие комбинированные операции присваивания:

- присваивание (=);
- сложение и присваивание (+=);
- вычитание и присваивание (-=);
- деление и присваивание (/=);
- умножение и присваивание (*=);
- деление по модулю и присваивание (%=);
- возведение в степень и присваивание (**=);
- деление с остатком и присваивание (//=).

Пример программы, показывающей использование комбинированных операций присваивания, приведен в листинге 2.12.

Листинг 2.12. Использование комбинированных операций присваивания

```
# Листинг 2.12
a = 4
print('Присваивание (=):', a)

a = 4
a += 1
print('Сложение и присваивание (+=):', a)

a = 4
a -= 1
print('Вычитание и присваивание (-=)', a)

a = 4
a /= 2
print('Деление и присваивание (/=)', a)

a = 4
a *= 2
print('Умножение и присваивание (*=)', a)

a = 4
a %= 2
print('Деление по модулю и присваивание (%=)', a)
```

```
a = 4
a **= 2
print('Возведение в степень и присваивание (**=)', a)

a = 4
a //= 2
print('Деление с остатком и присваивание (//=)', a)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.12. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.14).

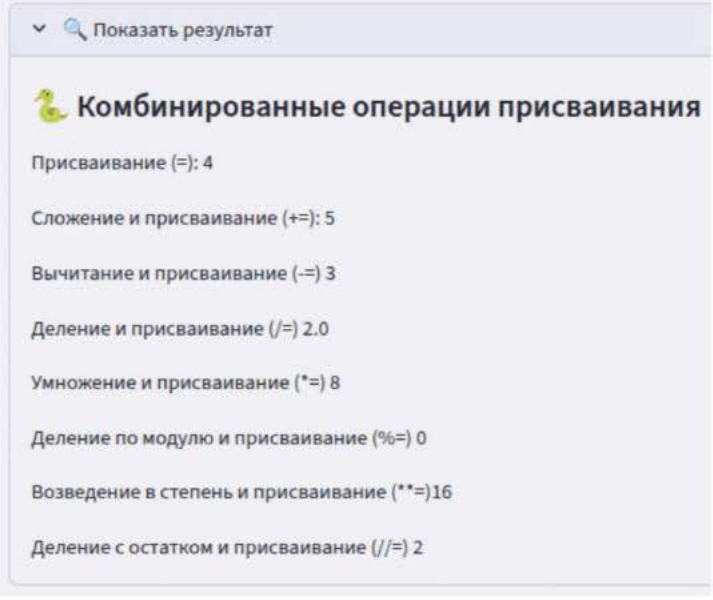


Рис. 2.14. Комбинированные операции присваивания

2.2.3. Комплексные числа

Для начала нужно сказать несколько слов о комплексных числах. Дело в том, что школах комплексные числа не изучают, так как считается, что это слишком сложно, а в институте считают, что не всем это нужно. Исторически комплексные числа впервые были введены в первой половине 16 века в связи с выведением формулы вычисления корней кубического уравнения, и сделал это итальянский математик Никколо Фонтана Тартальей. Но долгое время полученные им результаты считались сомнительными и лишь в 19 веке, после появления трудов немецкого математика Карла Фридриха Гаусса, существование комплексных чисел стало общепризнанным.

Комплексное число — это выражение вида a + bi, где a, b — действительные числа, а i — так называемая мнимая единица, или символ, квадрат которого равен –1, то есть i2 = –1. Число a называется действительной частью, а число b — мнимой частью комплексного числа. В математике комплексные числа принято обозначать буквой z (z = a + bi). Если b = 0, то вместо a + 0i пишут просто a. Отсюда следует, что действительные числа — это частный случай комплексных чисел.

Комплексные числа можно изобразить на комплексной плоскости (рис. 2.15).

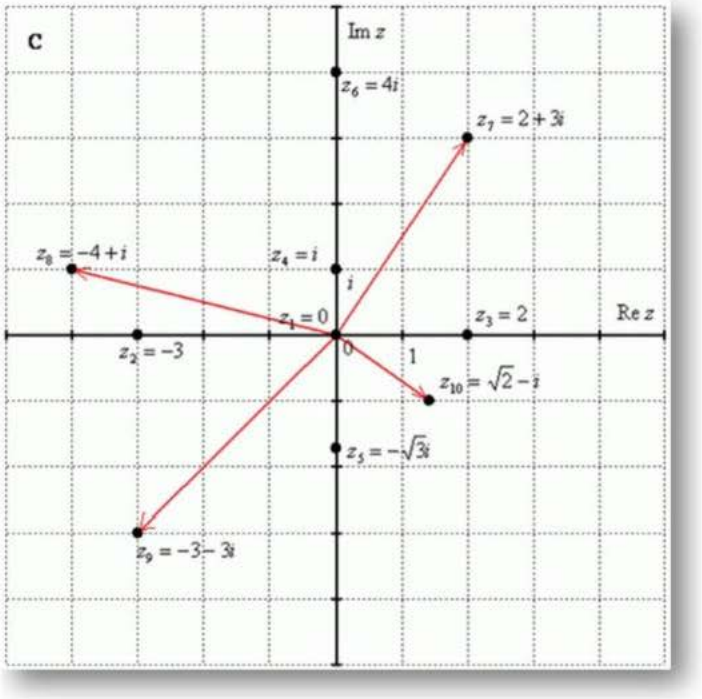


Рис. 2.15. Комплексная плоскость

Рассмотрим следующие комплексные числа: Z1=0, Z2=-3, Z3=2. Вы скажете, да это же обыкновенные действительные числа! И будете совершенно правы. Как было отмечено выше, действительные числа – это частный случай комплексных чисел. На оси Rez находится множество знакомых нам действительных чисел (это комплексные числа с нулевой мнимой частью).

Числа Z4, Z5 и Z6 – это, наоборот, чисто мнимые числа, т.е. числа с нулевой действительной частью. Они располагаются строго на мнимой оси Imz.

В числах Z7, Z8, Z9, Z10 и действительная и мнимая части не равны нулю. Такие числа тоже можно обозначить точками на комплексной плоскости.

С комплексными числами, как и с действительными, можно выполнять различные математические операции (складывать и вычитать, умножать и делить, извлекать корни и возводить в степень). Наиболее известные направления науки, где комплексные числа применяются на практике, это квантовая механика и электротехника.

Например, в электронике они используются для расчета цепей переменного тока. Все расчеты сводятся к сложению, умножению, вычитанию и делению комплексных чисел. А

без них пришлось бы решать интегрально-дифференциальные уравнения, что во много раз сложнее.

Комплексные числа в Python представлены типом complex, который хранит два вещественных числа: первое представляет собой действительную часть комплексного числа, а второе – мнимую. В математике действительная и мнимая части литерала комплексного числа объединяются знаком плюс или минус. Однако за мнимой частью в Python следует символ j, как это принято в инженерии (в математике мнимая единица обозначается через i).

Пример программы, показывающей выполнение операций с комплексными числами, приведен в листинге 2.13.

Листинг 2.13. Операции с комплексными числами

```
# Листинг 2.13
c1 = 1 + 2j
c2 = 2 + 4j
print('Сложение =', c1 + c2)
print('Вычитание =', c1 - c2)
print('Умножение =', c1 * c2)
print('Деление =', c1 / c2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.13. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.16).



Рис. 2.16. Операции с комплексными числами

Вполне возможно, что вам никогда не придется сталкиваться с комплексными числами, однако иметь представление о наличии такого типа данных все-таки не помешает. Вполне вероятно, что кто-то из вас будет заниматься наукой, работать в области ядерной физики, квантовой механики или электроники. В этом случае Python очень упростит вам процедуры оперирования с комплексными числами.

Сейчас же мы рассмотрим ряд математических функций модуля math, который входит в пакет стандартных библиотек Python.

2.2.4. Модуль math

Модуль math обеспечивает доступ к широкому кругу математических функций и констант, которые определены стандартом языка C. При этом модуль не поддерживает операции с комплексными числами, хотя они и определены стандартом языка Python. Но это не является большой проблемой, т.к. для операций с ними нужно просто воспользоваться модулем cmath, в котором легко найти требуемые функции, но уже с поддержкой комплексных чисел. Также стоит помнить, что, если явно не указано иное, все возвращаемые функциями модуля значения являются числами с плавающей точкой.

Для начала давайте рассмотрим представленные в модуле math математические константы:

- math.pi – представляет значение числа «**π**» с точностью, которая зависит от конкретной платформы. Мы его помним, как число 3.14;
- math.tau (чит. тау) – представляет удвоенное значение числа «**π**» с точностью, которая зависит от конкретной платформы (это отношение радиуса окружности к ее длине, приблизительно оно равно 6.28);
- math.e – представляет значение числа «**e**» с точностью, которая зависит от конкретной платформы (мы его помним, как число 2.72);
- math.inf (от англ. Infinity - бесконечность) – представляет положительную бесконечность.

Использование констант продемонстрировано в программном модуле, представленном в листинге 2.14.

Листинг 2.14. Использование констант модуля math

```
# Листинг 2.14
# импортир модуля math
import math

# Выведем число pi
print('Число pi =', math.pi)

# Выведем тау (удвоенное значение числа pi)
print('Число tau =', math.tau)

# Выведем число e
print('Число e =', math.e)

# Выведем бесконечность - inf
print('Положительная бесконечность:', math.inf)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.14. Результаты работы модуля можно увидеть, открыв контейнер «Показать

результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.17).

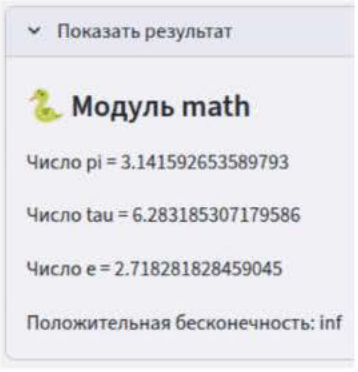


Рис. 2.17. Модуль math

В модуле math, представлен набор различных математических функций: логарифмические, тригонометрические, гиперболические и т.п. Все мы их здесь рассматривать не будем, приведем лишь короткий перечень наиболее употребительных:

1. **math.fsum** - сумма всех членов последовательности;
2. **math.exp**(X) – экспоненциальная функция (e^X);
3. **math.log1p**(X) - натуральный логарифм;
4. **math.log10**(X) - десятичный логарифм;
5. **math.pow**(X, Y) – возведение в степень (X^Y);
6. **math.sqrt**(X) – извлечение квадратного корня из X;
7. **math.sin**(X) - синус X (X указывается в радианах);
8. **math.cos**(X) - косинус X (X указывается в радианах);
9. **math.tan**(X) - тангенс X (X указывается в радианах);
10. **math.radians**(X) - конвертирует градусы в радианы.

Использование функций продемонстрировано в программном модуле, представленном в листинге 2.15.

Листинг 2.15. Использование функций модуля math

```
# Листинг 2.15
# импортир модуля math
import math

# Сумма
a = (1, 2, 3, 4, 5.5)
print('Сумма последовательности чисел =', math.fsum(a))
```

```
# Экспонента
print('Экспонента =', math.exp(10.5))

# Натуральный логарифм
print('Натуральный логарифм =', math.log1p(10.5))

# Десятичный логарифм
print('Десятичный логарифм =', math.log10(10.5))

# Возведение в степень
print('Возведение в степень =', math.pow(10.5, 2))

# Извлечение квадратного корня
print('Квадратный корень =', math.sqrt(100))

# Синус
z = math.radians(30)  # перевод градусов в радианы
print('Синус =', math.sin(z))

# Косинус
print('Косинус =', math.cos(z))

# Тангенс
print('Тангенс =', math.tan(z))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.15. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.18).

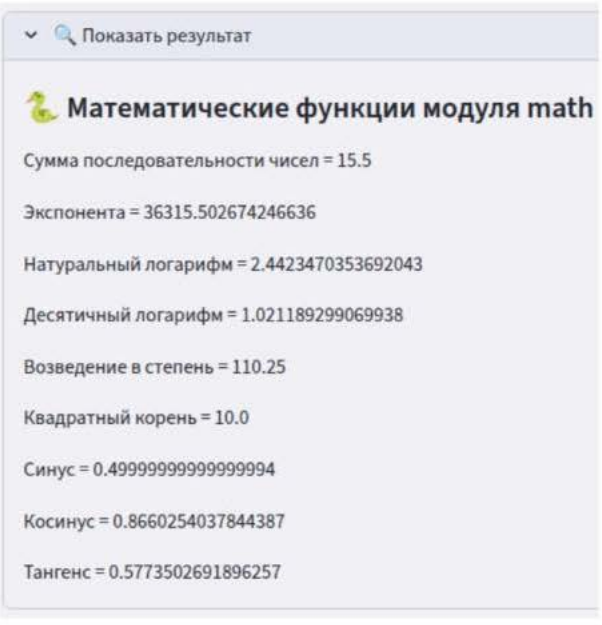


Рис. 2.18. Математические функции модуля math

Полный список функций модуля math можно получить на официальном сайте Python или в интернете в справочной литературе.

2.2.5. Модуль decimal

Потребность в максимальной точности расчетов на практике чаще всего возникает в отраслях и ситуациях, где некорректно выбранная точность расчетов может обернуться серьезными финансовыми потерями, например, при конвертации валют, в космической отрасли, или даже в отраслях пищевой промышленности. Дело в том, что при обработке чисел с плавающей точкой можно получить не совсем верный результат. Это демонстрирует программный модуль, приведенный в листинге 2.16.

Листинг 2.16. Оценка точности при выполнении арифметических операций

```
# Листинг 2.16
a = 0.1 + 0.1 + 0.1
print('a=', a)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.16. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.19).

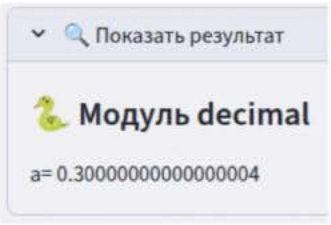


Рис. 2.19. Модуль decimal

Здесь мы должны были получить сумму – 0.3. Однако при работе этого программного модуля будет получен следующий результат.

То есть мы получили сумму 0.30000000000000004, вместо 0.3. Минимальная ошибка в 17 знаке после запятой не всегда, но все же может повлиять на итоговый результат цепочки вычислений. Для исключения таких ошибок можно использовать модуль decimal. Ключевым компонентом для работы с числами в этом модуле является класс Decimal. С использованием этого класса приведенную выше программу можно изменить следующим образом (листинг 2.17).

Листинг 2.17. Выполнение арифметических операций с модулем decimal

```
# Листинг 2.17
from decimal import Decimal
d = Decimal('0.1')
a = d + d + d
print('a=', a)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.17. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.20).

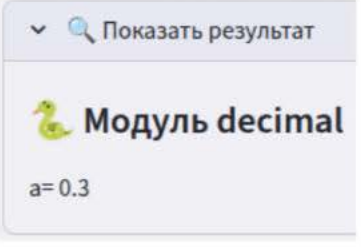


Рис. 2.20. Модуль decimal

То есть мы исключили появление возможных ошибок вычисления.

Из вышеприведенного примера видно, что, для создания числа фиксированной точности в Decimal нужно передать строку с указанием количества знаков после десятичной точки.

Если в выражении используются числа с различной точностью представления, то для результата интерпретатор выберет наилучшую точность представления. В принципе, в Decimal можно передавать не только строковые представления чисел, но и целые или вещественные числа, а также их кортежи.

2.2.6. Модуль fractions

Этот модуль пригодится в тех случаях, когда необходимо выполнить вычисления с дробями, при этом результат будет получен то же в формате дроби. Кроме того, он обеспечивает конвертирование десятичных чисел и текстовых значений в дроби. Ключевым компонентом для работы с дробями в этом модуле является класс Fraction. Пример программы с использованием класса Fraction приведен в листинге 2.18.

Листинг 2.18. Пример программы с использованием класса Fraction

```
# Листинг 2.18
from fractions import Fraction

# сумма дробей в результатом в виде дроби
f1 = Fraction('1/3')
f2 = Fraction('1/2')
print('f1=', f1, 'f2=', f2)
print('Сумма в виде дроби =', f1 + f2)
print('5/6 в десятичном виде это-', 5/6)
print('--------------------------------')

# сумма дробей в виде числа с плавающей точкой
a = 1/3
b = 1/2
print('a=', a, 'b=', b)
print('Сумма в виде десятичного числа=', a + b)

# Представление десятичного числа в виде дроби
print('--------------------------------')
print('Десятичное число 0.5 в виде дроби-', Fraction('0.5'))
```

Здесь мы выполнили несколько операций:

- два текстовых значения превратили в дроби, выполнили сложение дробей и получили результат в виде дроби;
- повторили туже операцию, используя только десятичные числа;
- десятичное число 0.5 превратили в дробь.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.18. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.21).

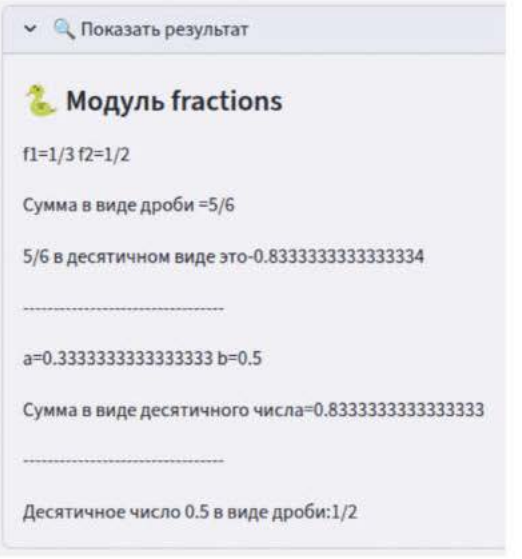


Рис. 2.21. Модуль fractions

В данном примере были выполнены следующие действия.

1. Сформировали две дроби (1/3 и 1/2), затем сложили их и в результате получили новую дробь (5/6). В десятичном виде это будет 0.83 … 34 .
2. Повторили туже операцию с десятичными числами и получили тот же результат (0.83… 34).
3. Конвертировали число 0.5 в дробь и получили результат – 1/2.

Как видно из данного примера, если работать с десятичными числами, то суммы, полученные при работе с «чистыми» дробями и полученные с десятичными числами, отличаются друг от друга в 16 знаке после запятой. Вроде бы мелочь, но при обработке финансовых данных эта мелочь может привести к серьезным неточностям.

Когда возникнет потребность оперирования дробями? К примеру, на кондитерской фабрике собираются готовить печенье по бабушкиному рецепту, в котором упоминается "положите 1/3 столовой ложки сахара". Сколько грамм составит эта треть, если применять ее не к одной порции печенья, а, например, к 1 тонне исходного сырья? А сколько это составит в килограммах, фунтах или унциях? Аналогичная потребность возникает при создании финансовых приложений, например при создании программ конвертации валют.

2.2.7. Краткие итоги раздела 2.2

1. Числа в Python представлены тремя базовыми типами: целые числа int, числа с плавающей точкой float и комплексные числа complex. Также доступны два модуля обработки чисел с фиксированной точностью Decimal, и операций с дробями Fraction.
2. Для записи целых чисел используется десятичная система счисления. Однако разрешается записывать их в двоичной (например, 0b101), восьмеричной

(например, -0o175) и шестнадцатеричной (например, 0xF5A) системах счисления. Для перевода целых чисел из одной системы счисления в другую имеется набор встроенных функции.

3. Целые числа поддерживают длинную арифметику и могут быть сколь угодно большими. При этом следует помнить, что при обработке очень больших чисел приходится жертвовать скоростью их обработки.
4. Для записи вещественных чисел используется привычный синтаксис, например, -2.57, 0.0, .28 (ноль перед точкой можно опускать). Также допустима запись вещественных чисел в экспоненциальной форме, например, 1.2345e2 (т.е. 1.2345*102) или -1.2345E-3 (т.е. -1.2345*10-3).
5. В обычных условиях точность вещественных чисел ограничивается максимум 17 значащими цифрами. Поэтому не стоит сравнивать их, доверяя точности до последнего знака.
6. Комплексные числа в Python хранят два вещественных числа: первое представляет собой действительную часть комплексного числа, а второе – мнимую. Например, -0.4+7j, 2-1.45j или 3.1j.
7. В Python поддерживаются все известные математические операции: сложение x + y, вычитание x - y, умножение x*y, деление x/y, деление с округлением x//y, получение остатка от деления x%y, возведение в степень x**y.
8. В дополнение к базовому оператору присваивания «=» в Python применяется целый ряд комбинированных операторов присваивания, которые объединяют операцию присваивания с другой операцией. Например, +=, -=, *=, /=, //=, %=.
9. Для сравнения чисел используются операторы <, >, <=, >=, == (равно), != (не равно).
10. Помимо стандартных арифметических операций имеется и ряд полезных встроенных функций для работы с числами:
    - pow(x, y) – возвращает x в степени y;
    - abs(x) – возвращает модуль числа x;
    - round(x, n) – округляет число до n-го разряда после запятой;
    - max(n_1, n_2, ..., n_n) - возвращает максимальное значение из последовательности чисел;
    - min(n_1, n_2, ..., n_n) – возвращает минимальное значение из последовательности чисел;
    - sum(n_1, n_2, ..., n_n) – возвращает сумму из двух и более чисел.
11. Если встроенных функций недостаточно, можно воспользоваться расширением math стандартной библиотеки, которое обеспечивает доступ к широкому кругу математических функций и констант.
12. Для случаев, когда требуется повышенная точность вычислений, в Python предусмотрены дополнительные объекты: для обработки чисел с фиксированной точностью Decimal, для работы с дробями - Fraction, которые явным образом хранят числитель и знаменатель рациональной дроби.

2.2.8. Вопросы для самоконтроля к разделу 2.2



☐1 Какие из литералов чисел относятся к типу int: 795, 3.4, 5.0, 0.0, 123+4.1j?
☐2 Какие из литералов чисел относятся к типу float: 3.4, 15, 123+4.1j, 795?
☐3 Какое число является десятичным: 755, 0xDD5A, 0b1101, 0o755?
☐4 Какое число является двоичным: 0b1101,755, 0xDD5A, 0o755?

5 Какое число является восьмеричным: 0o755, 755, 0xDD5A, 0b1101?
6 Какое число является шестнадцатеричным: 0xDD5A, 755, 0b1101, 0o755?
7 Чему будет равно A в выражении A = 2 + 4 / 2: (4, 1, 2, 3)?
8 Чему будет равно A в выражении A = (2 + 4) / 2: (3, 1, 2, 4)?

2.3. Строковые данные в Python

Для хранения и представления текстовой информации в Python используются строки, которые представляют собой отдельный тип данных str. Говоря о тексте, здесь подразумевается: содержимое текстовых файлов, исходный код программ, отдельные фразы, слова или даже символы.

Строки (от англ. strings) – это неизменяемые упорядоченные последовательности символов Юникода, которые используются в Python для хранения и представления текстовой информации. По умолчанию в скриптах используется кодировка UTF-8 - это одна из популярных и наиболее часто используемых кодировок символов.

Важно помнить, что строки в Python относятся к неизменяемым последовательностям. Отдельные символы в строке располагаются в строгом порядке слева направо. Строки индексированы, отдельный символ может быть получены при помощи своего индекса. Значение индекса для первого символа строки равно нулю. Добавить новый символ, изменить или удалить уже существующий символ не получится, можно только создать измененную копию строки.

Следует добавить, что строки в Python могут хранить и двоичные значения (0, 1). Это используется при обработке файлов изображений, аудиофайлов, а также при работе с упакованными данными. Для таких случаев доступны строковые типы bytes (строка байтов или по-другому неизменяемая байтовая строка) и bytearray (массив байтов или по-другому изменяемая байтовая строка). Оба типа более подробно будут рассмотрены в последующих разделах этого пособия.

2.3.1. Создание строк в Python

Создать строку в коде программы на Python довольно просто. Достаточно заключить требуемую последовательность символов в одинарные, двойные или тройные кавычки. Пример программы, демонстрирующей создание строк, приведен в листинге 2.19.

Листинг 2.19. Создание строк

```
# Листинг 2.19
# Строка в одинарных кавычках
str_1 = 'Это строка в одинарных кавычках.'
print(str_1)
print('---------------------------------')

# Строка в двойных кавычках
str_2 = "Это строка в двойных кавычках."
print(str_2)
print('---------------------------------')

# Используем три двойные кавычки
str_3 = """Это строка в тройных кавычках.
Может располагаться на нескольких строках."""
print(str_3)
```

```
print('----------------------------------')

# Используем три апострофа
str_4 = '''Это строка в тройных апострофах.
Может располагаться на нескольких строках.'''
print(str_4)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.19. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.22).



Рис. 2.22. Создание строк в Python

Использование для создания строк сразу нескольких видов кавычек обусловлено тем, что это дает возможность использовать в кавычках одного типа текст, заключенный в кавычки другого типа. При этом допускается использование кавычек любого разрешенного типа внутри строки, с кавычками того же типа. При этом кавычки внутри строки должны быть экранированы при помощи обратного слеша «\». Пример такого программного кода приведен в листинге 2.20.

Листинг 2.20. Экранирование строк

```
# Листинг 2.20
# Одинарные кавычки внутри двойных
str_1 = "Пример 'одинарных' кавычек внутри двойных."
print(str_1)

# Двойные кавычки внутри одинарных
str_2 = 'Пример "двойных" кавычек внутри одинарных.'
print(str_2)

# Двойные, одинарные и тройные одинарные внутри тройных
str_3 = """Это 'одинарные', "двойные", и
'''Тройные одинарные''' кавычки внутри тройных."""
print(str_3)

# Одинарные кавычки внутри одинарных
str_4 = 'Пример \'одинарных\' кавычек внутри одинарных.'
print(str_4)

# Тройные кавычки внутри тройных
str_5 = """Пример \"\"\"тройных кавычек\"\"\" внутри тройных."""
print(str_5)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.20. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.23).

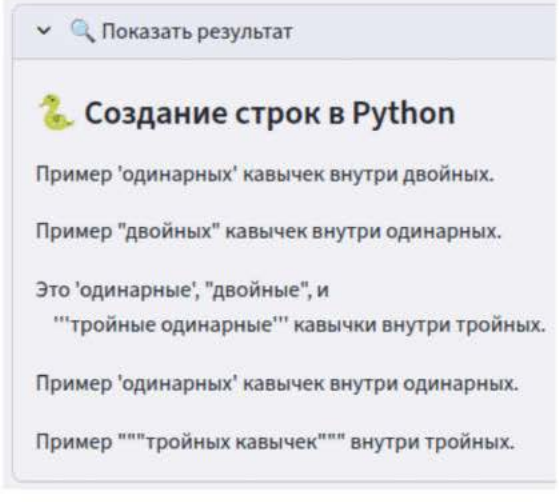


Рис. 2.23. Создание строк в Python

Если строка очень длинная, то возникает необходимость ее разбивки на несколько строк. Сделать это можно несколькими способами:

1. использовать обратный слеш «\» в требуемом месте переноса строки;
2. использовать строки в тройных кавычках, которые как раз и предназначены для оформления больших блоков текста внутри исходного кода;
3. разбить строку на несколько частей и затем объединить их при помощи знака **+**, при этом нужно заключить всю строку в круглые скобки.

О круглых скобках мы уже говорили раньше, их можно использовать для того, чтобы расположить длинную инструкцию на нескольких строках. В нашем случае такой инструкцией является объединение строки (конкатенация) при помощи оператора сложения.

Каждый оператор print() выводит информацию с новой строки. Однако у функции print() есть не обязательный аргумент end. С его помощью можно либо объединить две строки, выводимые разными операторами print(), либо вставить пустую строку после напечатанной, для этого аргументу end задаются следующие параметры:

- end = ‘ ‘ - (пробел) объединить две строки в одну;
- end = ‘\n\n‘ – вставить пустую строку после вывода очередной строки на экран.

Программный модуль, демонстрирующий описанные выше возможности, приведен в листинге 2.21.

Листинг 2.21. Оператор print() с аргументом end

```
# Листинг 2.21
# Используем обратный слеш для переноса строки.
str_1 = "Очень длинные блоки текста можно переносить \
на новую строку \nпри помощи обратного слеша."
print(str_1, end='\n\n')

# Используем для блоков текста тройные кавычки.
str_2 = """Наиболее удобный способ размещения
больших блоков текста сразу на нескольких строках -
это использование тройных кавычек."""
print(str_2, end='\n\n')

# Также строки можно переносить при помощи круглых скобок и знака +.
str_3 = ('Пример объединения ' +
         'трех строк программного кода ' +
         'в одну строку.')
print(str_3, end='\n\n')

# Объединить две строки, выводимые разными операторами print.
print('Это первый оператор print.', end=' ')
print('Это второй оператор print.')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.21. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.24).

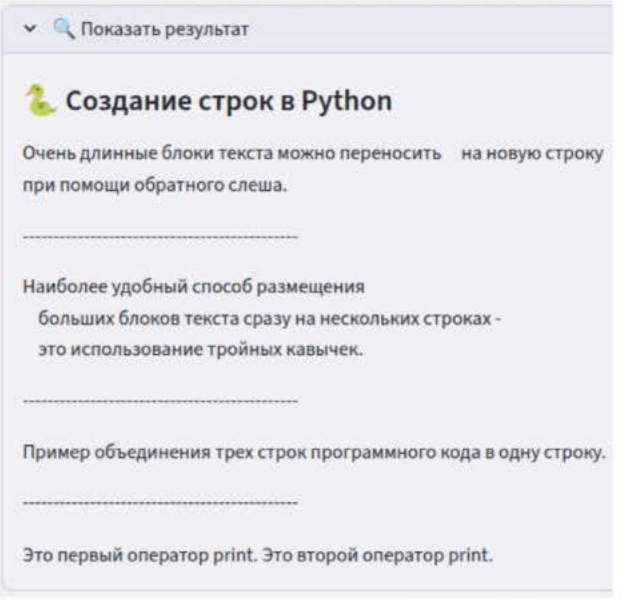


Рис. 2.24. Создание строк в Python

Как видно из данного примера использование кавычек разного типа, и использование аргумент end обеспечивает возможность, как разделения, так и объединения текстовых строк.

2.3.2.Экранирование выводимых символов

Не каждый символ можно свободно отобразить внутри выводимой строки. Поэтому в Python, как и во многих других языках программирования, имеется специальный набор экранированных (управляющих) последовательностей.

Экранированные последовательности — это последовательности, которые начинаются с символа "\" за которым следует один или более символов. Используемые выше символы - '\n' – это экранированная последовательность. Стоит отметить, что такие последовательности воспринимаются интерпретатором как единственный символ, т.е. несмотря на то, что последовательность может состоять из нескольких символов, ей соответствует всего один-единственный код.

Перечислим некоторые экранированные последовательности, используемые в Python:

- \0 – символ Null (не является признаком конца строки);
- \a – сигнал встроенного в систему динамика;

- \b – Backspace, т.е. возврат на одну позицию;
- \f – разрыв страницы;
- \n – перенос строки;
- \r – возврат курсора в начало строки;
- \t – горизонтальный отступ слева от начала строки (горизонтальная табуляция);
- \" – двойная кавычка;
- \' – одинарная кавычка;
- \\ – обратный слеш;
- \ooo – восьмеричный код (не более трех цифр);
- \uhhhh – Шестнадцатеричный код 16-битного символа Юникода (символ, кодируемый двумя байтами).

Программный модуль, демонстрирующий экранированные последовательности, приведен в листинге 2.22.

Листинг 2.22. Экранированные последовательности

```
# Листинг 2.22
# Горизонтальная табуляция
str_1 = 'Начало строки. \t Табуляция. \t\t Двойная табуляция.'
print(str_1)

# Символьная строка в восьмеричном коде.
s = '\167\157\162\154\144'
print('Строка в восмеричном коде -', s)

# Символьная строка в шестнадцатеричном коде.
s = '\x77\x6f\x72\x6c\x64'
print('Строка в шестнадцатеричном коде -', s)

# Здесь экранированная последовательность отсутствует.
s = 'Это будет очень\
... , очень-очень\
... , ну прямо ооооооочень\
...  длинная строка'
print(s)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.22. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.25).

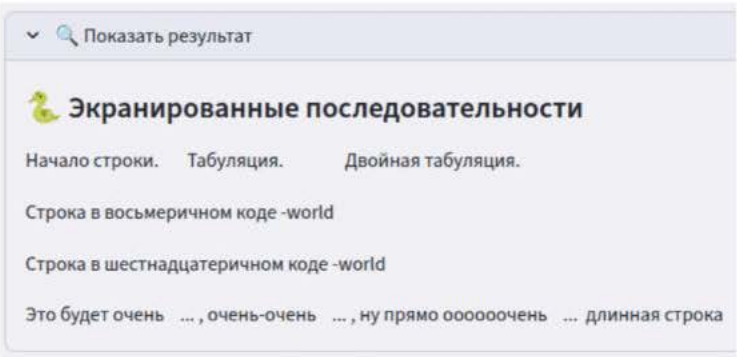


Рис.2.25. Экранированные последовательности

Как видно из данного примера, в последней строке обратный слеш ничего не экранирует, так как после него нет символа управляющей последовательности.

Бывают случаи, когда нужно запретить экранированные последовательности. Например, требуется вывести на печать путь к файлу, а в этом пути присутствует знак «\» - 'C:\python\new_file\test_1.py'. Здесь присутствует последовательность знаков «\n», и «\t». Интерпретатор воспримет их как перевод строки и горизонтальная табуляция. В таких ситуациях перед строкой нужно использовать символ «r», который отключает для конкретной строки механизм экранирования. Программный модуль, демонстрирующий отключение экранированной последовательности, приведен в листинге 2.23.

Листинг 2.23. Отключение экранированной последовательности

```
# Листинг 2.23
# Ничего не предпринимаем.
str_1 = 'C:\python\new_file\test_1.py'
print('Печать без подавления механизма экранирования')
print(str_1, end='\n\n')

# Подавляем механизм экранирования, используя сырую строку.
str_1 = r'C:\python\new_file\test_1.py'
print('Подавляем механизм экранирования символом  - "r"')
print(str_1, end='\n\n')

# Или экранируем сами слеши.
str_1 = 'C:\\python\\new_file\\test_1.py'
print('Подавляем механизм экранирования символом  - "\\"')
print(str_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.23. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.26).

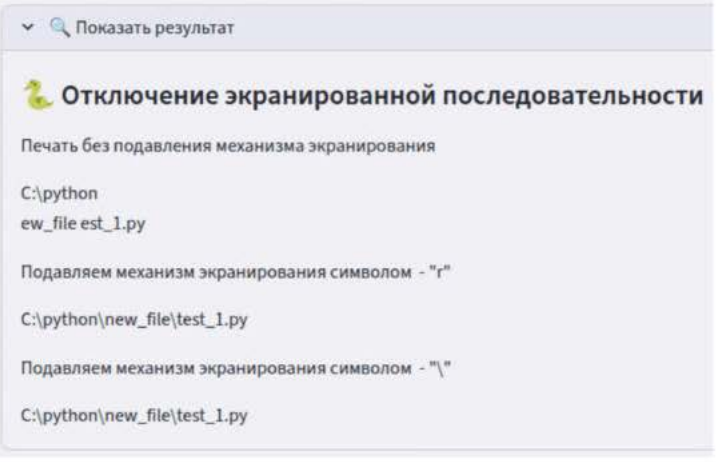


Рис.2.26. Отключение экранированной последовательности

Как видим, в нашем случае последовательность \n интерпретируется как символ новой строки, а последовательность \t – как символ горизонтальной табуляции. Ситуация исправляется либо за счет использования неформатированной строки (символ «r»), либо экранированием обратных слешей (символ «\»).

Следует добавить, что неформатированные строки не могут заканчиваться неэкранированным символом обратного слеша, т.к. он будет экранировать кавычки в конце строки. Например, при попытке выполнить код строки программы print('D:\text\') будет выдана ошибка. В таких случаях нужно экранировать все слеши ('D:\\text\\').

2.3.3. Доступ к символам строки по индексу и срезы строк

Как говорилось выше, строки в Python представляют собой неизменяемые упорядоченные последовательности символов Юникода. И хотя изменить строку нельзя, доступ к ее отдельным символам можно получить по индексу символа. Для этого нужно указать в квадратных скобках после имени или литерала строки порядковый номер символа, не забывая, что нумерация символов в строке начинается с нуля. В отличие от многих других языков программирования в Python доступ к символам строки возможен также и по отрицательным индексам. В этом случае можно считать, что отсчет ведется с конца строки и начинается с -1. Программный модуль, демонстрирующий доступ к символам строк, приведен в листинге 2.24.

Листинг 2.24. Доступ к символам строк

```
# Листинг 2.24
# Формируем строку
str_1 = 'Это строка'

# 1-й символ строки (нумерация начинается с нуля)
print('Первый символ строки ->', str_1[0])
```

```
# Последний символ строки(длина строки минус один).
print('Последний символ строки ->', str_1[len(str_1)-1])

# Можно использовать отрицательные индексы, тогда
# нумерация будет идти с конца и начинаться с -1
print('Предпоследний символ строки ->', str_1[-2])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.24. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.27).

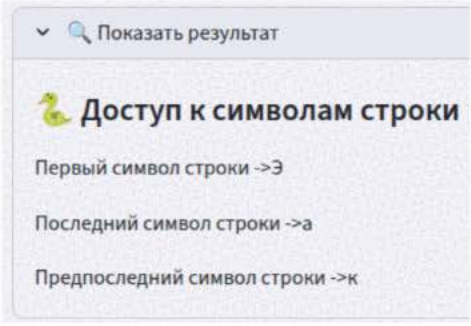


Рис. 2.27. Доступ к символам строки

Кроме того, в Python для строк (и других упорядоченных последовательностей) доступна операция извлечения среза, которая позволяет извлекать целые диапазоны символов, не прибегая при этом к специальным методам или встроенным функциям. Программный модуль, демонстрирующий извлечение группы символов по их индексам, приведен в листинге 2.25.

Листинг 2.25. Извлечение группы символов по их индексам

```
# Листинг 2.25
# Формируем строку
str_1 = 'Коля, Витя, Маша'

# Выводим первое слово
print('Первое слово - str_1[0:4] ->', str_1[0:4])
# Можно и так
print('Первое слово - str_1[:4] ->', str_1[:4])
print('-----------------------')

# Выводим второе слово
print('Второе слово - str_1[6:10] ->', str_1[6:10])
print('-----------------------')
```

```
# Выводим третье слово (используем отрицательные индексы)
print('Третье слово - str_1[-4:] ->', str_1[-4:])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.25. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.28).

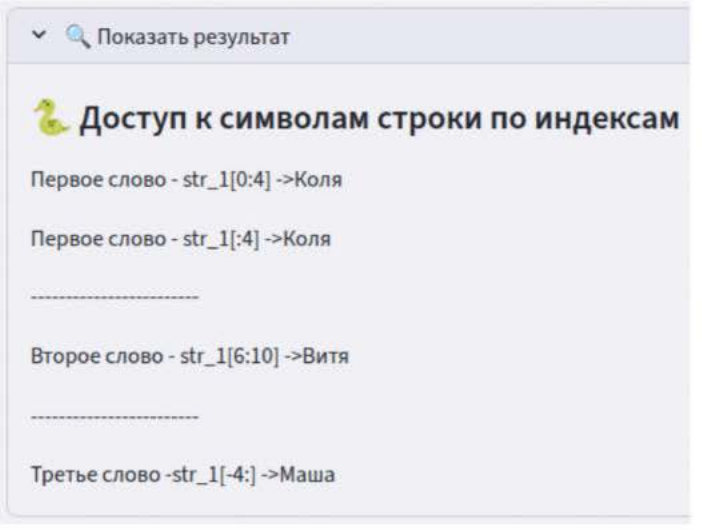


Рис. 2.28. Доступ к символам строки по индексам

При извлечении срезов следует помнить, что нумерация символов в строках начинается с нуля, а символ с индексом конца диапазона, указанного в квадратных скобках, не будет присутствовать в срезе. Например, в срезе str_1[0:5] будут присутствовать первые пять символов с индексами от 0 до 4, не включая символ с индексом 5.

В конце добавим, что к классу упорядоченных последовательностей относятся не только строки, но также списки и кортежи. Для них так же можно использовать операции индексирования и получения среза.

2.3.4. Арифметические операторы для обработки строк

Не смотра на то, что строки это набор символов, с ними можно выполнять некоторые арифметические операции. Например, с использованием оператора сложения «+» можно складывать, а вернее объединять несколько строк в одну. Соответственно комбинированный оператор «+=» добавляет новую строку к текущей строке. Еще одним арифметическим оператором, который применим к строкам, является оператор умножения «*», который обеспечивает многократное повторение текущей строки. Для этого можно использовать и соответствующий комбинированный оператор «*=». Программный модуль, демонстрирующий арифметические операции со строками, приведен в листинге 2.26.

Листинг 2.26. Арифметические операции со строками

```
# Листинг 2.26
# Формируем строки
str_1 = 'Первая строка '
str_2 = 'Вторая строка'
str_3 = '-'

# Сложение строк
print('Сложение строк ->', str_1 + str_2)

# Сложение строк
print('Умножение строки на 3 ->', str_1 * 3)

# Сложение строк комбинированным оператором
str_1 += 'Сложение строк комбинированным оператором "+="'
print('Сложение строк оператором "+=" ->', str_1)

# Умножением удобно выводить повторяющиеся символы
print('Умножение символа ->', str_3*30)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.26. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.29).

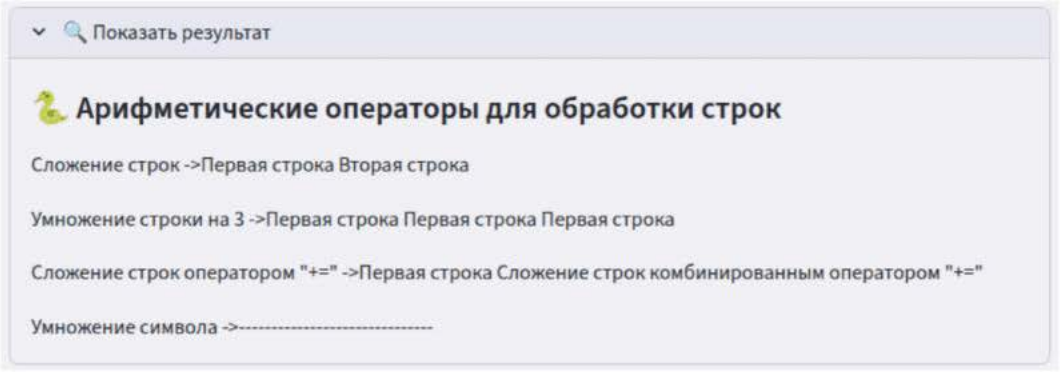


Рис. 2.29. Арифметические операции для обработки строк

Следует помнить, что смешивать числа и строки в выражениях сложения нельзя, т.к. интерпретатор не занимается автоматическим приведением типов и просто выдаст ошибку. В этих случаях числа нужно конвертировать в строки.

Теперь давайте познакомимся с инструкцией (оператором) in, которая позволяет найти в строке заданный фрагмент текста. Эта инструкция состоит из двух частей: левая часть (что

ищем), правая часть (где ищем). В случае успеха будет возвращено значение True, в противном случае – False. Программный модуль, демонстрирующий использование инструкции in, приведен в листинге 2.27.

Листинг 2.27. Использование инструкции in со строками

```
# Листинг 2.27
# Формируем строку
str_1 = 'Это строка символов'
print("Исходная строка->", str_1)

# Проверяем наличие слова в строке
answer = 'строка' in str_1
print("Наличие слова 'строка' ->", answer)

answer = 'Строка' in str_1
print("Наличие слова 'Строка' ->", answer)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.27. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.30).

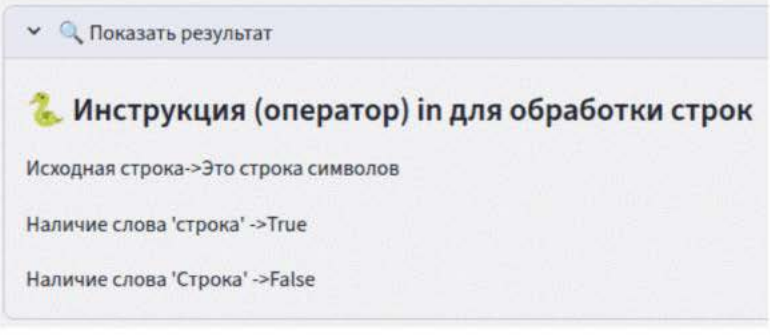


Рис. 2.30. Использование инструкции in со строками

2.3.5. Методы и функции для обработки строк

В Python для обработки строк имеется богатый набор методов и функций класса str. При этом нужно четко понимать разницу между терминами «метод» и «функция». При использовании метода программный код имеет последовательность – «имя_объекта.метод_класса». Например, str_1.upper(), str_2.lower(). При использовании функции программный код имеет последовательность: «имя_функции(a1, a2, a3, …)», где a – это аргументы, которые передаются в функцию. Далее в примерах данного пособия будут рассмотрены и методы, и функции для обработки строк.

Методы и функции для обработки строк можно условно разбить на несколько групп:

- изменение регистра в символах строки;
- поиск фрагментов текста в строке;
- выравнивание элементов строки;
- удаление пробелов и части строки;
- разбиение строки на фрагменты;
- анализ наличия или отсутствия определенных символов в строке;
- смена кодировки в строке символов;
- функции для определения длины строки и положения заданных символов в строке.

Последовательно рассмотрим эти методы и функции на конкретных примерах.

2.3.6.Изменение регистра в символах строки.

Для изменения регистров в символах строки используются следующие методы класса str:

- str.capitalize() – возвращает копию строки, переводя первый символ строки в верхний регистр, а все остальные в нижний;
- str.title() – возвращает копию строки, в которой первые символы каждого слова преобразованы в символы верхнего регистра, а все остальные символы - в символы нижнего регистра;
- str.upper() – возвращает копию строки, в которой все символы приведены к верхнему регистру;
- str.lower() – возвращает копию строки, в которой все символы приведены к нижнему регистру;
- str.swapcase() – возвращает копию строки, в которой все символы верхнего регистра преобразованы в символы нижнего регистра, а все символы нижнего регистра - в символы верхнего регистра.

Программный модуль, демонстрирующий использование методов изменения регистров, приведен в листинге 2.28.

Листинг 2.28. Использование методов изменения регистров строк

```
# Листинг 2.28
# Исходная строка символов
str_1 = "Это пРостая Строка символоВ"

# Перевод первого символа строки в верхний регистр
print('Первый символ в верхнем регистре->', str_1.capitalize())
print("Исходная строка->", str_1)
print('-'*40)

# Перевод первого символа каждого слова в верхний регистр
print('Первый символ каждого словав в верхнем регистре->',
str_1.title())
```

```
print('-'*40)

# Все символы в верхний регистр
print('Все символы в верхнем регистре->', str_1.upper())
print('-'*40)

# Все символы в нижний регистр
print('Все символы в нижнем регистре->', str_1.lower())
print('-'*40)

# Смена регистров
print('Смена регистров->', str_1.swapcase())
print('-'*40)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.28. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.31).

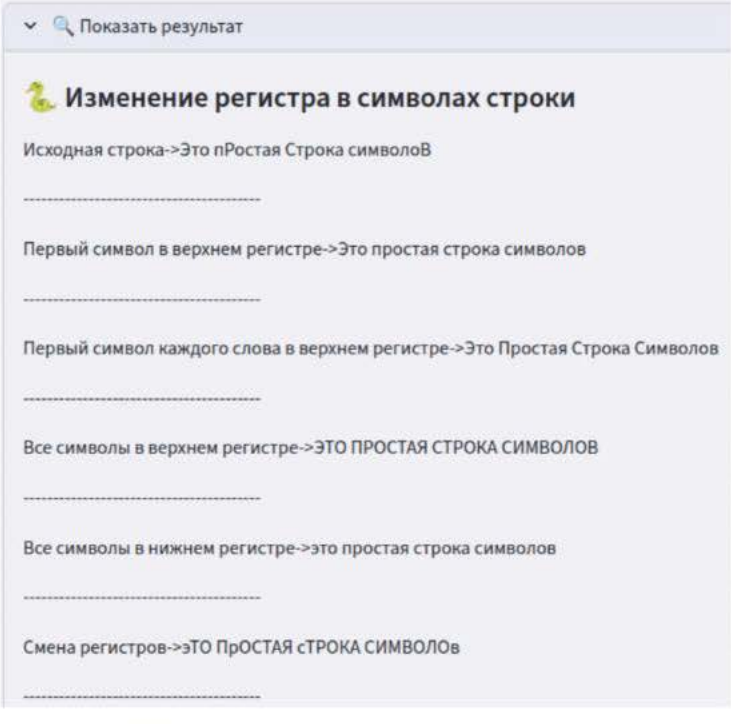


Рис. 2.31. Использование методов изменения регистров строк

В этом примере исходная строка была выведена на экран не сразу, а после того, как она была обработана методом - capitalize(). Этим мы показали, что данные методы возвращают копию строки, не меняя при этом текста исходной строки.

2.3.7. Поиск фрагментов текста в строке.

Для поиска фрагментов текста в строке используются следующие методы класса str.

1. str.count(sub[, start[, end]]) – возвращает число вхождений подстроки sub в строку str или в срез строки str[start:end], если он задан. При этом подсчитываются только непересекающиеся последовательности символов, то есть в строке «аааа» будет найдено всего две подстроки «аа».
2. str.find(sub[, start[, end]]) – возвращает индекс первого вхождения подстроки sub в строку str или в срез строки str[start:end], если он задан. В случае неудачного поиска метод возвращает -1. Обратите внимание, что при использовании среза отсчет индекса искомой подстроки все равно отсчитывается от начала всей строки (то же самое касается и аналогичных методов ниже).
3. str.index(sub[, start[, end]]) – также возвращает индекс первого вхождения подстроки sub в строку str или в срез строки str[start:end], если он задан. Однако в случае неудачного поиска метод вызывает исключение ValueError.
4. str.rfind(sub[, start[, end]]) – возвращает индекс последнего вхождения подстроки sub в строку str или в срез строки str[start:end], если он задан. В случае неудачного поиска метод возвращает -1.
5. str.rindex(sub[, start[, end]]) – также возвращает индекс последнего вхождения подстроки sub в строку str или в срез строки str[start:end], если он задан. Однако в случае неудачного поиска метод вызывает исключение ValueError.

Программный модуль, демонстрирующий использование методов поиска фрагментов текста в строке, приведен в листинге 2.29.

Листинг 2.29. Поиска фрагментов текста в строке

```
# Листинг 2.29
# Исходная строка символов
str_1 = "Это простая строка символов"
print("Исходная строка->", str_1)
print('-'*40)

# Сколько символов "о"
n = str_1.count('о', 1, 26)
print('Количество символов "о" ->', n)
print('-'*40)

# С какого символа начинается подстрока "простая"
n = str_1.find('простая', 1, 26)
print('Подстрока "простая" начинается с символа № ->', n)
print('-'*40)

# С какого символа начинается подстрока "симв"
```

```
n = str_1.index('симв', 1, 26)
print('Подстрока "симв" начинается с символа № ->', n)
print('-'*40)

# Где находится последний символ "о"
n = str_1.rfind('о', 1, 26)
print('Последний символ "о" ->', n)
print('-'*40)

# Где находится последний символ "с"
n = str_1.rindex('с', 1, 26)
print('Последний символ "с" ->', n)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.29. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.32).



Рис. 2.32. Поиска фрагментов текста в строке

2.3.8. Выравнивание элементов строки.

Для выравнивания элементов строки используются следующие методы класса str.

1. str.ljust(width[, fillchar]) – возвращает копию строки, выровненную по левому краю в строке с длиной width. Недостающие символы по умолчанию заполняются пробелами или символом fillchar, если данный необязательный аргумент будет задан. Кроме того, если width < len(str), то будет возвращена копия исходной строки без каких-либо изменений.
2. str.rjust(width[, fillchar]) – возвращает копию строки, выровненную по правому краю в строке с длиной width. Недостающие символы по умолчанию заполняются пробелами или символом fillchar, если данный необязательный аргумент будет задан. Кроме того, если width < len(str), то будет возвращена копия исходной строки без каких-либо изменений.
3. str.center(width[, fillchar]) – возвращает копию строки, отцентрированную в строке с длиной width. Недостающие символы по умолчанию заполняются пробелами или символом fillchar, если данный необязательный аргумент будет задан. Кроме того, если width < len(str), то будет возвращена копия исходной строки без каких-либо изменений.
4. str.expandtabs(tabsize=8) – возвращает копию строки, в которой символы табуляции замещены пробелами с шагом 8 (по умолчанию) или в соответствии со значением необязательного аргумента tabsize. Если какая-то подстрока не помещается внутри одного столбца, то ширина данного столбца будет увеличена до необходимых размеров.
5. str.zfill(width) – возвращает копию строки, в которой исходная строка str дополняется нулями слева так, чтобы длина новой строки стала равна width.

Программный модуль, демонстрирующий использование методов выравнивания элементов строки, приведен в листинге 2.30.

Листинг 2.30. Выравнивание элементов строки

```
# Листинг 2.30
# Исходная строка символов
str_1 = "Это простая строка символов"
print("Исходная строка->", str_1)
print('-'*60)

# Дополнение знаками "." в конце строки до 35 символов
str_2 = str_1.ljust(35, ".")
print('Знаки "." до 35-го символа в конце строки ->', str_2)
print('-'*60)

# Дополнение знаками "." в начале строки до 35 символов
str_2 = str_1.rjust(35, ".")
print('Знаки "." до 35-го символа в начале строки ->', str_2)
print('-'*60)

# Дополнение знаками "." в начале и в конце строки до 35 символов
str_2 = str_1.center(35, ".")
print('Знаки "." до 35-го символа в начале и в конце строки ->', str_2)
```

```
print('-'*60)

# Замена символов табуляц пробелами
str_5 = "0\t1\t2\t3\t4\t5\t6"
print('Вставить 2 пробела ->', str_5.expandtabs(tabsize=2))
print('Вставить 4 пробела ->', str_5.expandtabs(tabsize=4))
print('-'*60)

# Дополнение нулями "0" в начале строки до 35 символов
str_2 = str_1.zfill(35)
print('Дополнение нулями слева ->', str_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.30. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.33).

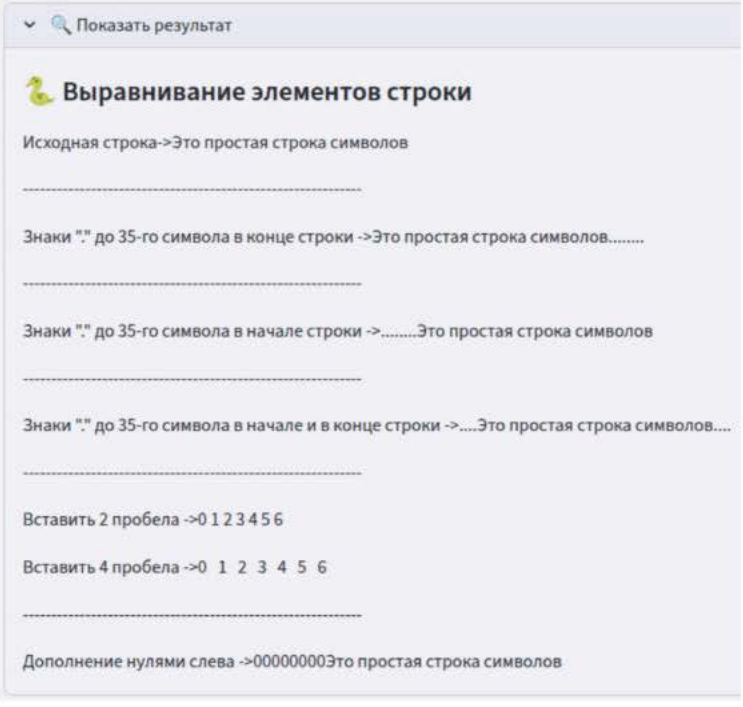


Рис. 2.33. Выравнивание элементов строки

2.3.9. Удаление пробелов и части строки.

Для и удаления пробелов и части строки используются следующие методы класса str.

- str.replace(old, new[, count]) – возвращает копию строки, в которой все подстроки old заменены на подстроки new. Если указан аргумент count, метод производит только указанное количество замен.
- str.strip([chars]) – возвращает копию строки, из которой удалены начальные и завершающие пробельные символы (по умолчанию) или символы, входящие в строку chars.
- str.lstrip([chars]) – возвращает копию строки, из которой удалены начальные пробельные символы (по умолчанию) или символы, входящие в строку chars.
- str.rstrip([chars]) – возвращает копию строки, из которой удалены завершающие пробельные символы (по умолчанию) или символы, входящие в строку chars.

Программный модуль, демонстрирующий удаление пробелов и части строки, приведен в листинге 2.31

Листинг 2.31. Удаление пробелов и части строки

```
# Листинг 2.31
str_1 = "Красная машина"
print("Исходная строка->", str_1)
print('-'*60)

# Заменая части строки
str_2 = str_1.replace("машина", "роза")
print('Обновленная строка ->', str_2)
print('-'*60)

# Удаление пробелов в начале и конце строки
str_1 = "   Хорошая машина   "
str_2 = str_1.strip()
print('Удаление пробелов слева и справа ->', str_2)
print('-'*60)

# Удаление пробелов слева
str_1 = "   В строке есть пробелы слева"
str_2 = str_1.lstrip()
print('Удалены пробелы слева->', str_2)
print('-'*60)

# Удаление пробелов справа
str_1 = "В строке есть пробелы справа"
str_2 = str_1.rstrip()
print('Удалены пробелы справа->', str_2 + "-Новая строка")
print('-'*60)

# Удаление заданных символов
```

```
str_1 = "<><><>Комментарии<><><>"
str_2 = str_1.strip('<>')
print('Удалены символы "<>" ->', str_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.31. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.34).

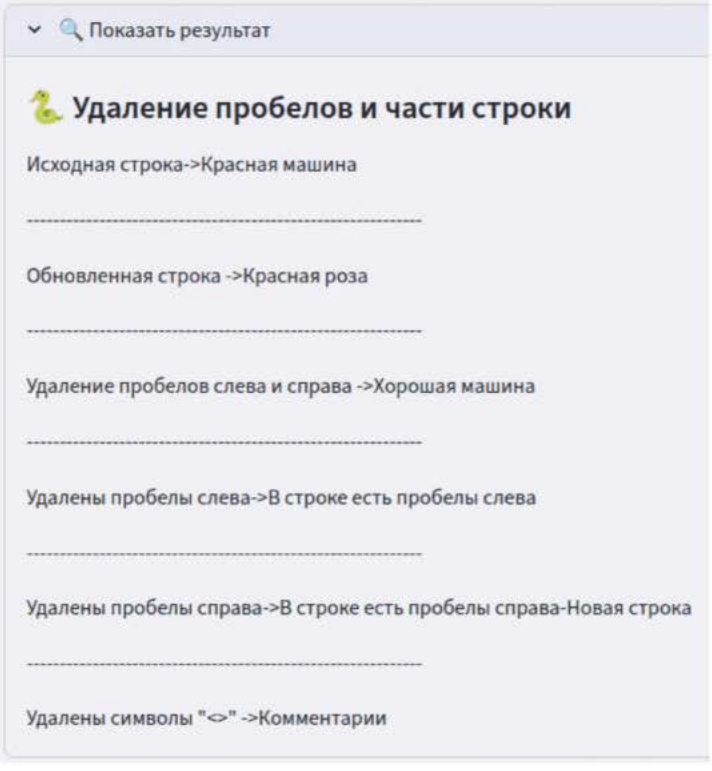


Рис. 2.34. Удаление пробелов и части строки

2.3.10. Разбиение строки на фрагменты.

Для разбиения строки на фрагменты используются следующие методы класса str.

1. str.split(sep=None, maxsplit=-1) – выполняет разбиение исходной строки на подстроки по пробельным символам (по умолчанию) или подстроке sep, если она задана, и возвращает их в виде списка. Следует добавить, что когда аргумент sep не указывается, интерпретатор считает все подряд идущие пробельные символы одним пробелом, а разбиение пустой строки или строки, состоящей из одних пробельных символов, возвращает пустой список. Однако если разбить пустую строку, использовав отличный

от пробела разделитель, метод вернет список с пустой строкой. Если указан аргумент maxsplit, метод производит только указанное количество разбиений, которые будут выполнены слева направо. По умолчанию строка разбивается по всем имеющимся разделителям.

2. str.rsplit(sep=None, maxsplit=-1) – это то же самое, что и предыдущий метод. Однако в этом случае разбиение выполняется с конца строки, то есть справа налево.
3. str.splitlines([keepends]) – возвращает список строк, выполняя разбиение строки по символам перевода строки и, по умолчанию, удаляя их. Однако если указать необязательный аргумент keepends=True, символы перевода строки будут сохранены. К допустимым символам перевода строки относятся:
    - \n – перенос строки;
    - \r – возврат каретки;
    - \r\n – возврат каретки и перенос строки;
    - \v или \x0b – вертикальный отступ;
    - \f или \x0c – разрыв страницы;
    - \x1c – разделитель файлов;
    - \x1d – разделитель групп;
    - \x1e – разделитель строк;
    - \x85 – следующая строка;
    - \u2028 – разделитель строк;
    - \u2029 – разделитель абзацев.
4. str.partition(sep) – разбивает строку в направлении слева направо по первому найденному разделителю sep и возвращает кортеж из трех элементов: строка до разделителя, сам разделитель и строка после разделителя. Если разделитель найден не будет, метод вернет кортеж так же состоящий из трех элементов: исходная строка str и две пустые строки.
5. str.rpartition(sep) – разбивает строку в направлении справа налево (то есть поиск разделителя ведется с конца строки) по первому с конца разделителю sep и возвращает кортеж из трех элементов: строка до разделителя, сам разделитель и строка после разделителя. Если разделитель найден не будет, метод вернет кортеж так же состоящий из трех элементов: две пустые строки, а также сама исходная строка str.
6. str.join(iterable) – возвращает строку, полученную объединением всех элементов итерируемого объекта iterable (то есть элементы, которые можно пересчитать) через разделитель str, который может быть пустой строкой. Отметим, все элементы передаваемой последовательности должны быть строками, иначе будет вызвана ошибка.

Программный модуль, демонстрирующий удаление пробелов и части строки, приведен в листинге 2.32.

Листинг 2.32. Разбиение строки на фрагменты

```
# Листинг 2.32
str_1 = "Один два три четыре пять"
print("Исходная строка->", str_1)
print('-'*60)

# Разбиение исходной строки на подстроки
```

```
str_2 = str_1.split()
print("Разбивка на подстроки (все слова)->", str_2)
print('-'*60)

# Разбиение исходной строки на подстроки (выделить два слова справа)
str_2 = str_1.rsplit(maxsplit=2)
print("Разбивка на подстроки (выделить два слова справа)->", str_2)
print('-'*60)

# Создать список из нескольких строк
print("Ниже напечатано 3 строки")
str_1 = "Строка1\nСтрока2\nСтрока3"
print(str_1)
str_2 = str_1.splitlines()
print('Подстроки из трех строк ->', str_2)
print('-'*60)

# Создать список по символу разделителю (первый слева)
str_1 = "Строка до разделителя -  после разделителя 1 - после
разделителя 2"
print(str_1)
str_2 = str_1.partition('-')
print('Подстроки по разделителю (слева) ->', str_2)
print('-'*60)

# Создать список по символу разделителю (последний справа)
str_1 = "Строка до разделителя -  после разделителя 1 - после
разделителя 2"
print(str_1)
str_2 = str_1.rpartition('-')
print('Подстроки по разделителю (справа) ->', str_2)
print('-'*60)

# Объединить строки из подстрок
str_1 = ['1934 руб.', '(Одна тысяча ', 'девятьсот ', 'тридцать ',
'четыре руб.)']
print(str_1)
str_2 = ''.join(str_1)
print('Обьединение строки из подстрок ->', str_2)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.32. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.35).

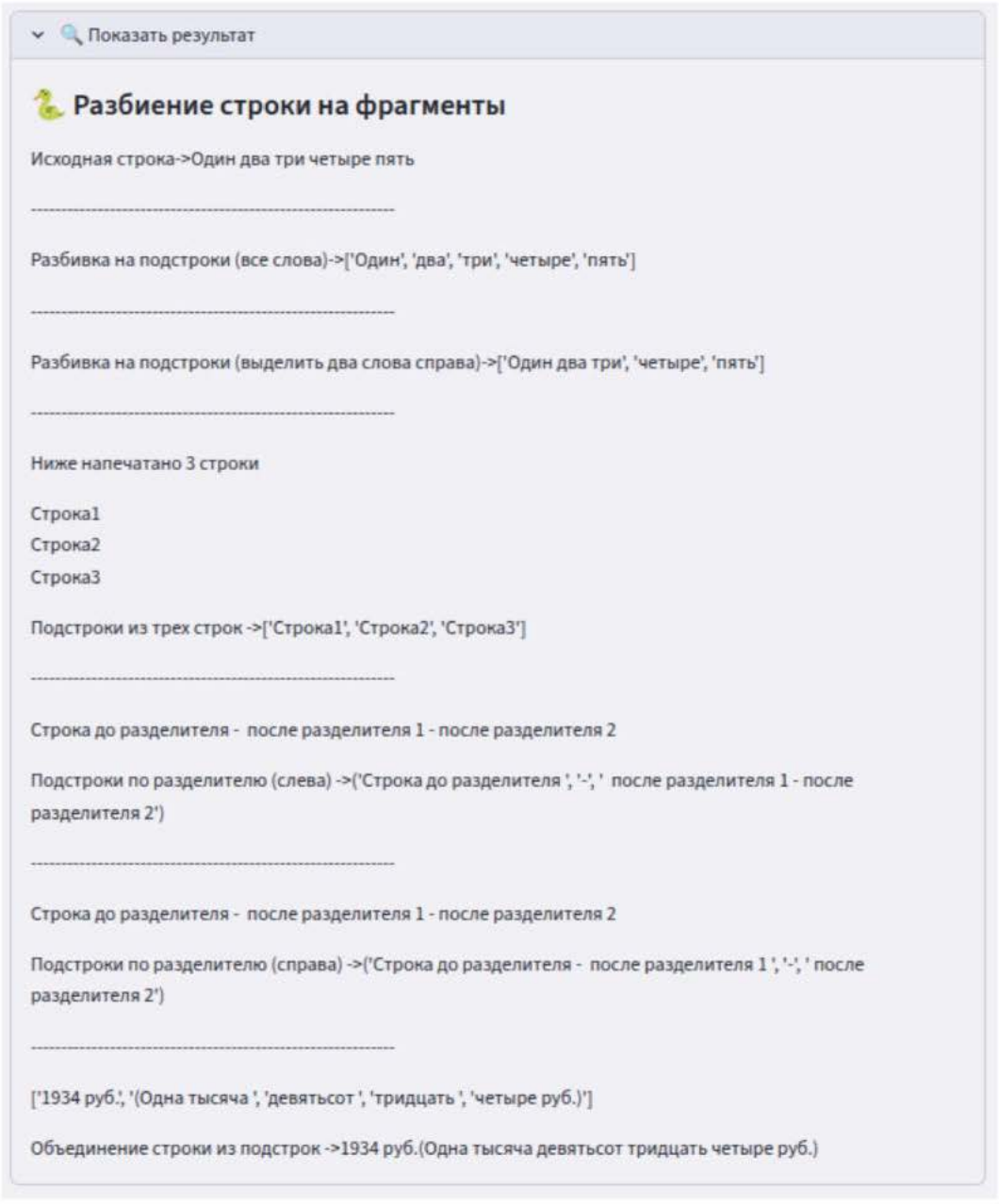


Рис. 2.35. Разбиение строки на фрагменты

2.3.11. Анализ наличия или отсутствия определенных символов в строке.

Для анализа наличия или отсутствия определенных символов в строке используются следующие методы класса str.

1. str.startswith(prefix[, start[, end]]) – возвращает True, если строка str или срез строки str[start:end], если он задан, начинается последовательностью символов prefix. В противном случае метод возвращает False.
2. str.endswith(suffix[, start[, end]]) – возвращает True, если строка str или срез строки str[start:end], если он задан, заканчивается последовательностью символов suffix. В противном случае метод возвращает False.
3. str.isidentifier() – возвращает True, если строка str не пустая и является допустимым идентификатором. В противном случае метод возвращает False.
4. str.isspace() – возвращает True, если строка str не пустая и состоит только из пробельных символов. В противном случае метод возвращает False.
5. str.isascii() – возвращает True, если строка str не пустая и содержит только символы ASCII. В противном случае метод возвращает False.
6. str.isdigit() – возвращает True, если строка str не пустая и содержит только символы ASCII, обозначающие цифры десятичной системы счисления. В противном случае метод возвращает False.
7. str.isalpha() – возвращает True, если строка str не пустая и состоит только из алфавитных символов Юникода. В противном случае метод возвращает False.
8. str.isdecimal() – возвращает True, если строка str не пустая и содержит только символы Юникода, обозначающие цифры десятичной системы счисления. В противном случае метод возвращает False.
9. str.isnumeric() – возвращает True, если строка str не пустая и состоит только из символов Юникода, которые используются для обозначения цифр (не только десятичной системы счисления). В противном случае метод возвращает False.
10. str.isalnum() – возвращает True, если строка str не пустая и состоит только из алфавитных символов и символов Юникода, которые используются для обозначения цифр (не только десятичной системы счисления). В противном случае метод возвращает False.
11. str.isprintable() – возвращает True, если строка str пустая или содержит только печатаемые символы, включая пробел, но не символ перевода строки. В противном случае метод возвращает False.
12. str.istitle() – возвращает True, если строка str не пустая и при этом имеет формат строки заголовка. В противном случае метод возвращает False.
13. str.islower() – возвращает True, если строка str имеет хотя бы один символ, который может быть представлен в нижнем регистре, и все такие символы находятся в нижнем регистре. В противном случае метод возвращает False.
14. str.isupper() – возвращает True, если строка str имеет хотя бы один символ, который может быть представлен в верхнем регистре, и все такие символы находятся в верхнем регистре. В противном случае метод возвращает False.

Программный модуль, демонстрирующий наличие или отсутствие определенных символов в строке, приведен в листинге 2.33.

Листинг 2.33. Анализ наличия или отсутствия определенных символов в строке

```
# Листинг 2.33
# Поиск набора символов с начале слов
print('Поиск набора символов с начала слов')
str_1 = "Сиреневый бульвар"
print('Ищем ("Си") в "Сиреневый бульвар" ->', str_1.startswith("Си"))
print('Ищем ("Ка") в "Сиреневый бульвар" ->', str_1.startswith("Ка"))
print('-'*60)

# Поиск набора символов в конце слов
print('Поиск набора символов в конце слов')
print('Ищем ("вар") в "Сиреневый бульвар" ->', str_1.endswith("вар"))
print('Ищем ("стоп") в "Сиреневый бульвар" ->', str_1.endswith("стоп"))
print('-'*60)

# Проверка правильности идентификатора
print('Проверка правильности идентификатора')
print('Идентификатор "file_25" допустим (да) ->',
'file_25'.isidentifier())
print('Идентификатор "25_file" допустим (нет) ->',
'25_file'.isidentifier())
print('-'*60)

# Проверка - строка состоит только из пробелов
print('Проверка - строка состоит только из пробелов')
print('Исходная строка: "Один пробел" ->', 'Один пробел'.isspace())
print('Исходная строка: "           " ->', '           '.isspace())
print('-'*60)

# Проверка - в строке только символы ASCII
print('Проверка - в строке только символы ASCII')
print('Исходная строка: "chr(88)" ->', 'chr(88)'.isascii())
print('Исходная строка: "Программа" ->', 'Программа'.isascii())
print('-'*60)

# Проверка - в строке только цифры
print('Проверка - в строке только цифры')
print('Исходная строка: "12345" ->', '12345'.isdigit())
print('Исходная строка: "123ab" ->', '123ab'.isdigit())
print('-'*60)

# Проверка - в строке только буквы
print('Проверка - в строке только буквы')
print('Исходная строка: "Слово" ->', 'Слово'.isalpha())
print('Исходная строка: "Слово 1" ->', 'Слово 1'.isalpha())
print('-'*60)
```

```
# Проверка - в строке только десятичные цифры
print('Проверка - в строке только десятичные цифры')
print('Исходная строка: "123"->', '123'.isdecimal())
print('Исходная строка: "0b1101" ->', '0b1101'.isdecimal())
print('-'*60)

# Проверка - в строке только любые цифры
print('Проверка - в строке только любые цифры')
print('Исходная строка: "123"->', '123'.isnumeric())
print('Исходная строка: "XII" ->', 'XII'.isnumeric())
print('Исходная строка: "1.23" ->', '1.23'.isnumeric())
print('-'*60)

# Проверка - в строке только цифры, в которых есть буквы алфавита
print('Проверка - в строке только цифры, в которых есть буквы алфавита')
print('Двоичное число: "0b1101"->', '0b1101'.isalnum())
print('Восьмеричное число: "0o755" ->', '0o755'.isalnum())
print('Шестнадцатеричное число: "0xDD5A" ->', '0xDD5A'.isalnum())
print('Десятичное число: "1.23" ->', '1.23'.isnumeric())
print('-'*60)

# Проверка - в строке только печатаемые символы (без признаков перевода
строки)
print('Проверка - в строке только печатаемые символы (без признаков
перевода строки)')
print('Исходная строка: "Это строка"->', 'Это строка'.isprintable())
print('Исходная строка: "11.22" ->', '11.22'.isprintable())
print('Исходная строка: "Строка1 \\nСтрока1" ->', 'Строка1
\nСтрока1'.isprintable())
print('-'*60)

# Проверка - строка как заголовок
print('Проверка - строка как заголовок')
print('Исходная строка: "Python Is Good."->', 'Python Is
Good.'.istitle())
print('Исходная строка: "python is good." ->', 'python is
good.'.istitle())
print('-'*60)

# Проверка - все символы в нижнем регистре
print('Проверка - все символы в нижнем регистре')
print('Исходная строка: "это строка"->', 'это строка'.islower())
print('Исходная строка: "Это строка" ->', 'Это строка'.islower())
print('-'*60)

# Проверка - все символы в верхнем регистре
print('Проверка - все символы в верхнем регистре')
```

```
print('Исходная строка: "ЭТО СТРОКА"->', 'ЭТО СТРОКА'.isupper())
print('Исходная строка: "Это строка" ->', 'Это строка'.isupper())
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.33. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
Поиск набора символов с начала слов
Ищем ("Си") в "Сиреневый бульвар" -> True
Ищем ("Ка") в "Сиреневый бульвар" -> False
------------------------------------------------------------
Поиск набора символов в конце слов
Ищем ("вар") в "Сиреневый бульвар" -> True
Ищем ("стоп") в "Сиреневый бульвар" -> False
------------------------------------------------------------
Проверка правильности идентификатора
Идентификатор "file_25" допустим (да) -> True
Идентификатор "25_file" допустим (нет) -> False
------------------------------------------------------------
Проверка - строка состоит только из пробелов
Исходная строка: "Один пробел" -> False
Исходная строка: "          " -> True
------------------------------------------------------------
Проверка - в строке только символы ASCII
Исходная строка: "chr(88)" -> True
Исходная строка: "Программа" -> False
------------------------------------------------------------
Проверка - в строке только цифры
Исходная строка: "12345" -> True
Исходная строка: "123ab" -> False
------------------------------------------------------------
Проверка - в строке только буквы
Исходная строка: "Слово" -> True
Исходная строка: "Слово 1" -> False
------------------------------------------------------------
Проверка - в строке только десятичные цифры
Исходная строка: "123"-> True
Исходная строка: "0b1101" -> False
------------------------------------------------------------
Проверка - в строке только любые цифры
Исходная строка: "123"-> True
Исходная строка: "Ⅻ" -> True
Исходная строка: "1.23" -> False
------------------------------------------------------------
Проверка - в строке только цифры, в которых есть буквы алфавита
```

```
Двоичное число: "0b1101"-> True
Восьмеричное число: "0o755" -> True
Шестнадцатеричное число: "0xDD5A" -> True
Десятичное число: "1.23" -> False
------------------------------------------------------------
Проверка - в строке только печатаемые символы (без признаков перевода строки)
Исходная строка: "Это строка"-> True
Исходная строка: "11.22" -> True
Исходная строка: "Строка1 \nСтрока1" -> False
------------------------------------------------------------
Проверка - строка как заголовок
Исходная строка: "Python Is Good."-> True
Исходная строка: "python is good." -> False
------------------------------------------------------------
Проверка - все символы в нижнем регистре
Исходная строка: "это строка"-> True
Исходная строка: "Это строка" -> False
------------------------------------------------------------
Проверка - все символы в верхнем регистре
Исходная строка: "ЭТО СТРОКА"-> True
Исходная строка: "Это строка" -> False
```

2.3.12. Смена кодировки в строке символов.

По умолчанию в Python используется кодировка символов utf-8. Однако при импорте данных из внешних систем текстовая информация может иметь другую кодировку. В этом случае пользователь вместо текста увидит набор непонятных символов. Для смены кодировки символов в символьных строках класс str имеет следующие методы:

- str.encode(encoding='utf-8', 'replace') – возвращает новую строку типа bytes в кодировке utf-8 (по умолчанию) или в кодировке, определяемой аргументом encoding;
- str.decode(encoding='utf-8', 'replace') - выполняет обратное действие – преобразует байты в символьную строку в кодировке, определяемой аргументом encoding.

Программный модуль, демонстрирующий изменение кодировки символов в символьных строках, приведен в листинге 2.34.

Листинг 2.34. Изменение кодировки символов в символьных строках

```
# Листинг 2.34
# Кодируем - декодируем символьную строку в набор байт
a = 'Это исходная строка'
print('Исходная строка ->', a)

# Кодируем в набор байт (кодировка UTF-8)
encoded_bytes = a.encode('utf-8', 'replace')
print('Байтовая строка в кодировке utf-8 ->', encoded_bytes)

# Декодируем в кодировку ASCII
decoded_ascii = encoded_bytes.decode('ascii', 'replace')
```

```
print('Символьная строка после декодирования в ASCII ->', decoded_ascii)

# Декодируем в кодировку utf-8
decoded_utf8 = encoded_bytes.decode('utf-8', 'replace')
print('Символьная строка после декодирования в utf-8 ->', decoded_utf8)
```

В этой программе сначала исходная текстовая строка была преобразована в байтовую строку с кодировкой «UTF-8». Затем мы эту байтовую строку конвертировали в символьную строку с кодировками: «ASCII» и «UTF-8».

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.34. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.36).

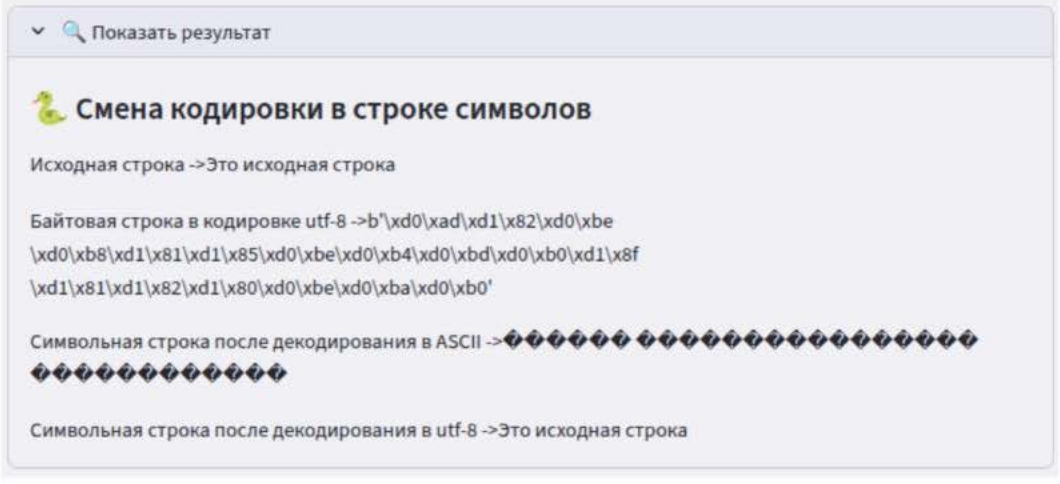


Рис. 2.36. Изменение кодировки символов в символьных строках

Как видно из данного примера, после декодирования байтовой строки в текстовую строку с кодировкой «ASCII» исходный текст превратился в набор нечитаемых символов. А вот после декодирования в строку с кодировкой «UTF-8» исходный текст принял первоначальный вид.

2.3.13. Функции для определения длины строки и положения заданных символов в строке.

Ниже приведен перечень некоторых встроенных функций, которые позволяют определить длину строки и положения заданных символов в строке:

- len(str) – встроенная функция, которая возвращает длину строки (количество символов в строке);

- ord(c) – встроенная функция, которая возвращает порядковый номер символа в Unicode, переданного в качестве аргумента «c»;
- chr(i) – встроенная функция, которая возвращает символ, номер которого в Unicode равен значению аргумента «i».

Программный модуль, демонстрирующий использование этих функций, приведен в листинге 2.35.

Листинг 2.35. Встроенные функции обработки символьных строк

```
# Листинг 2.35
# Исходная строка
str_1 = 'Это строка символов'

# Количество символов в строке
print('Количество символов ->', len(str_1))

# Номер символа "G" в Unicode
print('Номер символа "G" в Unicode ->', ord("G"))

# Символ с номером 71 в Unicode
print('Символ с номером 71 в Unicode ->', chr(71))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.35. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.37).

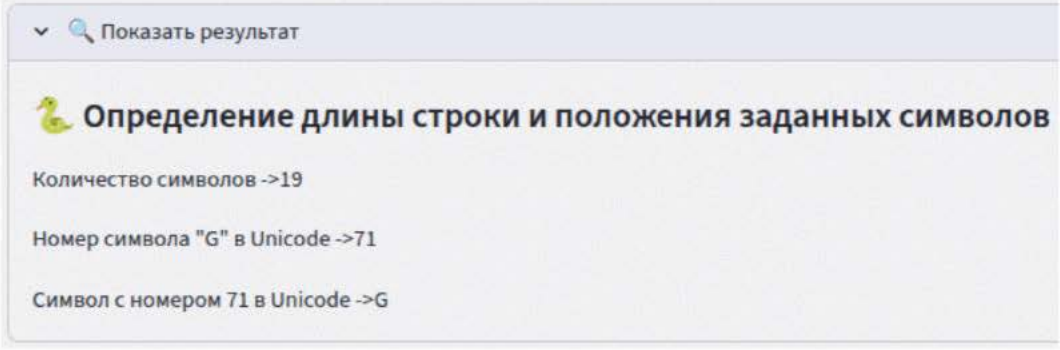


Рис.2.37. Встроенные функции обработки символьных строк

Как видно из приведенных выше примеров, для строк в Python предусмотрен достаточно богатый арсенал методов, при чем здесь были рассмотрены не все методы и функции, а наиболее часто употребляемые.

2.3.14. Метод форматирования строк

В процессе написания программ довольно часто возникают ситуации, когда в строку требуется вставить значения, динамически появляющиеся в процессе выполнения программы. Это могут быть вводимые пользователем данные, чтение данных из файлов, вычисления математических выражений. В Python для таких случаев предусмотрено несколько возможностей:

- использование оператора «%»;
- использование метода str.format() - или метода форматирования строк, который имеет достаточно большое количество аргументов.

Наиболее простой способ форматирование строк - использование оператора «%». Этот оператор позволяет вывести на экран значения из переменных различного типа.

2.3.15. Использование оператора «%» и f-строки для форматирования строк

Использование оператора «%» и f-строки позволяет использовать встроенные выражения Python внутри строковых констант. Простейший пример использования оператора % и f-строки приведен в листинге 2.36.

Листинг 2.36. Использование оператора «%» и f-строки для вывода символьных строк

```
# Листинг 2.36
name = 'Виктор'

# Использование оператора - %
print('Иванов %s' % name)

# Использование f-строки
print(f'Иванов {name}')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.36. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.38).

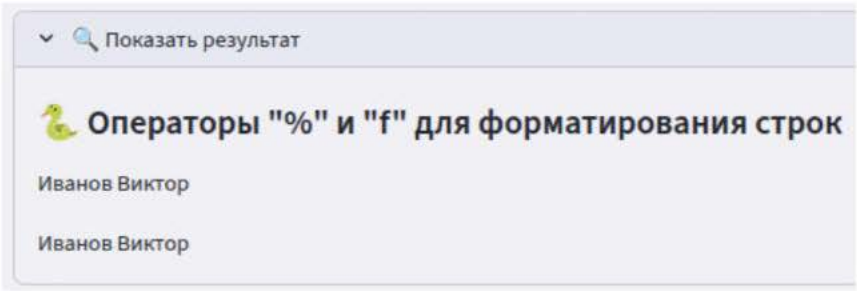


Рис. 2.38. Использование оператора «%» и f-строки для вывода символьных строк

Что здесь произошло? В строковой переменной символы «s%» и идентификатор, который находится в фигурных скобках - {name} были заменены на значение, которое содержалось в переменной name. Почему в паре со знаком «%» был поставлен символ «s»? Потому, что с использованием символа «s» мы указали интерпретатору, что в переменной name содержится текстовая информация. Ниже приведены наборы символов, которые позволяют указать какой тип данных будет выведен на печать:

1. '%d', '%i', '%u' - десятичное целое число;
2. '%o' - число в восьмеричной системе счисления;
3. '%x' - число в шестнадцатеричной системе счисления (буквы в нижнем регистре);
4. '%X' - число в шестнадцатеричной системе счисления (буквы в верхнем регистре);
5. '%e' - число с плавающей точкой с экспонентой (экспонента в нижнем регистре);
6. '%E' - число с плавающей точкой с экспонентой (экспонента в верхнем регистре);
7. '%f', '%F' - число с плавающей точкой (обычный формат);
8. '%g' - число с плавающей точкой. С экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат;
9. '%G' - число с плавающей точкой. С экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат;
10. '%c' - символ (строка из одного символа или число - код символа);
11. '%r' - строка (литерал python);
12. '%s' - строка (обычная текстовая строка).

Пример использования этих символов в операторе «%» приведен в листинге 2.37.

Листинг 2.37. Использование различных символов в операторе «%»

```
# Листинг 2.37
# Десятичное целое число
str_1 = 35
print('Десятичное целое число (35)-> %d' % str_1)

# Восьмеричное число
str_1 = 0o101
print('Восьмеричное число (0o101)-> %o' % str_1)
```

```
# Шестнадцатеричной число
str_1 = 0x101
print('Шестнадцатеричное число (0x101)-> %o' % str_1)

# Число с плавающей точкой с экспонентой
str_1 = 1234.34
print('Число с плавающей точкой с экспонентой (1234.34)-> %e' % str_1)

# Число с плавающей точкой обычный формат
str_1 = 1234.34
print('Число с плавающей точкой обычный формат (1234.34)-> %f' % str_1)

# Строка из одного символа
str_1 = "A"
print('Строка из одного символа ("A")-> %c' % str_1)

# Строка (литерал python)
str_1 = "print"
print('Строка (литерал python) ("print")-> %r' % str_1)

# Строка обычного текста
str_1 = "Строка обычного текста"
print('Строка (Строка обычного текста)-> %s' % str_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.37. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.39).

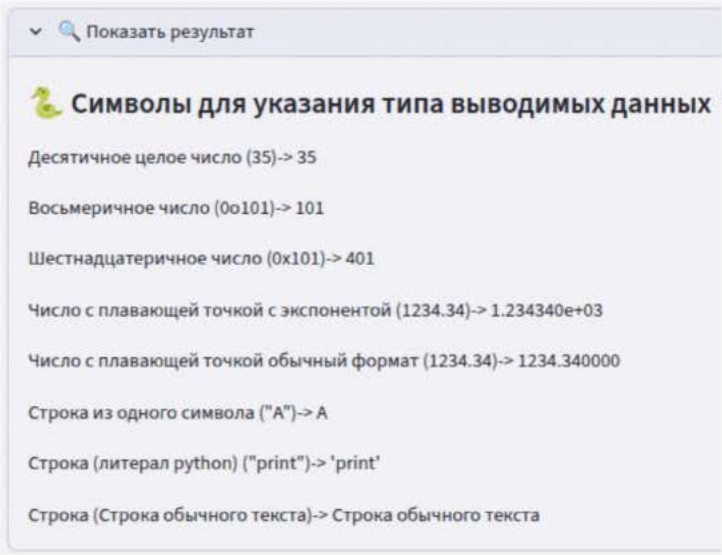


Рис. 2.39. Использование различных символов в операторе «%»

2.3.16. Использование метода str.format() для форматирования строк

Метод str.format() выполняет операцию форматирования строки str. Он возвращает копию строки, в которой каждое «замещающее поле» заменяется строковым значением соответствующего аргумента. Строки str метода str.format() содержат «замещающие поля», которые обозначены фигурными скобками - {}. Весь текст вне фигурных скобок, является обычным текстом, который копируется (выводится на экран) без изменений. Если необходимо включить в текст дополнительные символы скобок, то это можно сделать путем их удвоения, например {{текст}}. Внутри фигурных скобок может быть следующая информация.

1. Использование аналога оператора - %, когда на место фигурных скобок подставляется значение переменной.
2. {} – (пусто), в этом случае говорят о позиционированных аргументах. В этом случае в позиции расположения скобок будут вставлены блоки текста, которые в аргументах следуют один за другим.
3. {0} …{1}… {2} … - в этом случае говорят об индексированных аргументах. В этом случае в позиции расположения скобок будут вставлены блоки текста, которые в аргументах имеют соответствующий индекс (номер по порядку). Индексация аргументов начинается с нуля «0».
4. {str_1} …{ str_2}… { str_3} …- в этом случае говорят об именованных аргументах. В этом случае в позиции расположения скобок будут вставлены блоки текста, которые в аргументах имеют соответствующие имя (идентификатор). В аргументах каждому идентификатору присваивается определенное значение (например, str_1=”Текст строки”).

Пример использования аргументов в методе format() приведен в листинге 2.38.

Листинг 2.38. Использование аргументов в методе format()

```
# Листинг 2.38
# Использование аналога оператора - %
name = 'Виктор'
print('Иванов {}'.format(name))

# Используем позиционные аргументы
str_1 = "Это число-{}, а это число-{}"
print(str_1.format('1234', '4321'))

# Используем в фигурных скобках индексы (номера аргументов).
str_2 = str_1 = "Это слов с индексом-{1}, а это слово с индексом-{0}"
print(str_2.format('Ноль', 'Один'))

# Используем в вызове именованные аргументы
new_str = "Это {str_2} на {str_3} Python"
print(new_str.format(str_2='программа', str_3='языке программирования'))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.38. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.40).

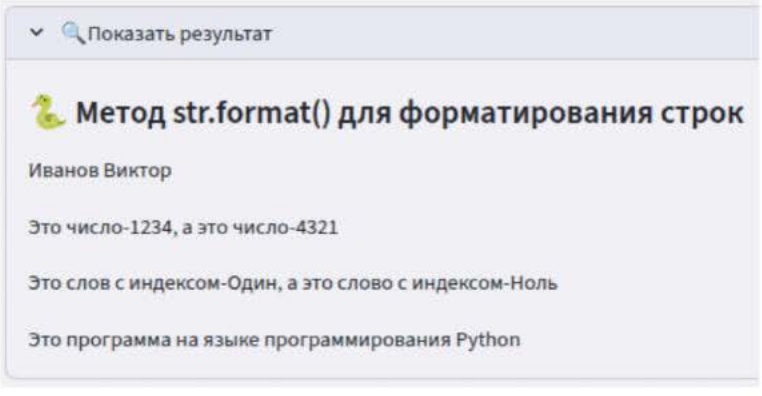


Рис. 2.40. Использование аргументов в методе format()

Как видно из примера, в простейших случаях пустые фигурные скобки или фигурные скобки с индексами замещаются значениями соответствующих позиционных аргументов передаваемых методу. В случае использования в вызове именованных аргументов, вместо фигурных скобок с именами подставляются значения соответствующих именованных аргументов. Однако не стоит забывать, что именованные аргументы должны передаваться после позиционных аргументов.

Следует добавить, что метод может принимать значения любого типа данных. Так что можно смело использовать не только строки, но и, например, числа, списки, словари или другие объекты. При этом после индекса или имени в фигурных скобках форматируемой строки допускается использование синтаксиса квадратных скобок для доступа по ключу к элементу словаря или по индексу к элементу последовательности, а также точки для доступа к атрибутам объекта, если, конечно, соответствующий аргумент метода позволяет это сделать.

Пример использования аргументов в методе format() для списков, словарей приведен в листинге 2.39.

Листинг 2.39. Использование аргументов в методе format() для списков и словарей

```
# Листинг 2.39
# Импортируем модуль math
import math

# Передаем методу список и словарь
print('Печатаем данные из словаря и списка')
str_1 = "{} и {}".format([1, 2, 3, 4], {'Один-': 1, 'Два-': 2, 'Три-':
'Текстовая строка'})
print(str_1)
print('-' * 40)

# Используем доступ по индексу и ключу (без кавычек)
print('Используем доступ по индексу и ключу')
str_2 = "Список - {0[0]}, {0[1]}, {0[2]}, Словарь - {1[one]}, {1[two]}"
str_2 = str_2.format(['А', 'Б', 'В'], {'one': 'Один', 'two': 'Два'})
print(str_2)
print('-' * 40)

# Тоже самое, но для именованных аргументов
print('Используем доступ по именованным аргументам')
str_3 = "Список - {spisok[0]}, {spisok[1]}, Словарь {slov[one]}
{slov[two]}"
str_3 = str_3.format(spisok=['а', 'б'], slov={'one': 'в', 'two': 'г'})
print(str_3)
print('-' * 40)

# Используем именованный аргумент для вывода числа Пи из библиотеки math
print('Вывод числа Пи из библиотеки math')
str_5 = "Число Пи равно {m.pi}".format(m=math)
print(str_5)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.39. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.41).

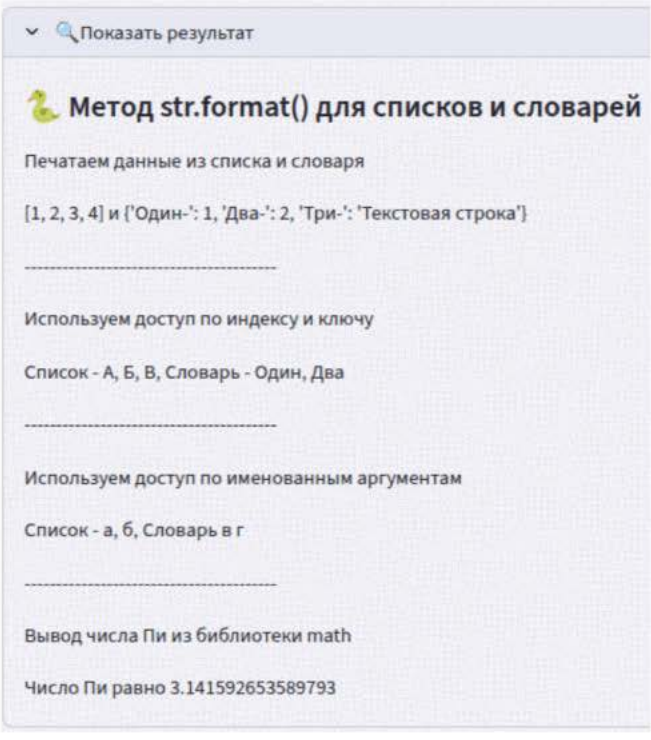


Рис. 2.41. Использование аргументов в методе format() для списков и словарей

Метод format(), помимо вставки в строку значений аргументов (на место фигурных скобок), так же обеспечивает и форматирование самой строки. Такое форматирование выполняется в соответствии с формальным синтаксисом выражения для форматирования:

{[field_name][!conversion_flag][:format_spec]}.

В этом выражении в фигурных скобках указаны не обязательные параметры, которые имеют следующий смысл:

- field_name – необязательный порядковый номер или имя именованного аргумента, за которым может следовать неотрицательный индекс элемента последовательности в виде [index], необязательное имя ключа словаря в виде [key_name], или необязательное имя атрибута объекта в виде [.attribute_name];
- conversion_flag – необязательный флаг конвертации (после восклицательного знака) указывающий, какую из встроенных функций нужно применить к значению в форматированной строке;
- format_spec – необязательная спецификация формата, которая указывается после двоеточия.
- К значению в форматированной строке можно применить следующие функции:
- s – удобное для восприятия представление символов через str() (используется по умолчанию);

- r – внутреннее представление через repr();
- a – форма внутреннего представления через ascii(), когда вместо символов не входящих в набор ASCII выводятся соответствующие экранированные последовательности для символов Юникода.

Пример использования флага conversion_flag приведен в листинге 2.40.

Листинг 2.40. Использование флага conversion_flag

```
# Листинг 2.40
# Импортируем модуль чисел с фиксированной точностью
import decimal

# Создадим переменную заданной точности
d = decimal.Decimal('3.47')

# Преобразуем d в строку
str_1 = "{!s}".format(d)
print('Значение d как строка->', str_1)

# Преобразуем d в во внутренне представление через "r"
str_1 = "{!r}".format(d)
print('Внутренне представление d через "r"', str_1)

# Преобразуем d в во внутренне представление через "a"
str_1 = "{!a}".format(d)
print('Внутренне представление d через "a"', str_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.40. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.42).

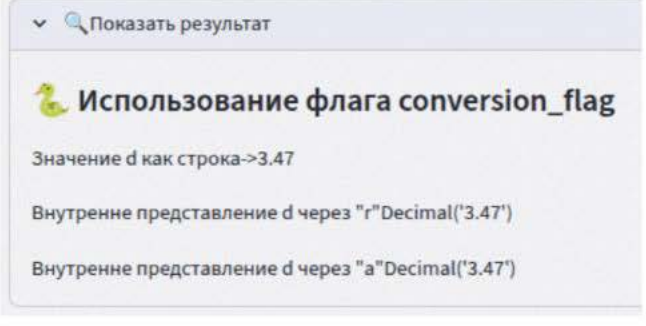


Рис. 2.42. Использование флага conversion_flag

Сразу, после идентификатора, через двоеточие, может следовать «format_specc» помощью которого и выполняется форматирование подставляемого значения. Синтаксис «format_spec» выглядит довольно замысловато:

[[fill] align] [sign] [#] [0] [width] [grouping_option] [.precision] [type]

По сути, это целый мини-язык, который позволяет точно описывать параметры форматирования. Каждый из этих параметров может принимать следующие значения:

- fill — один символ, выступающий в роли заполнителя (по умолчанию – пробел);
- align — символы "<", ">", "=" и "^", позволяющие указать способ выравнивания вставляемого значения;
- sign — символы "+", "-" и " ", позволяющие управлять управляет отображением знака для числовых значений;
- width — число, указывающее ширину поля, в котором будет размещено подставляемое значение;
- grouping_option - задает разделитель разрядов для тысяч (для больших чисел);
- precision — определяет точность числового, подставляемого значения;
- type — определяет тип, подставляемого значения, с помощью символов: "b", "c", "d", "e", "E", "f", "F", "g", "G", "n", "o", "s", "x", "X", "%".

Параметры [#] и [0]. Наличие параметра [#] заставляет интерпретатор использовать альтернативную форму для преобразования чисел. Числа могут быть следующих типов: int, float и complex, причем для каждого типа существует отдельная альтернативная форма представления. Параметр [0] перед параметром width применяется только к числовым значениям и указывает, на то, что заполнение до нужной длины должно выполняться нулями.

Операторы <, ^, > и = используются для выравнивания в том случае, если указана конкретная ширина поля. Они имеют следующий смысл:

- < - выравнивается по левому краю;
- ^ - выравнивается по центру;
  - - выравнивается по правому краю;
- = - сдвигает знаки (+ или -) к левой позиции.

Пример использования символа заполнителя (fill), знаков выравнивания (align) и указания ширины поля (width) приведен в листинге 2.41.

Листинг 2.41. Использование символа заполнителя (fill), знаков выравнивания (align)

```
# Листинг 2.41
# Присваиваем значения переменным
num = 23.55      # число
str_1 = 'Слева'  # символьная строка

print('Выравниваем по левому краю одно значение с заполнителем')
str_2 = "{:*<10}".format(str_1)
print(str_2)
print('-'*40)
```

```
print('Выравниваем по левому краю два значения с заполнителем')
str_2 = "({0:*<10}) ({1:*<10})".format(num, str_1)
print(str_2)
print('-'*40)

print('Выравниваем по правому краю два значения с заполнителем')
str_2 = "({0:*>10}) ({1:*>10})".format(num, str_1)
print(str_2)
print('-'*40)

print('Выравниваем по центру два значения с заполнителем')
str_2 = "({0:*^10}) ({1:*^10})".format(num, str_1)
print(str_2)
print('-'*40)

num1 = -23.55
print('Выравниваем знак "-" у числа')
str_2 = "({0:*=10})".format(num1)
print(str_2)
```

Для наглядности работы программы в этом примере в качестве заполнителя (аргумент fill) использовался знак «*». Значение "=" параметра align применяется только к числовым значениям и добавляет указанное количество пробелов (по умолчанию) или указанных символов слева от числа. Но особенность действия "=" заключается в том, что заполнение происходит между знаком числа и его значением.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.41. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.43).

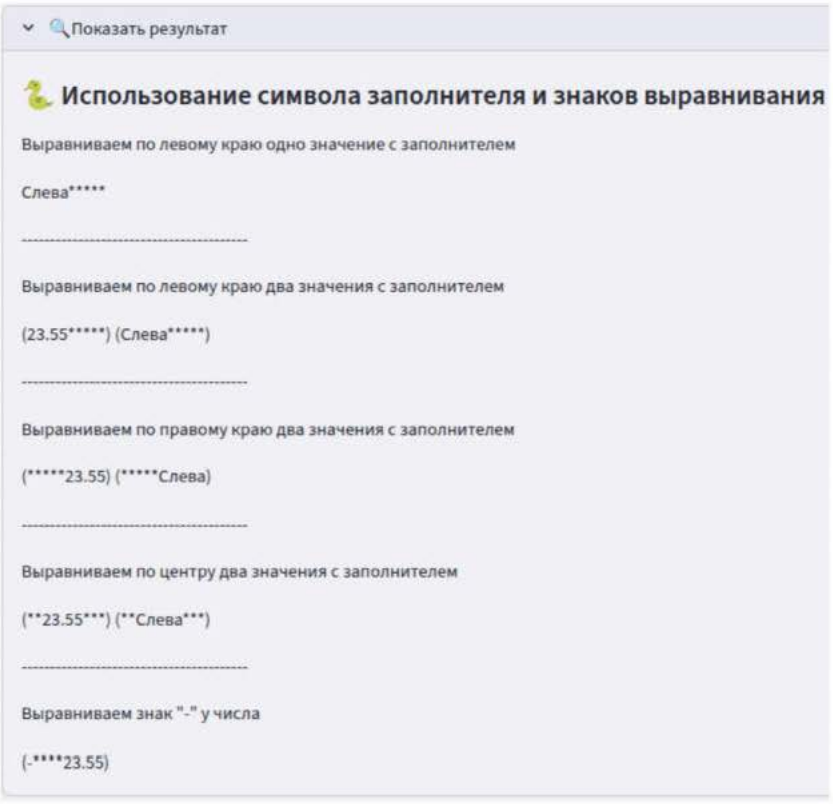


Рис. 2.43. Использование символа заполнителя (fill), знаков выравнивания (align)

Параметр sign – применяется только для чисел, и может принимать следующие значения:

- ·1 знак минуса (-) – отрицательные числа будут отображаться со знаком «-», положительные – вообще без знака (используется по умолчанию);
- ·2 знак плюса (+) – положительные числа будут отображаться со знаком «+», отрицательные – со знаком «-»;
- ·3 символ пробела ( ) – для положительных чисел вместо знака «+» будет отображаться пробел, а для отрицательных «-».

Пример использования параметра sign приведен в листинге 2.42.

Листинг 2.42. Использование параметра sign

```
# Листинг 2.42
print("Вывод знака числа по умолчанию")
str_1 = "({0}, {1})".format(23.5, -3.142)
print(str_1)

print("Обязательный вывод знака числа")
str_2 = "({0:+}, {1:+})".format(23.5, -3.142)
print(str_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.42. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.44).

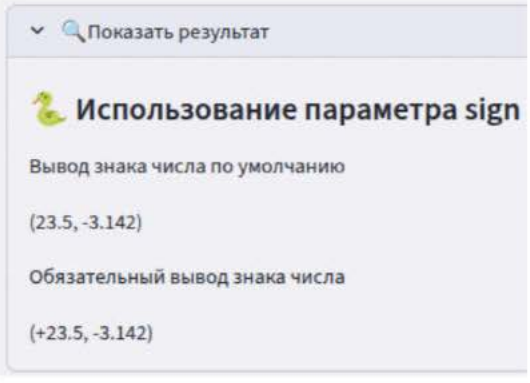


Рис. 2.44. Использование параметра sign

Символ # – используется только в тех случаях, когда требуется форматировать числа. После этого знака можно использовать различные символы. Для целых чисел:

- #b – число будет выведено в двоичной системе счисления;
- #o – число будет выведено в восьмеричной системе счисления;
- #x – число будет выведено в шестнадцатеричной системе счисления.

Для вещественных чисел будет гарантироваться наличие десятичной точки (нужно просто указать решетку без всяких дополнительных символов). При этом, если параметр type установлен в g или G, то конечные нули сохраняются.

Для комплексных чисел также гарантирует наличие десятичной точки, как у действительной, так и у мнимой части числа. При этом, если параметр type установлен в g или G, то конечные нули сохраняются. Возможные значения параметра type будут приведены немного позже.

Пример использования символа # для вывода чисел приведен в листинге 2.43.

Листинг 2.43. Использование символа # для вывода чисел

```
# Листинг 2.43
# Создадим целое число
num = 543

print('Целое число с параметрами форматирования по умолчанию')
print("{0:#}".format(num))

print('Целое число двоичной системе счисления')
```

```
print("{0:#b}".format(num))
print('Целое число шестнадцатеричной системе счисления')
print("{0:#x}".format(num))

num = 543.25
print('Вещественное число с гарантией наличия десятичной точки')
print("{0:#}".format(num))

num = 3.+5.j
print('Комплексное число с гарантией наличия десятичной точки')
print("{0:#}".format(num))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.43. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.45).

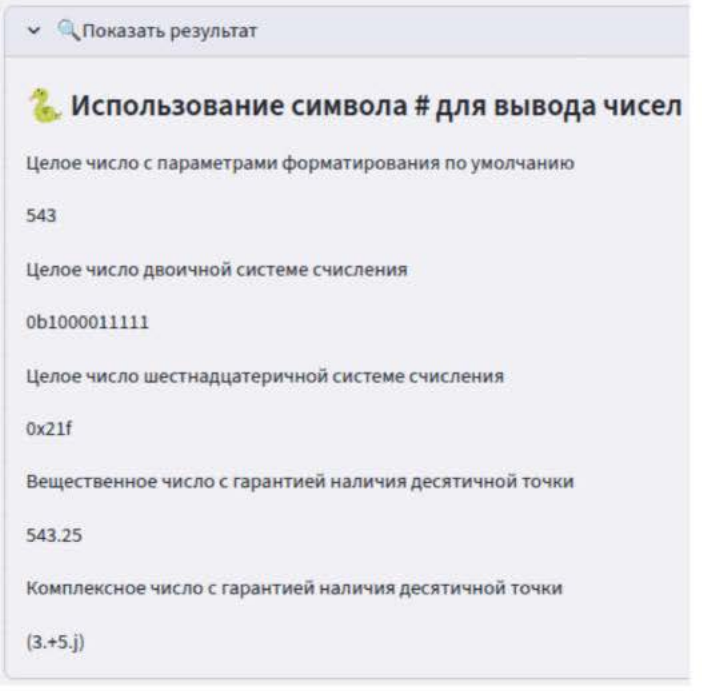


Рис. 2.45. Использование символа # для вывода чисел

Символ 0 – используется только для чисел, и указывает на необходимость заполнения выводимых чисел нулями до ширины, задаваемой параметром width, при этом параметр width – задает минимальную ширину поля для подставляемого значения.

Для лучшей читаемости больших чисел можно использовать параметр grouping_option, он задает разделитель разрядов для тысяч. В качестве разделителя можно использовать следующие символы:

- запятая (,) – задает разделение посредством запятой (при этом для получения разделителей лучше опускать этот параметр, используя параметра type со значением n);
- символ нижнего подчеркивания (_) – задает разделение посредством нижнего подчеркивания в числах с плавающей запятой и целых числах с кодом d (при этом для целых чисел с кодом b, o, x или X разделители будут вставлены через каждые 4 цифры, а для чисел с другими кодами будет вызвано исключение).

Пример использования символов «,» и «_» для вывода чисел приведен в листинге 2.44.

Листинг 2.44. Использование символов «,» и «_» для вывода чисел

```
# Листинг 2.44
# Присваиваем переменной числовое значение
num = 23054303.9641
print('Исходно число->', num)
print('-'*40)

print('Задаем ширину поля в 20 символов с разделителем ","')
print('Заполняем свободное место пробелами')
str_1 = "({0:20,})".format(num)
print(str_1)
print('-'*40)

print('Задаем ширину поля в 20 символов с разделителем "_"')
print('Заполняем свободное место нулями')
str_2 = "({0:020_})".format(num)
print(str_2)
print('-'*40)

print('Задаем ширину поля в 20 символов с разделителем ","')
print('Заполняем свободное место нулями')
str_3 = "({0:0>20,})".format(num)
print(str_3)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.44. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.46).

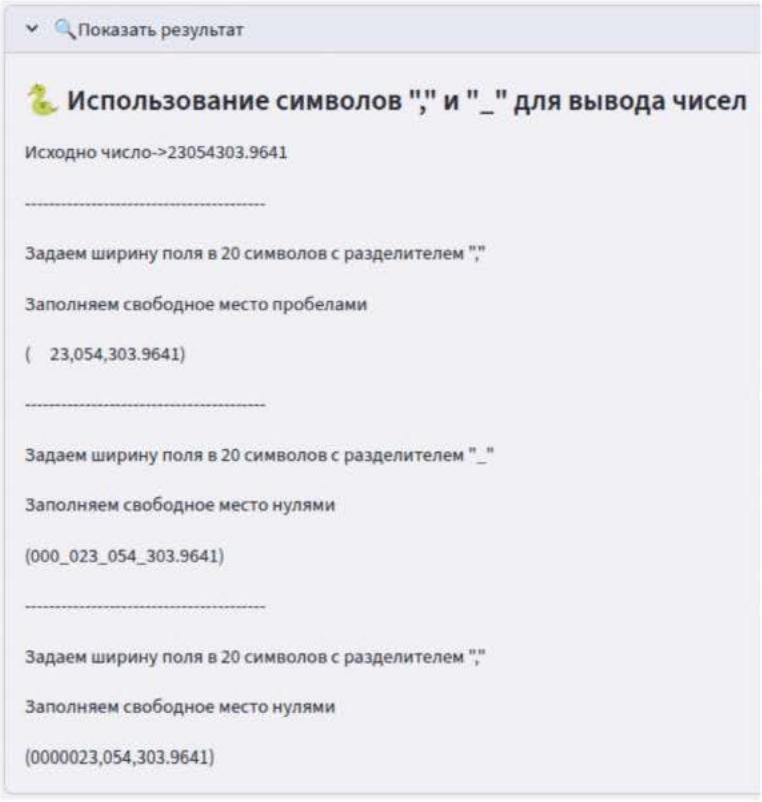


Рис. 2.46. Использование символов «,» и «_» для вывода чисел

Обратите внимание, когда мы использовали в качестве разделителя символ «_», то он разбил на группы, как само число, так и добавленные перед числом нули. Когда же мы использовали в качестве разделителя символ «,», то он разбил на группы, только само число, не трогая добавленные перед числом нули.

Параметр precision – выполняет двойную функцию:

- управляет точностью вещественных чисел;
- задает максимальную ширину поля для строк.

Форматирование вещественных чисел будет зависеть от значения параметра type:

- если type установлен в g или G, то параметр задает общее количество отображаемых цифр числа;
- если type установлен в f или F, то параметр задает количество отображаемых цифр числа после десятичной точки.

Пример использования параметра precision для вывода чисел и строк приведен в листинге 2.45.

Листинг 2.45. Использование параметра precision для вывода чисел и строк

```
# Листинг 2.45
# Присваиваем переменной числовое значение
num = 123.456789
print('Исходно число->', num)
print('-'*40)

# Используем точность по умолчанию
str_1 = '{0:}'.format(num)
print('Точность по умолчанию ->', str_1)
print('-'*40)

# Установим точность в 5 знаков
str_2 = '{0:.5}'.format(num)
print('Точность 5 знаков ->', str_2)
print('-'*40)

# Число отформатированное при помощи флага f
str_3 = '{0:.5f}'.format(num)
print('Флага f5, 5 знаков после точки ->', str_3)
print('-'*40)

# Повысим точность с флагом f
str_4 = '{0:.7f}'.format(num)
print('Флага f7, 7 знаков после точки ->', str_4)
print('-'*40)

# Присваиваем переменной символьное значение
stroka = 'Это строка символов'

# Выводим строку без форматирования
print('Полная строка->', stroka)
print('-'*40)

# Задаем максимальную ширину в 5 символов
str_5 = '{0:.5}'.format(stroka)
print('Строка из 5-ти символов->', str_5)
print('-'*40)

# Увеличим максимальную ширину в 10 символов
str_6 = '{0:.10}'.format(stroka)
print('Строка из 10-ти символов->', str_6)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.45. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.47).

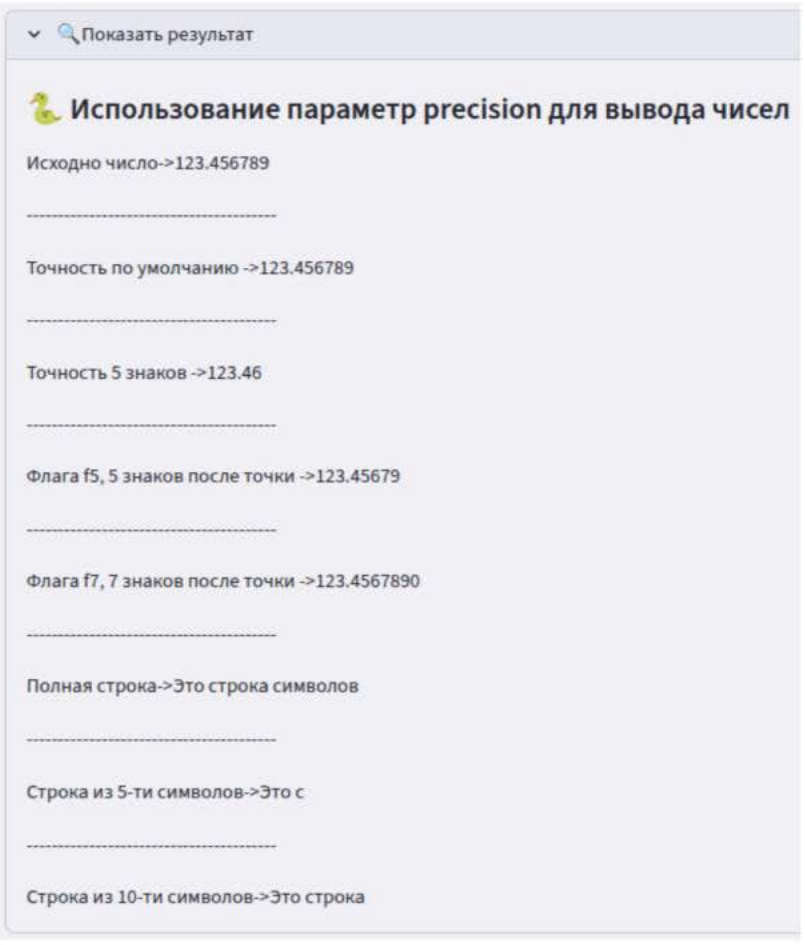


Рис. 2.47. Использование параметра precision для вывода чисел и строк

Теперь рассмотрим возможности форматирования чисел и строк с использованием параметра type, который задает как тип подставляемых данных, так и способ их отображения.

Для строк доступно всего два кода:

- "s" — строковый формат, установлен по умолчанию и может быть опущен;
- код не указан — эквивалентно s.

Для форматирования целых чисел доступны следующие символы:

- b – преобразует целые числа в двоичную систему счисления;
- c – преобразует целые числа в соответствующие символы Юникода в шестнадцатеричном представлении (число сначала переводится в шестнадцатеричном систему счисления, после чего формируется соответствующая экранированная последовательность Юникода вида \uhhhh, которая и представляет целевой символ);

- d – используется по умолчанию для целых чисел и перед выводом преобразует их в десятичную систему счисления;
- – преобразует целые числа в восьмеричную систему счисления;
- x – преобразует целые числа в шестнадцатеричную систему счисления, используя строчные буквы из последовательности «abcdef» для обозначения цифр, превышающих цифру 9;
- X – преобразует целые числа в шестнадцатеричную систему счисления, используя заглавные буквы из последовательности «ABCDEF» для обозначения цифр превышающих цифру 9;
- n – для целых чисел параметр аналогичен значению d, но применяет к числам символ разделителя разрядов тысяч (по умолчанию в качестве разделителя разрядов используется пробел, но в зависимости от настройки locale можно изменить этот символ);
- код не указан — эквивалентно d.

Для форматирования вещественных чисел доступны следующие символы:

- e – преобразует вещественные числа в экспоненциальную форму, в которой показатель степени обозначается строчной буквой e, а точность равна 6 знакам после запятой (спецификатор работает и с целыми числами, предварительно преобразуя их в вещественный тип);
- E – перед выводом преобразует вещественные числа в экспоненциальную форму, в которой показатель степени обозначается заглавной буквой E, а точность равна 6 знакам после запятой (спецификатор работает и с целыми числами, предварительно преобразуя их в вещественный тип);
- f – перед выводом преобразует вещественные числа в стандартную форму с точностью до 6 знаков после запятой;
- F – аналогичен f, однако для обозначения inf и nan используется верхний регистр символов (напомним, что константами inf и nan обозначаются положительная и отрицательная бесконечности);
- g – для вывода используется общий формат, т.е. для небольших чисел параметр действует как f, а для очень больших – как символ e;
- G – для вывода используется общий формат, т.е. для небольших чисел параметр действует как F, а для очень больших – как символ E;
- % – число умножается на 100 с последующим переключением на код f и отображением знака процента в конце.

Пример использования параметра type для форматирования строк, целых и вещественных чисел приведен в листинге 2.46.

Листинг 2.46. Использование параметра type для форматирования строк, целых и вещественных чисел

```
# Листинг 2.46
# Импорт модуля locale
import locale
```

```
# Присваиваем переменной символьное значение
stroka = 'Это строка символов'

print('----- Форматируем строки -----')
print('Форматируем по умолчанию -> {0:}'.format(stroka))
print('Используем символ s -> {0:s}'.format(stroka))
print('_'*40)

# Присваиваем переменной целое число
num = 123

print('----- Форматируем целые числа -----')
print('Форматируем по умолчанию -> {0:}'.format(num))
print('В двоичной системе счисления -> {0:b}'.format(num))
print('В символ Юникода -> {0:c}'.format(num))
print('В десятичной системе счисления -> {0:d}'.format(num))
print('В восьмеричной системе счисления -> {0:o}'.format(num))
print('В шестнадцатеричной системе счисления -> {0:x}'.format(num))
print('В шестнадцатеричной системе счисления -> {0:X}'.format(123))
locale.setlocale(locale.LC_ALL, "")
print('С использованием символа "n" (locale-"") ->
{0:n}'.format(123456))
locale.setlocale(locale.LC_ALL, "us")
print('С использованием символа "n" (locale-"us") ->
{0:n}'.format(123456))
locale.setlocale(locale.LC_ALL, "de")
print('С использованием символа "n" (locale-"de") ->
{0:n}'.format(123456))
locale.setlocale(locale.LC_ALL, "ru")
print('С использованием символа "n" (locale-"ru") ->
{0:n}'.format(123456))
print('_'*40)

# Присваиваем переменной вещественное число
num = 123.456

print('----- Форматируем вещественные числа -----')
print('В экспоненциальной форме (e) -> {0:e}'.format(num))
print('В экспоненциальной форме (E) -> {0:E}'.format(num))
print('В стандартной форме (6 знаков после точки) -> {0:f}'.format(num))
print('В стандартной форме (большое число) -> {0:F}'.format(num**10))
print('Вывод констант (-бесконечность, +бесконечность) -> {0:f},
{1:f}'.format(float('nan'), float('inf')))
print('Вывод констант (-бесконечность, +бесконечность) -> {0:F},
{1:F}'.format(float('nan'), float('inf')))
print('Небольшое число (параметр форматирования-g) ->
{0:g}'.format(num))
```

```
print('Большое число (параметр форматирования-G) ->
{0:G}'.format(num**10))
print('Обозначение процентов (параметр форматирования-%) ->
{0:%}'.format(num))
```

В этой программе был импортирован модуль locale. Он необходим для того, чтобы указать какой символ будет использоваться в качестве разделителя тысяч в тех случаях, когда для форматирования целых чисел будет присутствовать параметр «n». Дело в том, что в разных странах в качестве разделителей тысяч принято использовать разные символы. В данной программе мы использовали четыре типа разделителя: по умолчанию – «» (будет использоваться пробел), для Америки «en_US.utf8» (будет использоваться запятая), для Европы «en_DK.utf8» (будет использоваться точка), для России «ruRU.utf8» (будет использоваться пробел).

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.46. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
----- Форматируем строки -----
Форматируем по умолчанию -> Это строка символов
Используем символ s -> Это строка символов

_____
----- Форматируем целые числа -----
Форматируем по умолчанию -> 123
В двоичной системе счисления -> 1111011
В символ Юникода -> {
В десятичной системе счисления -> 123
В восьмеричной системе счисления -> 173
В шестнадцатеричной системе счисления -> 7b
В шестнадцатеричной системе счисления -> 7B
С использованием символа "n" (locale-"") -> 123 456
С использованием символа "n" (locale-"us") -> 123,456
С использованием символа "n" (locale-"de") -> 123.456
С использованием символа "n" (locale-"ru") -> 123 456

_____
----- Форматируем вещественные числа -----
В экспоненциальной форме (e) -> 1.234560e+02
В экспоненциальной форме (E) -> 1.234560E+02
В стандартной форме (6 знаков после точки) -> 123.456000
В стандартной форме (большое число) -> 822473693827674931200.000000
Вывод констант (-бесконечность, +бесконечность) -> nan, inf
Вывод констант (-бесконечность, +бесконечность) -> NAN, INF
Небольшое число (параметр форматирования-g) -> 123.456
Большое число (параметр форматирования-G) -> 8.22474E+20
Обозначение процентов (параметр форматирования-%) -> 12345.600000%
```

Следует отметить еще одну деталь. При форматировании допускается использование в форматируемой строке одних полей внутри других полей, т.е. можно использовать фигурные скобки внутри других фигурных скобок. При этом разрешается только один уровень вложенности, позволяющий подставлять параметры форматирования в процессе выполнения программы.

Пример использования вложенных полей при форматировании строк, приведен в листинге 2.47.

Листинг 2.47. Использование вложенных полей при форматировании строк

```
# Листинг 2.47
# Отобразим число в экспоненциальной форме.
str_1 = 'Число {0} в экспоненциальной форме: {0:{1}}'.format(45.75, 'e')
print(str_1)

# Отобразим число в обычной форме.
str_2 = 'Число {0} в стандартной форме: {0:{1}}'.format(45.75, 'f')
print(str_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.47. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.48).

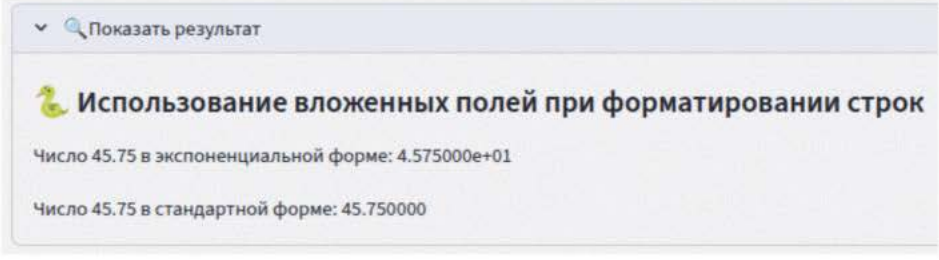


Рис. 2.48. Использование вложенных полей при форматировании строк

Методы форматирования чисел и строк в Python достаточно гибки и имеют много возможностей. Фактически это мини-язык высокого уровня, который позволяет оперировать параметрами форматирования. Конечно, в данном пособии рассмотрены не все возможности форматирования, но, при необходимости, всегда можно обратиться к оригинальной документации по языку программирования Python.

2.3.17. Краткие итоги раздела 2.3

1. Строки используются в Python для хранения и представления текстовой информации и представляют собой неизменяемые упорядоченные последовательности символов Юникода. По умолчанию в скриптах используется кодировка UTF-8.

2. Чтобы создать строку, достаточно заключить последовательность символов в одинарные, двойные или тройные кавычки. При этом допускается использование кавычек одного типа внутри строки, заключенной в кавычки другого типа, либо внутренние кавычки в строке экранировать при помощи обратного слеша «\».
3. Если строка в программном коде имеет большое количество символов, то ее можно разбить на несколько строк. Сделать это можно несколькими способами: использовать обратный слеш «\» в месте разрыва; использовать строки в тройных кавычках, разбить строку на несколько новых строк, а затем объединить их при помощи знака «+» (при этом нужно использовать круглые скобки).
4. Если нужный символ отсутствует на клавиатуре, отобразить его внутри строки программного кода можно при помощи экранированной (управляющей) последовательности. Например, для отображения символа греческого алфавита Ω можно использовать управляющую последовательность \u03A9.
5. Несмотря на то, что строки в Python представляют собой неизменяемые последовательности символов Юникода, доступ к ее отдельным символам можно получить при помощи их индексов. Для этого в квадратных скобках после имени или литерала строки нужно указать порядковый номер символа в строке, не забывая, что нумерация символов в строке начинается с нуля. Более того, доступ к символам строки возможен также и по отрицательным индексам. В этом случае отсчет ведется с конца строки и начинается с -1.
6. Также для строк доступна операция извлечения не только отдельных символов, но и группы символов из заданного диапазона без использования встроенных функций.
7. Для манипулирования строками можно использовать арифметические операции: «+» и «+=» для объединения строк, «*» и «*=» для получения повторяющейся копии строки.
8. Инструкция (оператор) проверки на вхождение in, позволяет совершает поиск требуемой подстроки в целевой строке.
9. Использование методов для объектов класса str позволяет выполнять широкий спектр манипулирования строками. Для вызова метода нужно указать имя объекта, а после точки имя метода.
10. Много возможностей предоставляют методы форматирования строк, которые позволяют, как вставлять данные в различные места строки, таки форматировать саму строку.

2.3.18. Вопросы для самоконтроля к разделу 2.3



1. Какие из представленных литералов относятся к типу str: '3.4', 5.0, 432?
2. Какая кодировка используется в python-скриптах по умолчанию: UTF-8, UTF-16, UTF-32, ASCII?
3. Как правильно использовать кавычки внутри строки? Варианты:
   - print('Что делает функция "print" в Python');
   - print('Что делает функция 'print' в Python');
   - print("Что делает функция "print" в Python");
   - print("Что делает функция 'print' в Python").

4. Какой набор символов относится к экранированной последовательности: \n, /n, //p, &n?
5. Как каким набором символов можно отключить механизм экранирования для строки: //, \\, /o, o/?
6. Как при помощи срезов из слова «лапу», получить слово «упал»:
    6. text = 'лапу'[::-1];
    7. text = 'лапу'[:-1:];
    8. text = 'лапу'[-1::].
7. Укажите метод, который позволяет привести все символы строки в нижний регистр: lower(), upper(), len().
8. Назовите метод строк, который позволяет привести все символы строки в верхний регистр: upper(), lower(), len().
9. Назовите метод строк, который позволяет подсчитать количество символов в строке: len(), lower(), upper().

2.4.Списки в Python

Список (от англ. list) – это изменяемая упорядоченная коллекция объектов произвольного типа. Сами объекты называются элементами списка, а доступ к ним может быть получен при помощи целочисленного индекса или среза.

Списки по своей сути – это аналоги массивов, которые используются в других языках программирования. Почему же разработчики Python вместо термина «массив» стали использовать термин «список». Дело в том, что списки, хоть это и аналог массивов, имеют важно принципиальное отличие. Массивы могут содержать элементы только оного типа, например, только числа или только строки. Списки же в качестве элементов содержат объекты. А в объектах, как мы уже знаем, могут содержаться и числа, и строки, и даже другие объекты. Поэтому списки являются более универсальным средством обработки данных, чем массивы. В один список можно вкладывать другие списки, таким образом, на основе списков можно формировать одномерные, двумерные, трехмерные массивы, и вообще массивы данных любой вложенности.

Как и строки, списки в языке Python относятся к упорядоченным последовательностям. Однако в отличие от строк, списки являются изменяемыми объектами и могут содержать значения любых типов: уже известные нам числа и строки, сами списки, а также объекты любого другого типа данных. При этом изменяться списки могут непосредственно без всяких дополнительных манипуляций и фокусов. Сделать это можно как при помощи операций присваивания по индексам и срезам, так и при помощи методов или специальной инструкции удаления del. По сути, списки являются самым гибким типом упорядоченных коллекций в Python, реализуя практически все необходимое для создания и обработки структур данных любой степени сложности.

2.4.1. Создание списков в Python

Списки в Python создаются при помощи квадратных скобок, внутри которых находятся элементы списка, перечисляемые через запятую. В случае необходимости последний элемент списка можно также завершать запятой. Это никак не повлияет на количество элементов списка, а значит и на его длину.

Пример программы формирования списков приведен в листинге 2.48.

Листинг 2.48. Пример программы формирования списков

```
# Листинг 2.48
# Пустой список
list_1 = []
print('Пустой список ->', list_1)

# Элементы списка
list_2 = [15, "Иванов", None, 33.5, True]
print('Элементы списка ->', list_2)

# Элементы списка с запятой в конце
list_3 = [1, 2, 3, ]
```

```
print('Элементы списка с запятой в конце ->', list_3)

# Списки внутри списка
list_4 = [[1, 2, 3], ['A', 'B', 'C'], [7, 8, 9]]
print('Списки внутри списка ->', list_4)

# В списке разные объекты
list_5 = [{'1': 'start', '2': ['stop', [4, 5]]}, 0.3]
print('В списке разные объекты ->', list_5)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.48. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.49).



Рис. 2.49. Пример программы формирования списков

Как видно из данного примера, внутри списков разрешается использовать объекты любого типа, создавая при этом любые их комбинации и уровни вложенности. А благодаря наличию в синтаксисе квадратных скобок, сложные и объемные литералы списков можно без проблем записывать на нескольких строках, придавая исходному коду удобочитаемый вид.

Посмотрим, как список list_2 будет размещен в памяти компьютера (рис. 2.50).

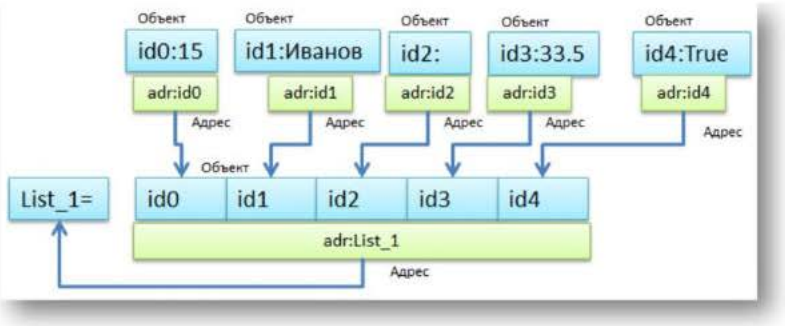


Рис.2. 50. Размещение списка list_1 в памяти компьютера

Как видно из данного рисунка, после инструкции присваивания по адресу, который мы условно назвали adr:List_1, был создан объект. Но в этом объекте не хранятся сами данные, там хранится пять ссылок на другие объекты с условными именами (или индексами): id0, id1, id2, id3, id4. Эти объекты расположились в других ячейках памяти с условными адресами: adr:id0, adr:id1, adr:id2, adr:id3, adr:id4. И уже в этих ячейках записаны значения данных разных типов. Таким образом, уникальность списков заключается в том, что через единственный идентификатор можно получить доступ к совершенно разным типам данных, просто указав значения индекса нужного элемента.

Если теперь тому же списку присвоить новый набор данных, например list_1 = [0.03,'Петров'], то будет создан совершенно новый объект по другому адресу, а старые данные Python просто почистит (рис. 2.51).

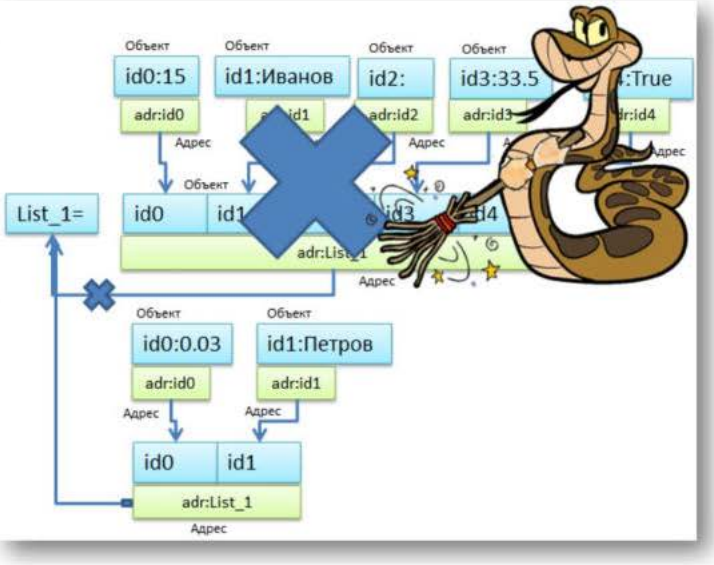


Рис. 2.51. Присвоение списку list_1 новых значений

Как видно из данного рисунка, переменная list_1 просто получила ссылку на новый объект и по новому адресу, а старые объекты Python просто удалит и освободит от них память.

Список можно создать и при помощи встроенного конструктора типа - list(). Если использовать его без аргумента, то он вернет нам пустой список. Если же передать ему итерируемый объект, например, строку или кортеж, то конструктор вернет список, элементами которого будут соответствующие элементы переданного объекта.

Пример программы формирования списков конструктором list() приведен в листинге 2.49.

Листинг 2.49. Пример программы формирования списков конструктором list()

```
# Листинг 2.49
# Пустой список
list_1 = list()
print('Пустой список ->', list_1)

# Список из строки
list_2 = list('123')
print('Список из строки ->', list_2)

# Список из кортежа
list_3 = list((15, "Иванов", None, 34.5, True))
print('Список из кортежа ->', list_3)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.49. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.52).

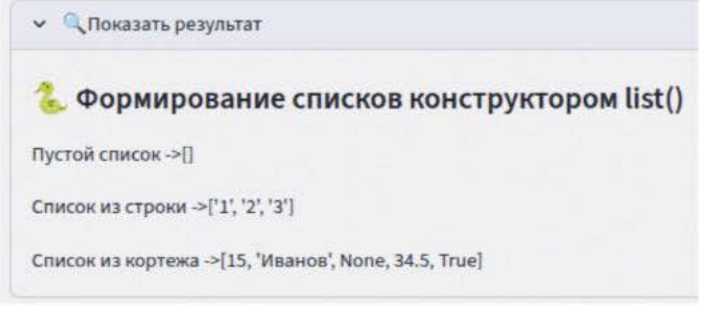


Рис. 2.52. Пример программы формирования списков конструктором list()

Так же в Python для создания списков могут использоваться специальные генераторы списков, которые применяют выражение к каждому элементу передаваемого итерируемого объекта, и на основе их вычисления возвращают новый список.

Пример программы формирования списков из обработанных строк и чисел приведен в листинге 2.50.

Листинг 2.50. Пример программы формирования списков из обработанных строк и чисел

```
# Листинг 2.50
# Список из обработки строк
list_1 = [len('Иванов'), len('Самохвалов'), len('Третьяк')]
print('Список из обработки строк ->', list_1)

# Список из обработки чисел
list_2 = [(3 / 2 + 3)**2,  int(3.1 + 8.5)*4]
print('Список из обработки чисел ->', list_2)

# Формирование списка из числел оператором цикла
num = [i for i in range(1, 10)]
print('Список из цикла обработки чисел ->', num)

# Формирование списка из строки оператором цикла
list_3 = [i for i in "123456789"]
print('Список из цикла обработки строки ->', list_3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.50. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.53).

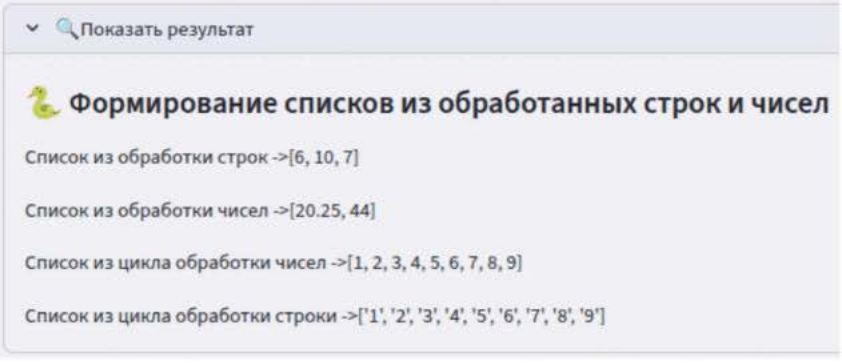


Рис. 2.53. Пример программы формирования списков из обработанных строк и чисел

Для генерации списка можно в квадратных скобках написать выражение, которое будет применяться к каждому элементу списка. А можно использовать операторы организации циклов. В данном примере использовались однострочные операторы организации цикла:

- i for i in range(1, 10) - для обработки чисел;
- i for i in "123456789" - для обработки строки.

Здесь для формирования цикла использовался конструкция «for», в которой:

- i – переменная, которая будет изменяться в цикле;
- for – оператор цикла;
- in range – указатель диапазона изменения переменной «i»
- (1,10) – диапазон изменения переменной «i».

Обратите внимание, что при обработке строки отсутствует, как указатель диапазона «range», так и параметры диапазона. Дело в том, что для строковых данных диапазон изменения переменной «i» будет определен автоматически, исходя из количества символов в строке. Более подробно способы организации циклов в Python будут рассмотрены в следующих разделах пособия.

2.4.2. Доступ к элементам списков по индексу и через срезы

Доступ к элементам списка реализован так же, как и для строк: через индексы элементов или через срезы.

Пример программы выборки элементов списков по индексу и по срезу приведен в листинге 2.51.

Листинг 2.51. Пример программы выборки элементов списков по индексу и по срезу

```
# Листинг 2.51
# Список из целых чисел
list_1 = [1, 2, 3, 4, 5]

# Исходный список
print('Исходный список ->', list_1)

# Первый элемент списка (нумерация начинается с нуля)
print('Первый элемент списка ->', list_1[0])

# Последний элемент списка (количество элементов списка минус один)
print('Последний элемент списка ->', list_1[len(list_1)-1])

# Последний элемент списка (по отрицательному индексу -1)
print('Последний элемент списка по индексу (-1) ->', list_1[-1])

# Элементы списка из среза (индексы 2-4)
print('Элементы списка из среза (индексы 2-4) ->', list_1[2:4])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.51. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.54).



Рис. 2.54. Пример программы выборки элементов списков по индексу и по срезу

В результате обращения к элементу списка по его индексу возвращается некоторый объект, расположенный в указанном месте. В качестве такого объекта может быть другой список, который имеет свой набор индексов. В этом случае для получения доступа к элементам на более глубоком уровне вложенности необходимо указать цепочку из нескольких индексов.

Пример программы выборки элементов по цепочке индексов приведен в листинге 2.52.

Листинг 2.52. Пример программы выборки элементов по цепочке индексов

```
# Листинг 2.52
# Простой вложенных списков
list_1 = [3.14, ['Пес', [4, 5]], 'Кот']

# Исходный список
print('Исходный список ->', list_1)

# Выводим число 3.14 (1-й элемент списка).
print('Число 3.14 ->', list_1[0])

# Выводим 3-й символ строки 'Пес'
print('3-й символ строки "Пес" ->', list_1[1][0][2])

# Выводим всю строку 'Пес'
print('Всю строку "Пес" ->', list_1[1][0][0:3])

# Выводим 2-й символ строки 'Кот'
print('2-й символ строки "Кот" ->', list_1[2][1])

# Выводим число 5 вложенного списка [4, 5]
print('Число 5 вложенного списка [4, 5] ->', list_1[1][1][1])

# Выводим число 4 вложенного списка со словарем
```

```
list_1 = [{'one': 'top', 'two': ['abc', [4, 5]]}, 0.3]
print('Число 4 вложенного списка со словарем ->',
list_1[0]['two'][1][0])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.52. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.55).

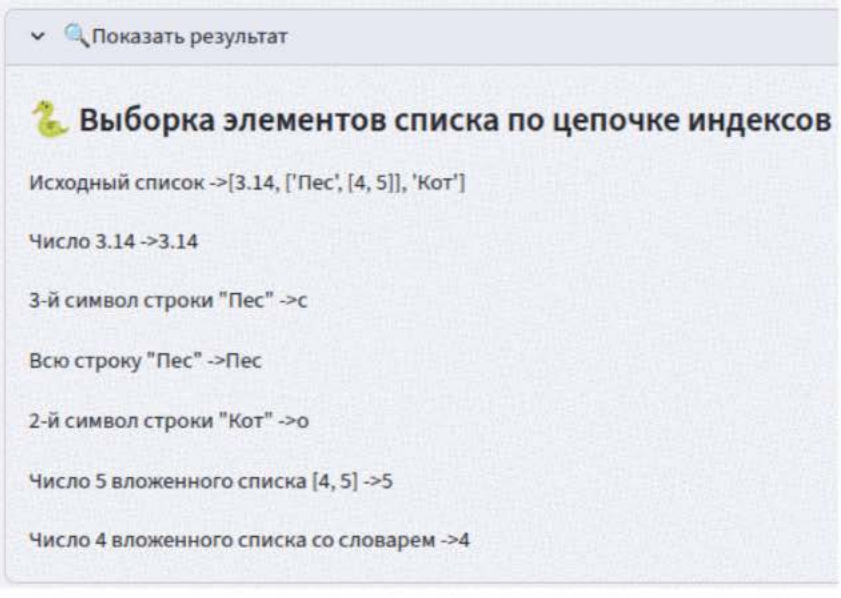


Рис. 2.55. Пример программы выборки элементов по цепочке индексов

Здесь цепочка индексов работает следующим образом.

list_1[0] – для внешнего списка указан индекс [0], на этом месте стоит число – 3.14, которое и выведено на экран.

list_1[1][0][2] – для внешнего списка указан индекс [1], на этом месте находится вложенный список из двух элементов - ['Пес', [4, 5]]. Для этого вложенного списка указан индекс [0], на этом месте находится строка - 'Пес'. Следующий индекс - [2] относится к этой строке, на этом месте находится символ «с», который и выведен на экран.

list_1[1][0][0:3] - для внешнего списка указан индекс [1], на этом месте находится вложенный список из двух элементов - ['Пес', [4, 5]]. Для этого вложенного списка указан индекс [0], на этом месте находится строка - 'Пес'. На следующей позиции находится срез - [0:3] относится к этой строке, на этом месте находится цепочка символов «Пес», которая и выведена на экран.

list_1[2][1] - для внешнего списка указан индекс [2], на этом месте находится строка ‘Кот’. Для этой строки указан индекс [1], на этом месте находится символ - 'о', который и выведен на экран.

list_1[1][1][1]- для внешнего списка указан индекс [1], на этом месте находится другой вложенный список элементов - ['Пес', [4, 5]]. Для этого вложенного списка указан индекс [1], на этом месте находится еще один вложенный список - [4, 5]. Для этого вложенного списка указан индекс [1], на этом месте находится цифра – 5, которая и выведена на экран.

Следует отметить, что цепочка доступа к конкретному элементу может иметь и более сложную конструкцию, включающую не только квадратные скобки с индексами для доступа к элементам списков, но и квадратные скобки с ключами для доступа к элементам словарей, и точки для доступа к атрибутам объектов. Например, для списка:

```
list_1 = [{'one': 'top', 'two': ['abc', [4, 5]]}, 0.3]
```

цепочка получения доступа к первому элементу вложенного списка [4, 5] будет иметь вид list_1[0]['two'][1][0]. В этом случае на экран будет выдано число 4.

Что касается получения срезов для списков, то в результате операции всегда возвращается новый список, состоящий из указанных в срезе элементов исходного списка. Сам же исходный список не изменяется до тех пор, пока мы не присвоим срезу новое значение.

При работе со срезами указывается три индекса - list_1[start:stop:step], каждый из которых имеет следующий смысл:

- start – индекс первого элемента списка;
- stop – индекс последнего элемента списка;
- step – шаг выборки элементов списка.

При использовании срезов нужно учитывать следующие особенности.

1. Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.
2. Отрицательное значение шага - выборка элементов ведется в обратном порядке справа налево.
3. Если не указан старт [:stop:step] - начинаем с самого края коллекции. Если шаг положительный, то с первого элемента (включая его), если шаг отрицательный, то перебор идет от конца к началу.
4. Если не указан стоп [start:: step] - идем до самого края коллекции. При этом, если шаг положительный с первого элемента (включая его), если шаг отрицательный, то перебор идет от конца к начал.
5. step = 1, то есть последовательный перебор слева направо (указывать не обязательно) - это значение шага по умолчанию. В таком случае достаточно указать [start:stop].
6. Можно не указывать значения индексов - [:], это значит взять коллекцию целиком.

Примечание.
ВАЖНО: При срезе, первый индекс входит в выборку, а второй нет!

Пример программы получения среза элементов списка по индексам приведен в листинге 2.53.

Листинг 2.53. Пример программы получения среза элементов списка по индексам

```
# Листинг 2.53
# Формируем список из целых чисел
list_1 = [1, 2, 3, 4, 5, 6, 7]

# Исходный список
print('Исходный список ->', list_1)

# Полный список
print('Полный список [:] ->', list_1[:])

# Получаем список из первых двух элементов
print('Первые два элемента [0:2] ->', list_1[0:2])
print('Первые два элемента [:2] ->', list_1[:2])

# Получаем список из последних двух элементов ([6, 7])
print('Последние два элемента [-2:] ->', list_1[-2:])

# Получаем список [3, 4, 5] из элементов с 3-го по 5-й
print('Элементы с 3-го по 5-й [2:5] ->', list_1[2:5])

# Получим [2, 3, 4, 5] (использование отрицательных индексов)
print('Элементы с 2-го по 5-й по отрицательным индексам [-6:-2] ->',
list_1[-6:-2])
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.53. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.56).

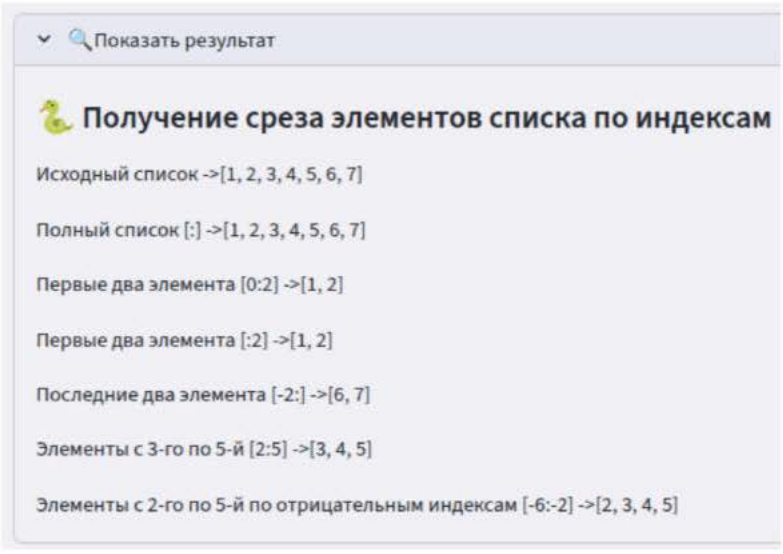


Рис. 2.56. Пример программы получения среза элементов списка по индексам

При извлечении срезов следует помнить, что нумерация элементов в списках начинается с нуля, а элемент с индексом второго предела, указанного в квадратных скобках, не будет присутствовать в срезе. Например, в срезе списка my_list[0:7] будут присутствовать первые семь элементов с индексами от 0 до 6, но будет отсутствовать элемент с индексом 7.

Как и в случае со строками, один или оба предела среза можно опускать. Если не указывать первый предел, интерпретатор будет начинать срез с первого элемента списка (с индексом 0), а при отсутствии второго предела срез будет идти до конца списка. Если опустить оба предела, мы получим копию списка.

При необходимости можно указать и третий предел, который будет задавать шаг среза. Пример программы получения среза элементов списка по индексам, используя третий индекса в качестве шага, приведен в листинге 2.54.

Листинг 2.54. Пример программы получения среза элементов списка по индексам

```
# Листинг 2.54
# Формируем список из целых чисел.
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Исходный список
print('Исходный список list_1 ->', list_1)

# Выводим каждый 2-й элемент списка (слева направо)
print('Каждый 2-й элемент среза (слева направо) [0:7:2] ->',
list_1[0:7:2])
print('Каждый 2-й элемент среза (слева направо) [:7:2] ->',
list_1[:7:2])

# Выводим каждый 2-й элемент списка, начитая с конца
```

```
print('Каждый 2-й элемент списка, начитая с конца [::-2] ->', list_1[::-
2])

# Выводим каждый третий элемент всего списка
print('Каждый 3-й элемент среза [::3] ->', list_1[::3])

# Выводим каждый 2-й элемент списка, считая с конца
print('Каждый 2-й элемент среза, считая с конца [8:2:-2] ->',
list_1[8:2:-2])

# Выводим каждый 3-й элемент списка, считая с конца
print('Каждый 3-й элемент среза, считая с конца [-3:1:-3] ->', list_1[-
3:1:-3])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.54. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.57).



Рис. 2.57. Пример программы получения среза элементов списка по индексам с шагом

Как видим, шаг среза также может иметь отрицательное значение. В этом случае отсчет элементов списка и шагов ведется в противоположном направлении, а границы среза указываются справа налево, то есть индекс элемента, стоящего правее в списке должен указываться первым. Например, в срезе примера my_list[-3:1:-3] отсчет ведется справа налево, начиная с 3-го элемента, отсчитываемого с конца списка, до элемента с индексом 1, который находится левее и в срез не включается. Для приведенного выше списка в этом случае был получен срез из элементов [8, 5].

Изменяемость списков означает, что мы можем не только получать доступ к элементам списка, но и изменять их значения, то есть удалять или добавлять в список новые элементы. При этом изменению подвергается не копия списка, а непосредственно сам список. Для изменения значения отдельного элемента списка нужно найти элемент в списке по индексу и присвоить ему новое значение.

Пример программы изменения элементов списка по индексам приведен в листинге 2.55.

Листинг 2.55. Пример программы изменения элементов списка по индексам

```
# Листинг 2.55
# Формируем список из 3-х элементов.
list_1 = ['и', ['b', [4, 5]], 0.3]

# Исходный список
print('Исходный список list_1 ->', list_1)

# Изменяем третий элемент списка [2] (0.3 на 555)
list_1[2] = 555
print('Изменили элемент с индексом [2] (0.3 на 555) ->', list_1)

# Изменяем 1-й элемент вложенного списка [4, 5] (4 на stop)
list_1[1][1][0] = 'stop'
# Выведет ['и', ['b', ['s', 5]], 4]
print("Изменили элемент с индексом [1][1][0] (4 на stop) -> ", list_1)

# Используем отрицательный индекс.
list_1[-3] = 'Итог'
print('Изменили элемент с индексом [-3] (и на Итог) ->', list_1)

# Заменим 1-й элемент на список ['Один', 'Два', 'Три']
list_1[0] = ["Один", "Два"]
print('Изменили элемент с индексом [0] (Итог на ["Один", "Два"]) ->',
list_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.55. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.58).

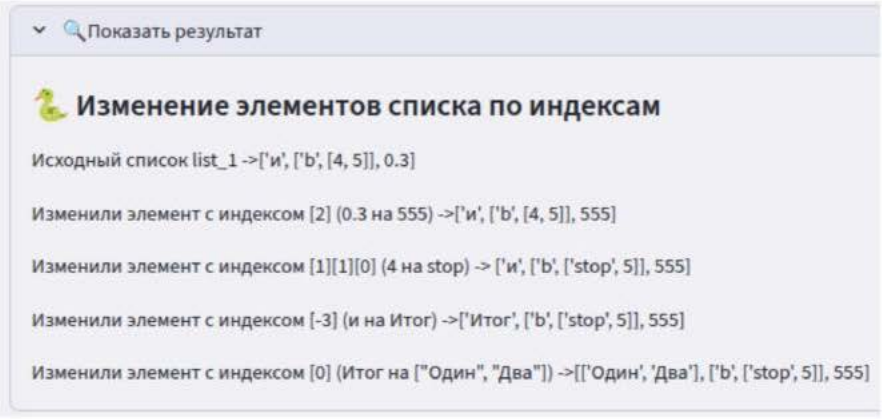


Рис. 2.58. Пример программы изменения элементов списка по индексам

В более сложных случаях, когда нужно удалить или добавить элементы, а также изменить значения сразу нескольких элементов, следует использовать срезы.

Пример программы изменения (добавления, удаления) элементов списка по срезам приведен в листинге 2.56.

Листинг 2.56. Пример программы изменения (добавления, удаления) элементов списка по срезам

```
# Листинг 2.56
# Формируем список
list_1 = [1, 2, 3, 4, 5, 6, 7]

# Исходный список
print('Исходный список list_1 ->', list_1)

# Меняем первый элемент списка на "один"
list_1[0:1] = ["Один"]
print('Меняем первый элемент списка [0:1] на "Один" ->', list_1)

# Меняем первые 2 элемента списка на "Раз-два" (сократили список)
list_1[0:2] = ["Раз-два"]
print('Меняем первые 2 элемента списка [0:2] на "Раз-два" ->', list_1)

# Меняем первый элемент списка тремя элементами (расширили список)
list_1[0:1] = [111, 222, 333]
print('Меняем первый элемент списка [0:1] на [111, 222, 333] ->',
list_1)

# Замена элементов среза на элементы этого же среза
list_1[0:2] = list_1[2:]
print('Замена элементов среза на элементы этого же среза [0:4]=[2:] ->',
list_1)
```

```
# Удаляем последние два элемента списка
list_1[-2:] = []
print('Удаляем последние два элемента списка [-2:] ->', list_1)

# Меняем элементы [0:4] на символы "A", "B", "C"
list_1[0:4] = 'ABC'
print('Меняем элементы [0:4] на символы "A", "B", "C" ->', list_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.56. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.59).

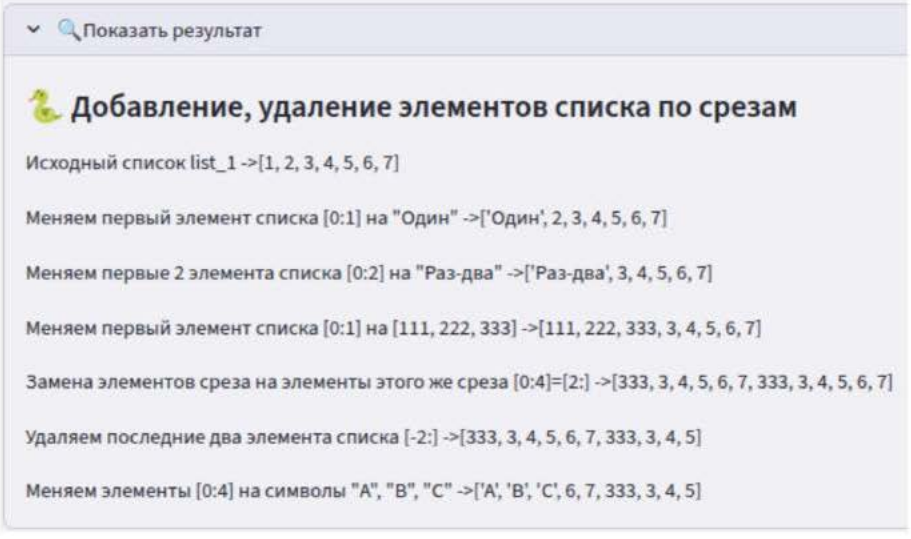


Рис. 2.59. Пример программы изменения (добавления, удаления) элементов списка по срезам

Обратите внимание, что для удаления конкретного элемента списка нужно взять срез, включающий только этот элемент, и присвоить ему пустой список. Если же использовать просто индекс, элемент останется, а пустой список станет его значением.

При желании в срезах можно использовать и третий параметр - шаг среза. Но здесь нужно быть осторожным, т.к. количество элементов такого среза должно соответствовать количеству элементов в присваиваемом объекте.

Пример программы изменения элементов списка с использованием третьего индекса в качестве «шага» приведен в листинге 2.57.

Листинг 2.57. Пример программы изменения элементов списка с использованием третьего индекса

```
# Листинг 2.57
# Формируем список.
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Изменим каждый второй элемент списка
list_1[0::2] = 'abcde'
print('Изменим каждый второй элемент списка [0::2] ->', list_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.57. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.60).

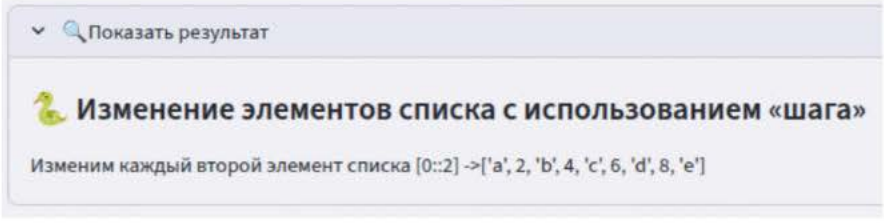


Рис. 2.60. Изменение элементов списка с использованием третьего индекса

Как показано в предыдущих примерах, операции взятия среза у списков дают широкие возможности. Для обработки списков могут быть задействованы и некоторые методы класса list, которые мы рассмотрим в следующих разделах.

2.4.3. Операторы и инструкции для обработки списков

Поскольку списки относятся к упорядоченным последовательностям, то для них можно использовать арифметические операторы «+», «+=», «*» и «*=».

Пример программы применения арифметических операций для обработки списков приведен в листинге 2.58.

Листинг 2.58. Применение арифметических операций для обработки списков

```
# Листинг 2.58
# Формируем списки
list_1 = [1, 2, 3]
list_2 = [4, 5, 6]

# Объединение списков
print('Объединение списков (list_1 + list_2) ->', list_1 + list_2)

# Объединение списков
list_1 += list_2
```

```
# Теперь li_1 == [1, 2, 3, 4, 5, 6]
print('Объединение списков (list_1 += list_2) ->', list_1)

# Объединяем с самим собой 3 раза
list_1 = [1, 2, 3]
print('Объединяем с самим собой 3 раза (list_1*3) ->', list_1 * 3)

# Объединение списков
list_3 = ['a', 'b', 'c']
print('Объединение списков (list_1 + list_3)', list_1 + list_3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.58. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.61).

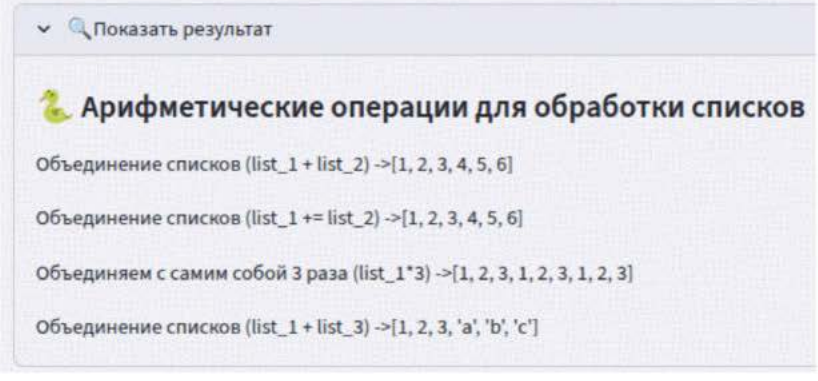


Рис. 2.61. Применение арифметических операций для обработки списков

Что касается инструкции проверки на вхождение in, то применительно к спискам она совершает поиск требуемого объекта (левый операнд) среди элементов целевого списка (правый операнд). В случае успеха операция возвращает True, иначе – False.

Пример программы применения оператора проверки на вхождение in для обработки списков приведен в листинге 2.59.

Листинг 2.59. Применение инструкции «in» для обработки списков

```
# Листинг 2.59
# Формируем список.
list_1 = [1, 2, 3, 4, 5]

# Проверяем наличие числа 8 в списке
answer = 5 in list_1
```

```
print('Число 5 присутствует в списке ->', answer)

# Проверяем наличие числа 8 в списке
answer = 8 in list_1
print('Число 8 присутствует в списке ->', answer)

# Проверяем наличие числа 8 в списке
answer = 8 not in list_1
print('Число 8 отсутствует в списке ->', answer)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.59. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.62).

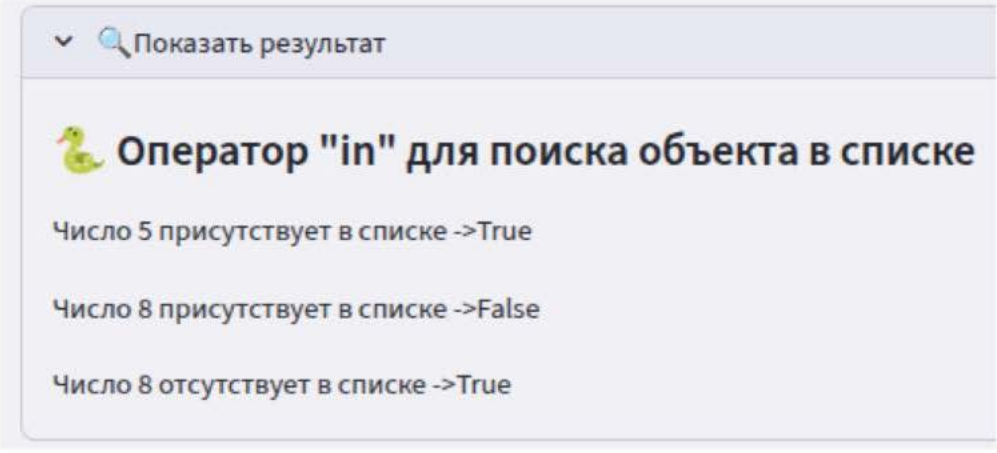


Рис. 2.62. Применение инструкции «in» для обработки списков

Напомним, что инструкция in может использоваться не только самостоятельно, но и совместно с инструкцией отрицания not, которая инвертирует возвращаемое логическое значение инструкции in.

Если возникает необходимость перебрать элементы списка, то, по аналогии со строками, можно использовать инструкцию организации цикла «for». Мы уже знакомились с этой инструкцией при изучении строк.

Пример программы применения инструкции «for» для выборки элементов списков приведен в листинге 2.60.

Листинг 2.60. Применение инструкции «for» для выборки элементов списков

```
# Листинг 2.60
# Формируем список.
list_1 = ['A', 'B', 'C', 'D', 'E', "F"]

# выводим элементы списка
for i in range(0, 6):
    print('Индекс i=', i, ' Элемент списка этим индексом->', list_1[i])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.60. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.63).

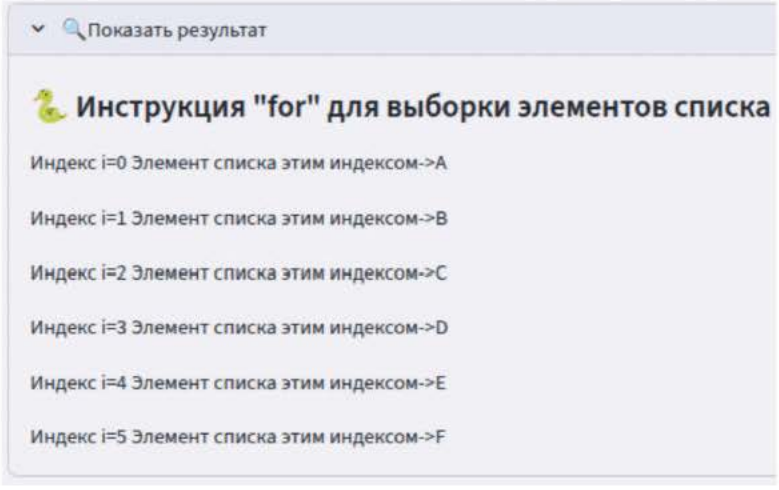


Рис. 2.63. Применение инструкции «for» для выборки элементов списков

Еще одной полезной и универсальной инструкции языка Python является инструкция удаления «del», которая позволяет с легкостью удалять переменные, элементы по индексу или ключу, атрибуты объектов и т.д. Применительно к спискам мы можем использовать «del» для удаления всего списка, срезов или отдельных его элементов.

Пример программы применения инструкции «del» для обработки элементов списков приведен в листинге 2.61.

Листинг 2.61. Применение инструкции «del» для обработки элементов списков

```
# Листинг 2.61
# Формируем список
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print('Исходный список', list_1)
```

```
# Удаляем первый элемент списка
del list_1[0]
print('Удален первый элемент списка list_1[0] ->', list_1)

# Удаляем первые 2 элемента списка
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del list_1[0:2]
print('Удалены первые два элемента списка list_1[0:2] ->', list_1)

# Удаляем последний элемент списка
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del list_1[-1]
print('Удален последний элемент списка list_1[-1] ->', list_1)

# Удаляем последние 2 элемента списка
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del list_1[-2:]
print('Удалены последние 2 элемента списка list_1[-2:] ->', list_1)

# Удаляем срез списка
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del list_1[2:4]
print('Удален срез списка [2:4] ->', list_1)

# Удаляем каждый 2-й элемент списка
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del list_1[::2]
# Теперь li_1 == [3, 7].
print('Удален каждый 2-й элемент списка [::2] ->', list_1)

# Удаляем сам список
del list_1
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.61. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.64).



Рис. 2.64. Применение инструкции «del» для обработки элементов списков

Здесь последней инструкцией удален весь список, то есть, уничтожен сам объект. Попытка вывести на печать несуществующий объект сгенерирует ошибку.

На этом рассмотрение базовых операций над списками мы заканчиваем. Теперь рассмотрим методы класса list и встроенные функции, применимые к спискам.

2.4.4. Функции и методы списков в Python

Как и строки, списки поддерживают ряд методов. Однако не стоит забывать, что списки относятся к изменяемым последовательностям, поэтому многие методы списков, в отличие от методов строк, изменяют список непосредственно, а не возвращают его измененную копию.

Перечислим методы, позволяющие определить некоторые количественные параметры списков:

1. .count(x) – возвращает количество элементов в списке с указанным значением «x» (в отличие от аналогичного метода для строк, здесь указывать интервал для поиска нельзя, так как метод принимает только один аргумент);
2. .index(x[, start[, end]]) – возвращает индекс «i» первого вхождения элемента [i] со значением «x» в списке или в срез списка [start:end], если он задан (при использовании среза отсчет индекса элемента все равно стартует с начала всего списка).

Пример программы применения методов «count» и «index» для обработки списков приведен в листинге 2.62.

Листинг 2.62. Применение методов «count» и «index» для обработки списков

```
# Листинг 2.62
# Создаем список для манипуляций.
list_1 = ['a', 'b', 'c', 'a', 'b', 'c', 'A', 'B', 'C']

# Исходный список
print('Исходный список ->', list_1)

# Количество элементов со значением - b
n_1 = list_1.count('b')
print('Количество элементов со значением "b" ->', n_1)

# Количество элементов со значением - d
n_2 = list_1.count('d')
print('Количество элементов со значением "d" ->', n_2)

# Индекс 1-го элемента со значением - c
index_1 = list_1.index('c')
print('Индекс 1-го элемента со значением - "c" ->', index_1)

# Первый элемента со значением - "c" не с начала списка
index_2 = list_1.index('c', 3)
print('Первый элемента со значением - "c" не с начала списка ->',
index_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.62. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.65).



Рис. 2.65. Применение методов «count» и «index» для обработки списков

Теперь рассмотрим методы, позволяющие добавлять в списки новые элементы:

- .insert(i,x) – вставляет элемент со значением «x» в указанную позицию «i». Тот же результат можно получить и при помощи инструкции [i:i] = [x].
- .append(x) – добавляет элемент со значением «x» в конец списка (то же самое что и [len(list):len(list)] = [x]).
- .extend(iterable) – добавляет элементы из итерируемого объекта «iterable» в конец списка (то же самое, что и [len(list):len(list)] = iterable) (не забываем, что итерируемыми являются, например, строки, списки, кортежи, но не числа).

Пример программы применения методов добавления в списки новых элементов приведен в листинге 2.63.

Листинг 2.63. Применение методов добавления в списки новых элементов

```
# Листинг 2.63
# Создаем список для манипуляций.
list_1 = ['a', 'b', 'c']
print('Исходный список->', list_1)

# Вставляем 'F' перед 'b'
list_1.insert(1, "F")
print('Вставляем "F" перед "b" ->', list_1)

# Вставляем 'G' перед 'b' через срез
list_1 = ['a', 'b', 'c']
list_1[1:1] = ['G']
print('Вставляем "G" перед "b" через срез ->', list_1)

# Добавляем в конец списка другой список
list_1 = ['a', 'b', 'c']
list_2 = [1, 2, 3]
list_1.append(list_2)
print('Добавляем в конец списка другой список ->', list_1)

# Добавляем в конец списка элементы другого списка (через len)
list_1 = ['a', 'b', 'c']
list_2 = [9, 8, 7]
list_1[len(list_1):len(list_1)] = list_2
print('Добавляем в конец списка элементы другого списка (через len) ->',
list_1)

# Добавляем в конец списка элементы другого списка (через +=)
list_1 = ['a', 'b', 'c']
list_1 += ['A', 'B', 'C']
print('Добавляем в конец списка элементы другого списка (через +=)->',
list_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел

«Листинг 2.63. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.66).

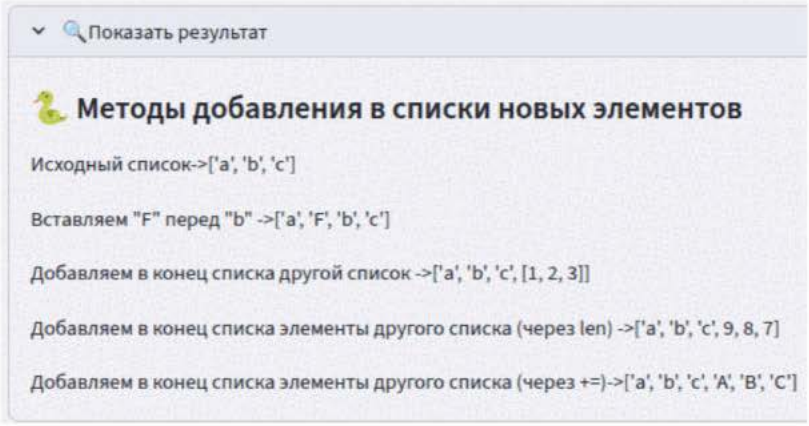


Рис. 2.66. Применение методов добавления в списки новых элементов

Теперь рассмотрим методы, позволяющие удалять элементы из списков:

- remove(x) – удаляет из списка первый попавшийся элемент, имеющий значение «x» (если элемента с указанным значением в списке не окажется, будет вызвано исключение ValueError);
- pop([i]) – удаляет из списка элемент с указанным индексом «i», а затем возвращает его (если индекс не указан, удаляется последний элемент списка, так как по умолчанию «i» принимается равным -1);
- clear() – удаляет все элементы из списка (данный метод аналогичен оператору del listi[:]).

Пример программы применения методов удаления элементов из списка приведен в листинге 2.64.

Листинг 2.64. Применение методов удаления элементов из списка

```
# Листинг 2.64
# Создаем список
list_1 = ['000', '111', '222', '333']
print('Исходный список ->', list_1)

# Удаляем элемент "111"
list_1.remove('111')
print('Удаляем элемент "111" ->', list_1)

# Удаляем элемент списка с индексом 2
list_1 = ['000', '111', '222', '333']
del_elem = list_1.pop(2)
```

```
print('Список без удаленного элемента [2]->', list_1)
print('Удаленный элемент [2] ->', del_elem)

# Очищаем список
list_1.clear()
print('Пустой список ->', list_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.64. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.70).

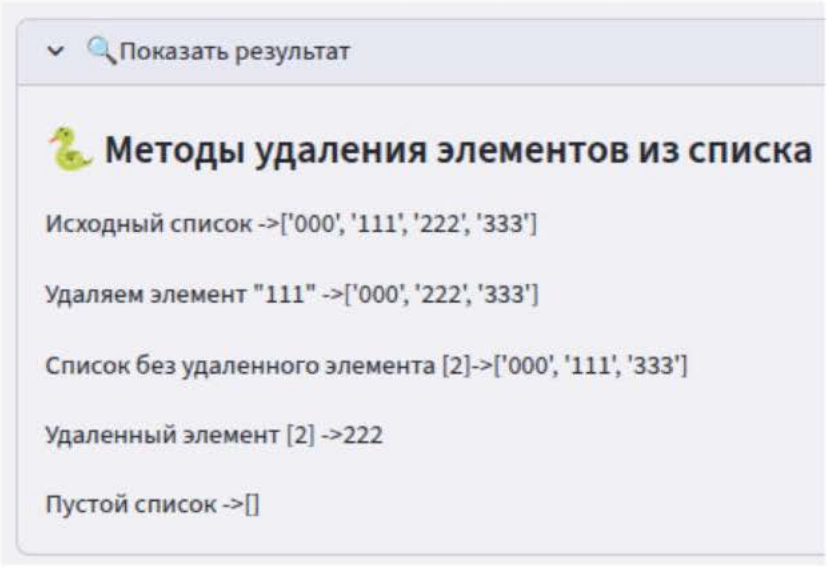


Рис. 2.70. Применение методов удаления элементов из списка

Как и для строк, для списков есть метод изменения порядка следования элементов на обратный, а также метод, позволяющий получить копию списка:

- reverse() – изменяет порядок следования элементов в списке на обратный (получить тот же результат можно и с помощью инструкции list = list[::-1]).
- copy() – возвращает копию списка (получить тот же результат можно и с помощью инструкции li[:]).

Пример программы применения методов «reverse» и «copy» приведен в листинге 2.65.

Листинг 2.65. Применение методов «reverse» и «copy» для списков

```
# Листинг 2.65
# Создаем список
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print('Исходный список:', list_1)

# Изменяем порядок следования методом reverse()
list_1.reverse()
print('Обратный порядок списка методом reverse() ->', list_1)

# Изменяем порядок следования через срез [::-1]
list_1 = list_1[::-1]
print('Обратный порядок списка через срез [::-1] ->', list_1)

# Копируем список методом copy()
copy_list_1 = list_1.copy()
print('Копируем список методом copy() ->', copy_list_1)

# Копируем список через срез [:]
copy_list_1 = list_1[:]
print('Копируем список через срез [:] ->', copy_list_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.65. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.71).



Рис. 2.71. Применение методов «reverse» и «copy» для списков

Для списков есть метод, позволяющий изменения порядка следования элементов – sort(). Данный метод имеет два аргумента:

- «key» – значение, которое должно быть именем функции, принимающей один аргумент и возвращающей ключ для сортировки списка.
- «reverse» – параметр, указывающий порядок сортировки списка (по умолчанию False – прямой порядок сортировки «по возрастанию», True - обратный порядок сортировки «по убыванию»).

Пример программы применения метода «sort» приведен в листинге 2.66.

Листинг 2.66. Применение метода «sort» для списков

```
# Листинг 2.66
# Создаем список
list_1 = ['Г', 'В', 'Б', 'А']
print('Исходный список:', list_1)

# Сортировка списка по умолчанию
list_1.sort()
print('Сортировка списка по умолчанию ->', list_1)
print('-'*40)

# Сортировка списка при reverse=True
list_1 = ['1', '2', '3', '4']
print('Исходный список:', list_1)
list_1.sort(reverse=True)
print('Сортировка списка при reverse=True ->', list_1)
print('-'*40)

# Вложенный список (фио, возраст, пол)
fio_age_gender = [['Тян П.П.', 20, 'М'], ['Пак С.С', 40, 'М'], ['Ли
М.М.', 30, 'Ж']]
print('Исходный вложенный список:', fio_age_gender)
print('-'*40)

# Сортировка вложенного списка по умолчанию
fio_age_gender.sort()
print('Сортировка вложенного списка по умолчанию ->', fio_age_gender)
print('-'*40)

# функция пользователя
def my_sort(person):
    return person[1]

# Сортировка вложенного списка по ключу key
fio_age_gender.sort(key=my_sort)
print('Сортировка вложенного списка по возрасту ->', fio_age_gender)
print('-'*40)
```

Опишем некоторые особенности этого программного модуля. Дело в том, что для сортировки по ключу «key» необходимо использовать пользовательскую функцию, а мы с

ними еще не сталкивались. Пользовательские функции будут подробно описаны в одной из следующих глав, а пока рассмотрим только основные правила создания пользовательских функций.

Пользовательская функция начинается с ключевого слова «def», за ним следует имя функции, в скобках указывается перечень принимаемых параметров, а заканчивается строка двоеточием. В нашем примере это записано в строке:

```
def my_sort(person):
```

В теле функции могут быть различные манипуляции с данными, а заканчивается она оператором return, после которого может быть перечень возвращаемых значений. В нашем примере это строка имеет вид:

```
return person[1]
```

В данном примере с использованием пользовательской функции выполнялось сортировка вложенного списка «fio_age_gender», в котором содержится информация о персонах:

- фамилия имя отчество;
- возраст;
- пол.

По умолчанию вложенный список сортируется по первому элементу - в нашем случае по «фио», что было сделано с использованием строки - fio_age_gender.sort(). А вот для сортировки по второму элементу вложенного списка (возраст) необходимо использовать пользовательскую функцию. Здесь мы создали пользовательскую функцию my_sort. Сортировка списка fio_age_gender по ключу «key» задана строкой:

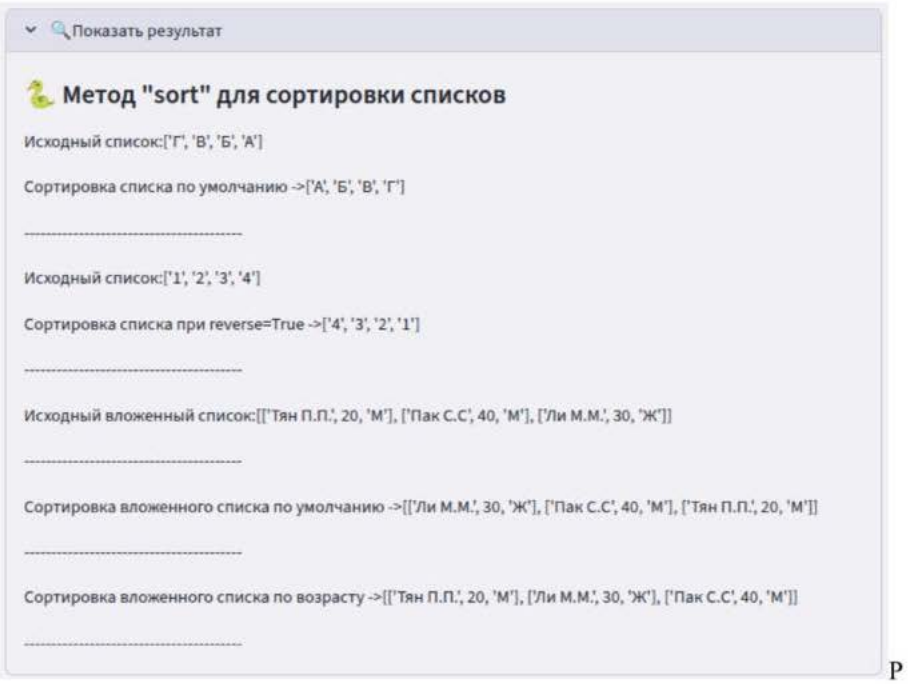
```
fio_age_gender.sort(key=my_sort)
```

Здесь мы указали идентификатор вложенного списка «fio_age_gender», а ключу «key» присвоили имя пользовательской функции «my_sort». Пользовательская функции вернет индекс элемента вложенного списка, по которому будет производиться его сортировка, т.е. индекс [1] (return person[1]). В нашем примере в этом случае будет выполнена сортировка по возрасту персоны.

Может быть, при первом знакомстве этот подход вам показался довольно сложным, но листинг программы и результаты ее работы позволит разобраться в такой конструкции метода sort().

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.66. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.72)



Р

Рис. 2.72. Применение метода «sort» для списков

Помимо собственных методов к спискам применим и ряд встроенных функций:

- min() - возвращает элемент с минимальным значением, при этом элементы должны быть одного типа с возможностью сравнения;
- max() - возвращает элемент с максимальным значением, при этом элементы должны быть одного типа с возможностью сравнения;
- sum() - возвращает сумму элементов списка, при этом элементы должны быть числами;
- len() - возвращает количество элементов в списке;
- sorted(iterable, *, key=None, reverse=False) - встроенная функция сортировки.

Пример программы применения этих встроенных функций приведен в листинге 2.67.

Листинг 2.67. Применение встроенных функций для списков

```
# Листинг 2.67
# Создаем список
list_1 = [1, 2, 3, 4, 5, 6]
print('Исходный список->', list_1)
print('Минимальное значение ->', min(list_1))
print('Максимальное значение ->', max(list_1))
print('Сумма ->', sum(list_1))
print('Количество элементов ->', len(list_1))
print('Обратная сортировка ->', sorted(list_1, reverse=True))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.67. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.73)

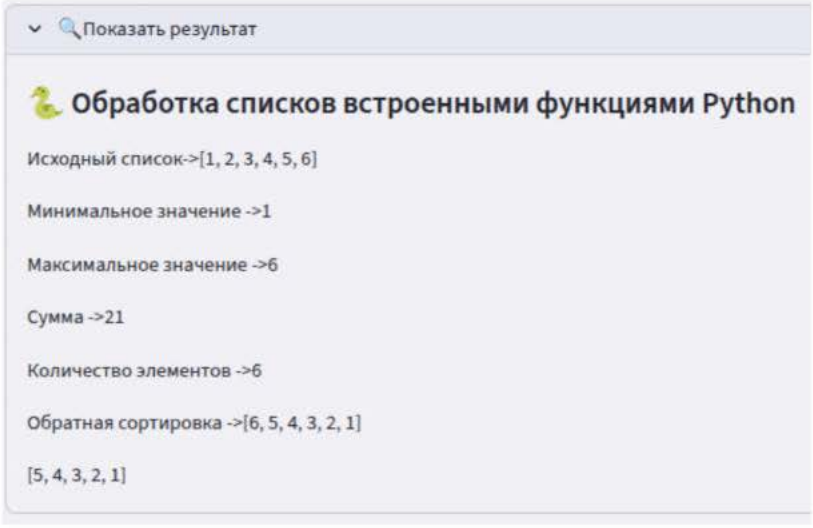


Рис. 2.73. Применение встроенных функций для списков

2.4.5. Краткие итоги раздела 2.4

1. В Python списки представляют собой упорядоченные изменяемые коллекции объектов произвольного типа с доступом по индексу.
2. Чтобы создать список достаточно перечислить требуемые объекты (элементы списка) через запятую внутри квадратных скобок. Доступ к элементу списка можно получить по его индексу, то есть порядковому номеру элемента в списке. При этом следует помнить, что индексация элементов в списках начинается с нуля.
3. Также списки могут быть созданы при помощи конструктора list(iterable) (например, инструкция list('abc') вернет список ['a', 'b', 'c']) и генератора списков (например, [simb for simb in 'abc'] вернет тот же список ['a', 'b', 'c']).
4. Что касается срезов, то операция среза всегда возвращает новый список, состоящий из указанных в срезе элементов исходного списка. Сам же исходный список не изменяется.
5. Списки можно изменять: удалять или добавлять новые элементы, изменять значения уже существующих элементов.
6. К спискам применимы арифметические операторы («+», +=) - для объединения списков, «*» и «*=» для дублирования элементов списков указанное число раз.
7. К спискам применимы инструкции «in» и «for». Первая проверяет наличие требуемого элемента в списке, а вторая позволяет поочередно перебрать все элементы списка.

8. Для списков доступна инструкция удаления «del», которая позволяет удалять отдельные элементы списка, срезы элементов и сам список.
9. Списки имеют ряд методов, которые дают возможность выполнять широкий спектр операций над списками, а для их использования не нужно импортировать никаких модулей.

Не стоит забывать, что в отличие от строк, списки изменяемы, поэтому их методы изменяют сам список, а не возвращают его измененную копию.

2.4.6. Вопросы для самоконтроля к разделу 2.4



1. Какие из литералов относятся к типу список: ['Иванов', 'Петров'], 5.0, " Python"?
2. Какое значение будет присвоено переменной num в строке программы: List_1 = [1, 2, 3, 4, 5]; num = List_1[1]:
   - num=2;
   - num=1;
   - num=3;
   - num=5;
   - num=4.
3. Какое значение будет присвоено переменной num в строке программы: List_1 = [1, 2, 3, 4, 5]; num = List_1[-1]:
   - num=5;
   - num=1;
   - num=3;
   - num=2;
   - num=4.
4. Какое значение будет присвоено переменной num в строке программы: List_1 = [1, 2, 3, 4, 5]; num = List_1[3]:
   - num=4;
   - num=1;
   - num=3;
   - num=2;
   - num=5.
5. При помощи какого среза можно отсортировать список List_1 = [1, 2, 3, 4, 5] в обратном порядке:
   - List_1[::-1];
   - List_1[:-1:-1];
   - List_1[-1:-1:-1];
   - List_1[-1::].

2.5. Кортежи в Python

Кортеж (от англ. tuple) – это неизменяемая упорядоченная коллекция объектов произвольного типа. Сами объекты называются элементами кортежа, а доступ к ним может быть получен при помощи целочисленного индекса или среза.

Кортежи во всем похожи на списки, однако в отличие от списков кортежи не могут изменяться, поэтому многие операции применимые к спискам, становятся недоступными для кортежей. Кроме того, литералы кортежей записываются не в квадратных, а в круглых скобках.

2.5.1. Создание кортежей в Python

Кортежи в Python создается при помощи круглых скобок, внутри которых через запятую перечисляются элементы кортежа. В случае необходимости последний элемент кортежа можно также завершать запятой. Это никак не повлияет на количество элементов кортежа, а значит и его длину. Однако если кортеж состоит всего из одного элемента, то запятую после него нужно указывать обязательно, иначе интерпретатор не сможет отличить кортеж от простого значения или выражения.

Пример программы, демонстрирующей создание кортежей, приведен в листинге 2.68.

Листинг 2.68. Создание кортежей

```
# Листинг 2.68
# Пустой кортеж
tpl_1 = ()
print('Пустой кортеж->', tpl_1)

# Кортеж с одним элементом
tpl_2 = (3,)
print('Кортеж с одним элементом->', tpl_2)

# Кортеж из нескольких элементов
tpl_3 = (15, "Иван", None, 34.5, True)
print('Кортеж из нескольких элементов->', tpl_3)

# Кортеж из нескольких элементов с запятой в конце
tpl_4 = (1, 2, 3, 4, 5,)
print('Кортеж из нескольких элементов с запятой в конце->', tpl_4)

# Кортеж с вложенными элементами
tpl_5 = ({'1': 'top', '2': ['abc', (4, 5)]}, 0.3)
print('Кортеж с вложенными элементами->', tpl_5)

# Кортеж с вложенными элементами
tpl_5 = (
        {'1': 'top', '2': ['abc', (4, 5)]},
```

```
        0.3
        )
print('Кортеж с вложенными элементами->', tpl_5)

# Кортеж из чисел - скобки можно не указывать
tpl_6 = 1, 2, 3, 4, 5,
print('Кортеж из чисел - скобки можно не указывать>', tpl_6)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.68. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.74).

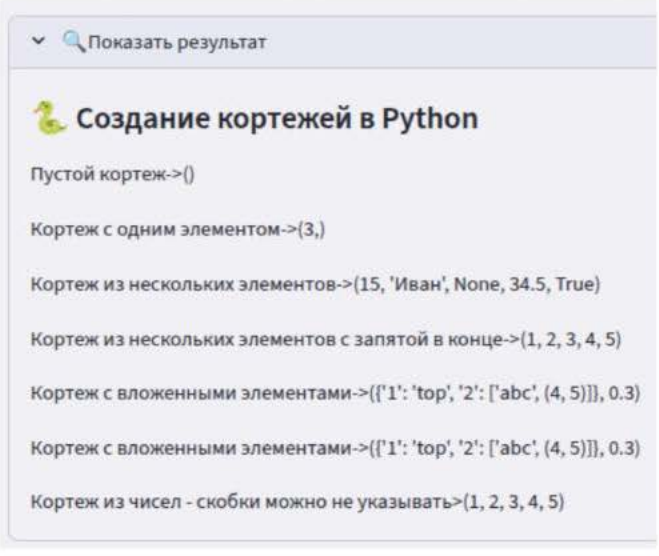


Рис. 2.74. Создание кортежей

Как видно из данного примера, внутри кортежей действительно можно использовать объекты любого типа данных, в любых комбинациях и уровнях вложенности. А благодаря наличию в синтаксисе круглых скобок, в программном коде сложные кортежи можно записывать в несколько строк, придав исходному коду удобочитаемый вид. При создании кортежа скобки можно не указывать, но лучше этого не делать, поскольку наличие скобок будет явно говорить, что это кортеж.

Создать кортеж можно и при помощи встроенного конструктора типа - tuple(). Если использовать его без аргумента, то он вернет нам пустой кортеж. Если же передать ему итерируемый объект, например, строку или список, то конструктор вернет кортеж, элементами которого будут соответствующие элементы переданного объекта.

Пример программы, демонстрирующей создание кортежей конструктором tuple(), приведен в листинге 2.69.

Листинг 2.69. Создание кортежей конструктором tuple()

```
# Листинг 2.69
# Пустой кортеж
tpl_1 = tuple()
print('Пустой кортеж->', tpl_1)

# Кортеж из строки
tpl_2 = tuple('12345')
print('Кортеж из строки->', tpl_2)

# Кортеж из списка
tpl_3 = tuple([155, "Иванов", None, 3.14, True])
print('Кортеж из списка->', tpl_3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.69. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.75).

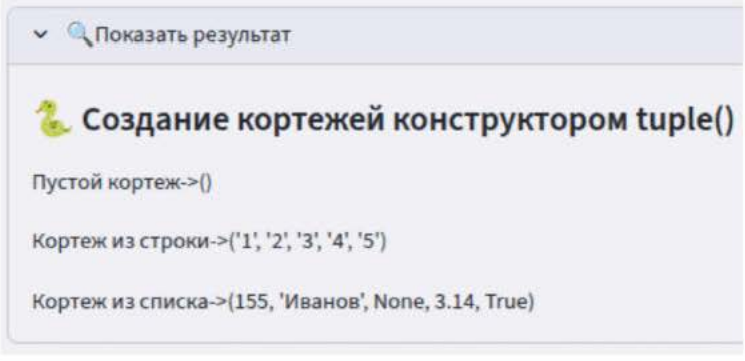


Рис. 2.75. Создание кортежей конструктором tuple()

2.5.2. Доступ к элементам кортежей по индексу и срезу

Операции доступа к элементу по индексу и получения срезов для кортежей имеют практически тот же самый синтаксис и смысл, что и для списков.

Пример программы, демонстрирующей доступ к элементам кортежа по индексу и по срезам, приведен в листинге 2.70.

Листинг 2.70. Доступ к элементам кортежа по индексу и по срезам

```
# Листинг 2.70
# Кортеж из целых чисел
tpl_1 = (1, 2, 3, 4, 5)
print('Кортеж из целых чисел', tpl_1)

# Получим 1-й элемент кортежа индекс: [0]
print('1-й элемент кортежа индекс: [0]', tpl_1[0])

# Последний элемент кортежа, индекс: [1]
print('Последний элемент кортежа. индекс: [1] ->', tpl_1[-1])

# Последний элемент кортежа, индекс: [len(tpl_1)-1]
print('Последний элемент кортежа, индекс: [len(tpl_1)-1] ->',
tpl_1[len(tpl_1)-1])

# Элементы кортежа по срезу [0:3]
print('Элементы кортежа по срезу [0:3] ->', tpl_1[0:3])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.70. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.76).



Рис. 2.76. Доступ к элементам кортежа по индексу и по срезам

Поскольку в результате обращения к элементу кортежа по его индексу возвращается некоторый объект, то для получения доступа к элементам на более глубоком уровне вложенности может потребоваться цепочка из нескольких индексов.

Пример программы, демонстрирующей доступ к элементам кортежа, который включает вложенные списки, приведен в листинге 2.71.

Листинг 2.71. Доступ к элементам кортежа, который включает вложенные списки

```
# Листинг 2.71
# Исходный кортеж с вложенными списками
tpl_1 = (3.14, ['Пес', [4, 5]], 'Кот')
print('Исходный кортеж с вложенными списками ->', tpl_1)

# Выводим 1-й элемент кортежа
print('Выводим 1-й элемент кортежа, [0] ->', tpl_1[0])

# Выводим строку Кот
print('Выводим строку Кот, [2][0:3]  ->', tpl_1[2][0:3])

# Выводим 3-й символ строки 'Пес'
print('Выводим 3-й символ строки "Пес", [1][0][2] ->', tpl_1[1][0][2])

# Выводим число 5 вложенного списка
print('Выводим число 5 вложенного списка, [1][1][1] ->', tpl_1[1][1][1])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.71. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.77).

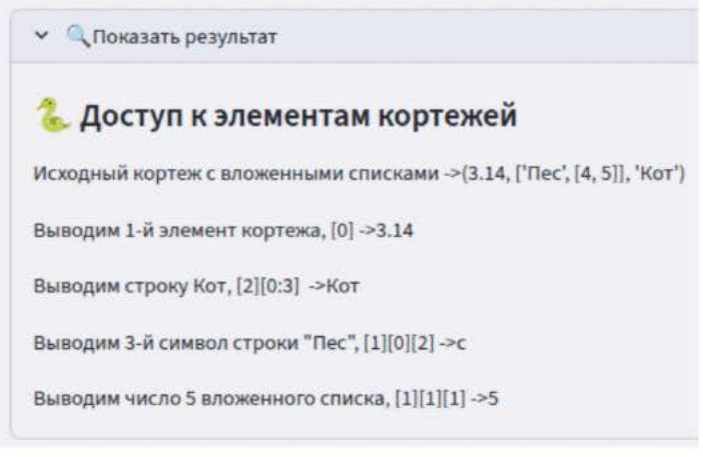


Рис. 2.77. Доступ к элементам кортежа, который включает вложенные списки

Следует отметить, что цепочка доступа к конкретному элементу может иметь и более сложную конструкцию, включающую не только квадратные скобки с индексами для доступа к элементам строк, кортежей или списков, но и квадратные скобки с ключами для доступа к элементам словарей, и точки для доступа к атрибутам объектов. Например, для кортежа:

```
tpl = ({'Первый': 'top', 'Второй': ['abc', [4, 5]]}, 0.3)
```

цепочка получения доступа ко второму элементу вложенного списка [4, 5] будет иметь вид tpl[0][' Второй '][1][1].

Пример программы, демонстрирующей обработку представленного выше кортежа, приведен в листинге 2.72.

Листинг 2.72. Обработки кортежа с вложениями

```
# Листинг 2.72
# Исходный кортеж с вложениями
tpl = ({'Первый': 'top', 'Второй': ['abc', [4, 5]]}, 0.3)
print('Исходный кортеж ->', tpl)

# Получить второй элемент вложенного списка [4, 5]
print("Второй элемент вложенного списка [4, 5], индекс-
[0]['Второй'][1][1]->", tpl[0]['Второй'][1][1])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.72. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.78).

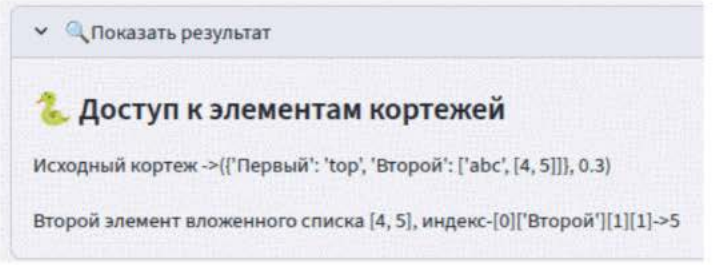


Рис. 2.78. Обработки кортежа с вложениями

Что касается получения срезов для кортежей, то для них справедлив тот же порядок, что и для других упорядоченных коллекций, то есть строк и списков.

Пример программы, демонстрирующей получение срезов кортежа, приведен в листинге 2.73.

Листинг 2.73. Получение срезов кортежа

```
# Листинг 2.73
# Кортеж из целых чисел
tpl_1 = (1, 2, 3, 4, 5, 6, 7, 8, 9)

# Исходный кортеж
```

```
print('Исходный кортеж ->', tpl_1)

# Первые два элемента
print('Первые два элемента, [0:2] ->', tpl_1[0:2])

# Первые два элемента
print('Первые два элемента, [:2] ->', tpl_1[:2])

# Последние два элемента
print('Последние два элемента, [-2:] ->', tpl_1[-2:])

# Элементы с 3-го по 5-й
print('Элементы с 3-го по 5-й, [2:5] ->', tpl_1[2:5])

# Элементы по отрицательным индексам
print('Элементы по отрицательным индексам, [-6:-2] ->', tpl_1[-6:-2])

# Копия кортежа
tpl_2 = tpl_1[:]
print('Копия кортежа, [:] ->', tpl_2[:])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.73. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.79).

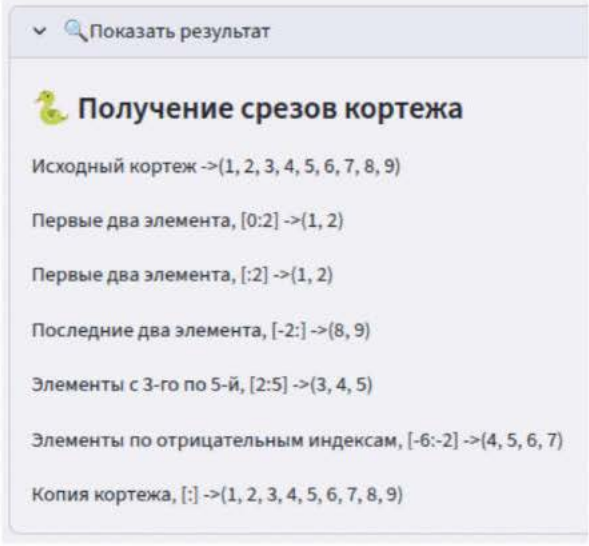


Рис. 2.79. Получение срезов кортежа

При извлечении срезов следует помнить, что нумерация элементов в кортежах начинается с нуля, а элемент с индексом второго предела, указанного в квадратных скобках, не будет присутствовать в срезе.

Как и в случае со строками и списками, один или оба предела среза можно опускать. Если не указывать первый предел, интерпретатор будет начинать срез с первого элемента кортежа (с индексом 0), а при отсутствии второго предела срез будет идти до конца кортежа. Если опустить оба предела, мы получим копию кортежа.

При необходимости можно указать и третий предел, который будет задавать шаг среза.

Пример программы, демонстрирующей получение срезов кортежа с использованием шага, приведен в листинге 2.74.

Листинг 2.74. Получение срезов кортежа с использованием шага

```
# Листинг 2.74
tpl_1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# Выводим его для наглядности.
print('Исходный кортеж ->', tpl_1)

# Каждый 2-й элемент среза (слева направо)
print('Каждый 2-й элемент среза (слева направо), [0:7:2] ->',
tpl_1[0:7:2])

# Каждый 2-й элемент среза (слева направо)
print('Каждый 2-й элемент среза (слева направо), [:7:2] ->',
tpl_1[:7:2])

# Каждый третий элемент всего кортежа
print('Каждый третий элемент всего кортежа, [::3] ->', tpl_1[::3])

# Каждый 2-й элемент, считая с конца среза
print('Каждый 2-й элемент, считая с конца среза, [8:2:-2] ->',
tpl_1[8:2:-2])

# Каждый 3-й элемент среза, считая справа налево
print('Каждый 3-й элемент среза, считая справа налево, [-3:1:-3] ->',
tpl_1[-3:1:-3])

# Каждый 2-й элемент, считая с конца кортежа
print('Каждый 2-й элемент, считая с конца кортежа, [::-2] ->', tpl_1[::-
2])
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.74. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.80).

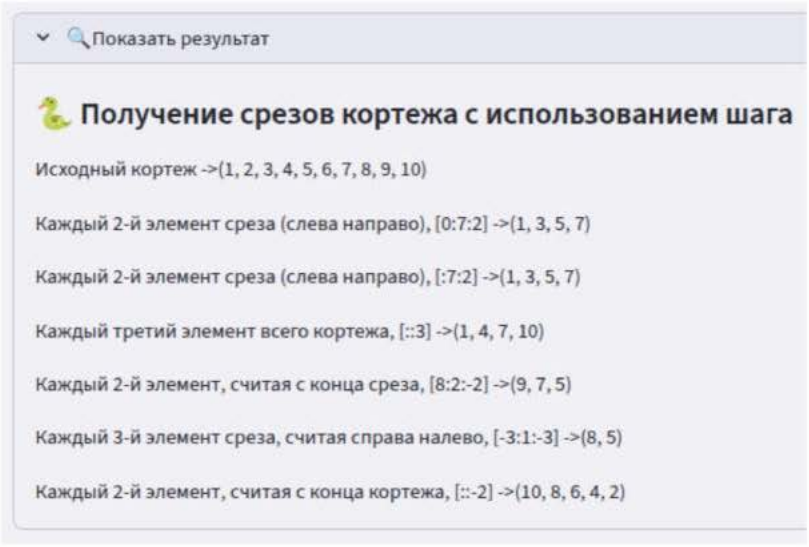


Рис. 2.80. Получение срезов кортежа с использованием шага

Как видим, шаг среза также может иметь отрицательное значение. В этом случае отсчет элементов кортежа и шагов ведется в противоположном направлении, а границы среза указываются справа налево, то есть индекс элемента, стоящего правее в кортеже должен указываться первым. Например, в срезе tpl_1[-3:1:-3] отсчет ведется справа налево, начиная с 3-го элемента, отсчитываемого с конца кортежа, до элемента с индексом 1, который находится левее и в срез не включается.

Неизменяемость кортежей означает, что для них запрещены операции добавления новых элементов или удаления уже существующих. При этом мы можем изменять конкретные элементы кортежа, если они это допускают (например, списки или словари), но удалить их уже не получится.

Пример программы, демонстрирующей изменение в кортеже элемента вложенного списка, приведен в листинге 2.75.

Листинг 2.75. Изменение в кортеже элемента вложенного списка

```
# Листинг 2.75
# Формируем кортеж из 3-х элементов
tpl = ('Иванов', ['Петров', [444, 555]], 100.35)
print('Исходный кортеж')
print(tpl)
print('-'*40)

# Изменяем 1-й эл-т вложенного списка [444, 555]
print("В [444, 555] Меняем 444 на 'Сидоров'")
tpl[1][1][0] = 'Сидоров'
print(tpl)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.75. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.81).

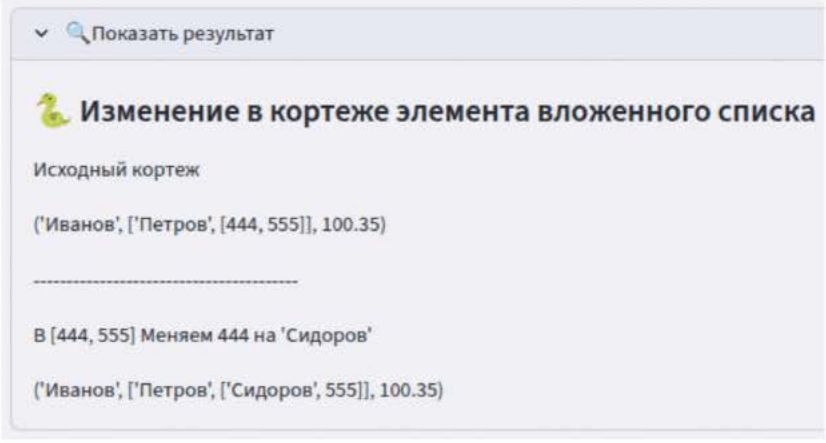


Рис. 2.81. Изменение в кортеже элемента вложенного списка

2.5.3. Операторы и инструкции обработки кортежей

Поскольку кортежи относятся к упорядоченным последовательностям, операторы «+», «+=», «*» и «*=» имеют для них практически тот же смысл, что и для списков и строк.

Пример программы, демонстрирующей выполнение арифметических операций с кортежами, приведен в листинге 2.76.

Листинг 2.76. Операторы и инструкции обработки кортежей

```
# Листинг 2.76
# Формируем кортежи
tpl_1 = (1, 2, 3)
tpl_2 = ('Иванов', 'Петров', 'Сидоров')
print('Исходный кортеж tpl_1->', tpl_1)
print('Исходный кортеж tpl_2->', tpl_2)
print('-'*40)

# Объединяем кортежи (tpl_1 + tpl_2)
tpl_3 = tpl_1 + tpl_2
print('Объединяем кортежи (tpl_1 + tpl_2) ->', tpl_3)

# Тиражируем кортеж (tpl_1 * 3)
tpl_4 = tpl_1 * 3
```

```
print('Тиражируем кортеж (tpl_1 * 3) ->->', tpl_4)

# Объединяем кортежи (tpl_1 += tpl_2)
tpl_1 += tpl_2
print('Объединяем кортежи (tpl_1 += tpl_2) ->', tpl_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.76. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.82).

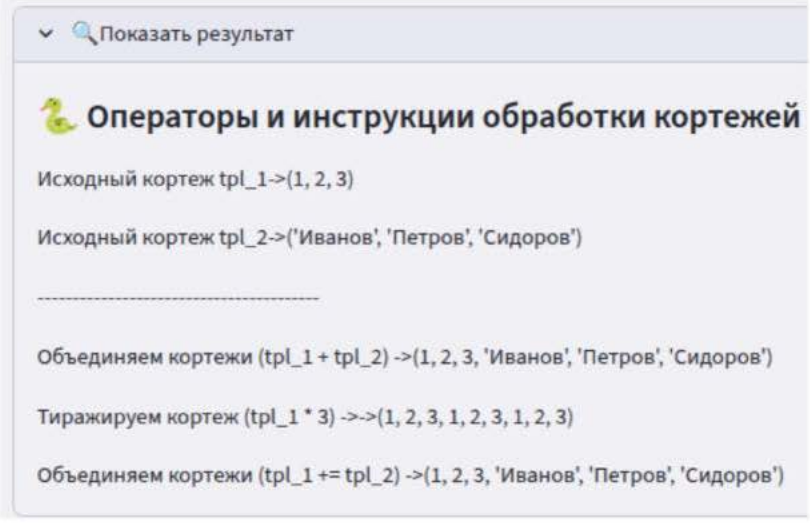


Рис. 2.82. Операторы и инструкции обработки кортежей

В отношении элементов кортежа можно использовать инструкцию проверки на вхождение «in», которая совершает поиск требуемого объекта (левый операнд) среди элементов целевого кортежа (правый операнд). В случае успеха операция возвращает «True», иначе – «False».

Пример программы, демонстрирующей поиск информации в кортеже с использованием инструкции проверки на вхождение «in», приведен в листинге 2.77.

Листинг 2.77. Поиск информации в кортеже с использованием инструкции «in»

```
# Листинг 2.77
# Формируем кортеж
tpl = (1, 2, 3, [4, 5])
print('Исходный кортеж->', tpl)
print('-'*35)
```

```
answer = 2 in tpl
print('Число 2 есть в tpl? ->', answer)

answer = 5 in tpl
print('Число 5 есть в tpl? ->', answer)

answer = 5 in tpl[3]
print('Число 5 есть в tpl[3]? ->', answer)

answer = 2 in tpl[3]
print('Число 2 есть в tpl[3]? ->', answer)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.77. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.83).

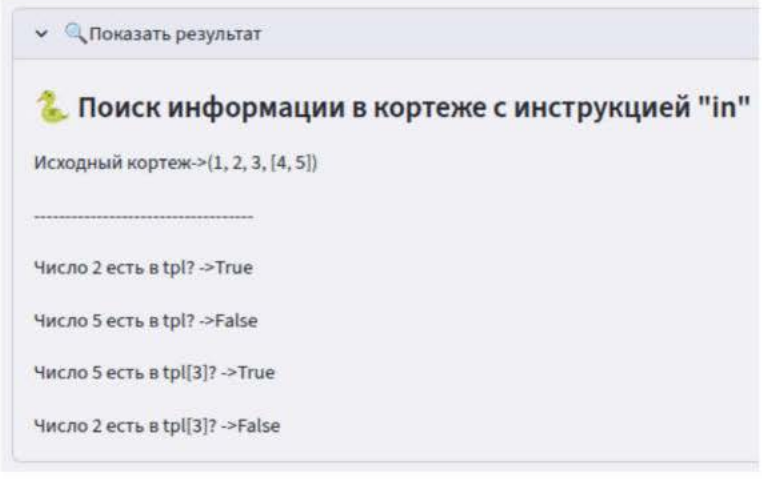


Рис. 2.83. Поиск информации в кортеже с использованием инструкции «in»

Напомним, что инструкция «in» может использоваться не только самостоятельно, но и совместно с инструкцией отрицания «not», которая инвертирует возвращаемое логическое значение инструкции «in».

Если возникает необходимость в переборе всех элементов кортежа, то, по аналогии со строками и списками, можно использовать инструкцию цикла «for».

Пример программы, демонстрирующей перебор всех элементов кортежа с использованием инструкции «for», приведен в листинге 2.78.

Листинг 2.78. Перебор элементов кортежа с использованием инструкции «for»

```
# Листинг 2.78
# Формируем кортеж
tpl_1 = ('Один', 'Два', 3, 4, 'Пять')

# Перебор всех элементов кортежа
for i in range(len(tpl_1)):
    print('Индекс ->', i, 'Элемент кортежа->', tpl_1[i])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.78. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.84).

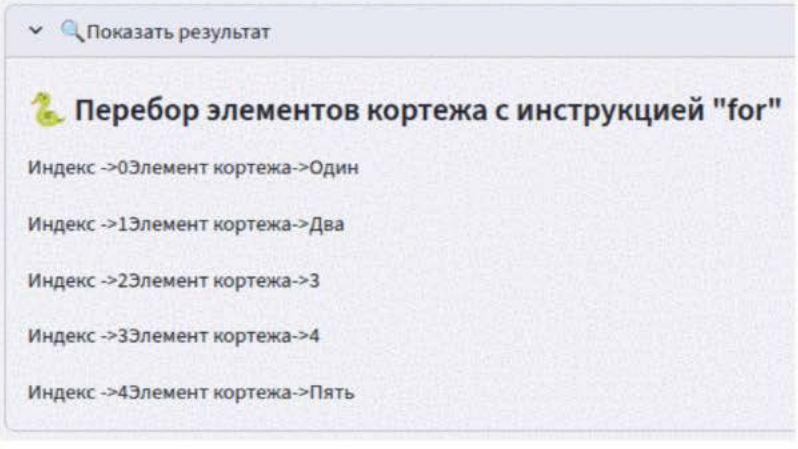


Рис. 2.84. Листинг 2.78. Перебор элементов кортежа с использованием инструкции «for»

2.5.4. Функции и методы для обработки кортежей

Что касается методов кортежей, то для них применимы общие методы упорядоченных последовательностей, которые не приводят к их изменению. К таким методам относятся:

- count(x) – возвращает количество элементов с указанным значением «x» в целевом кортеже (здесь указывать интервал для поиска нельзя, т.к. метод принимает только один аргумент);
- index(x) – возвращает индекс «i» первого вхождения элемента со значением «x», или в срезе кортежа [x, start, end], если он задан.
- у кортежей, при работе со срезами, может быть задано три аргумента:
- «x»- значение искомого элемента;
- «start» – индекс элемента, с которого начинается поиск;
- «end» индекс элемента, на котором заканчивается поиск.

Для обработки кортежей имеются и встроенные функции, которые обеспечивают доступ к элементам кортежа, но не изменяют его. К таким функциям относятся:

- len() - возвращает количество элементов в кортеже;
- min() - возвращает элемент с минимальным значением, при этом элементы должны быть одного типа;
- max() - возвращает элемент с максимальным значением, при этом элементы должны быть одного типа;
- sum() - возвращает сумму элементов кортежа, при этом элементы должны быть числами;

Пример программы, демонстрирующей методы и функции для обработки кортежей, приведен в листинге 2.79.

Листинг 2.79. Методы и функции для обработки кортежей

```
# Листинг 2.79
# Создаем кортеж
tpl = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8)

# Ищем количество элементов со значением - 9
n_1 = tpl.count(9)
print('Количество чисел 9 ->', n_1)

# # Ищем количество элементов со значением - 55
n_2 = tpl.count(55)
print('Количество чисел 55 -> ->', n_2)

# Индекс 1-го элемента со значением '8' во всем кортеже
index_1 = tpl.index(8)
print('Индекс 1-го элемента со значением "8" во всем кортеже - 8 ->',
index_1)

# Индекс 1-го элемента со значением '9', начиная с 6-го элемента
index_2 = tpl.index(9, 6)
print('Индекс 1-го элемента со значением "9", начиная с 6-го элемента -
>', index_2)

# Индекс 1-го элемента со значением '9', между элементами 9-11
index_3 = tpl.index(9, 10, 11)
print('Индекс 1-го элемента со значением "9", между элементами 9-11 ->',
index_3)

# Количество элементов в кортеже
tpl_len = len(tpl)
print('Количество элементов в кортеже ->', tpl_len)

# Минимальное значение элемента кортежа
```

```
tpl_min = min(tpl)
print('Минимальное значение элемента кортежа ->', tpl_min)

# Максимальное значение элемента кортежа
tpl_max = max(tpl)
print('Максимальное значение элемента кортежа ->', tpl_max)

# Сумма значений элементов кортежа
tpl_sum = sum(tpl)
print('Сумма значений элементов кортежа ->', tpl_sum)

# Удалить кортеж
del tpl
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.79. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.85).

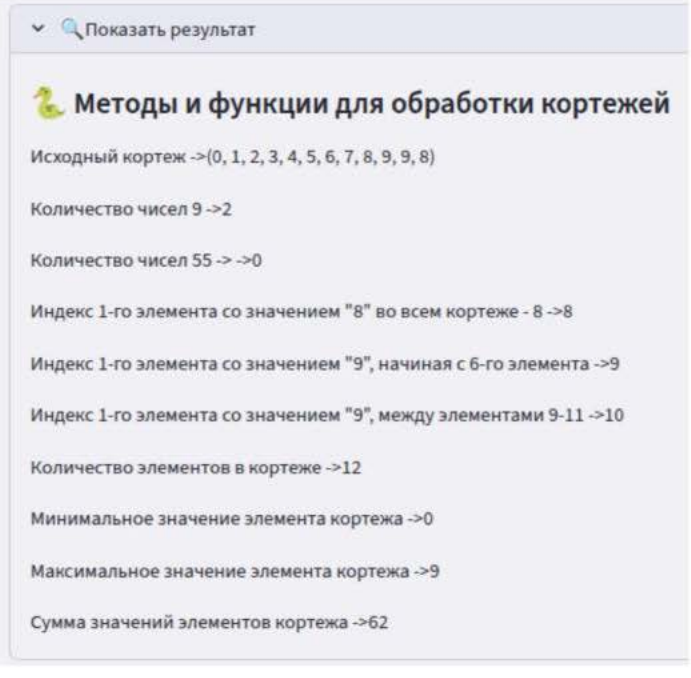


Рис. 2.85. Листинг 2.79. Методы и функции для обработки кортежей

Обратите внимание на последнюю строку данной программы. В отличие от списков, элементы кортежа не могут быть удалены с помощью инструкции «del», поскольку кортежи

неизменяемы. Чтобы удалить весь кортеж, мы можем использовать инструкцию «del» вместе с именем кортежа.

2.5.5. Отличие кортежей от списков

На первый взгляд кортежи похожи на списки, которые ограничены в возможностях (в частности они неизменяемые). Однако неизменяемость это одно из преимуществ кортежей. Благодаря этому свойству можно быть уверены, что кортеж не будет изменен в каком либо фрагменте программы. При выборе между списками и кортежами нужно руководствоваться следующими принципами:

- если в программе необходимо использовать упорядоченные коллекции объектов с возможностью их изменения - нужно выбирать списки;
- если в программе необходимы фиксированные коллекции объектов - следует выбирать кортежи.

Следует помнить, что хотя сами кортежи и неизменяемы, они могут содержать элементы изменяемых типов (например, списки или словари), а те могут быть изменены в программе. Такие кортежи нельзя использовать в качестве ключей словаря.

Второй важной особенностью кортежей является то, что за счет неизменяемости, а значит и более простой внутренней структуры, кортежи в общем случае занимают в памяти меньше места, чем списки. Как следствие, операции с кортежами являются более производительными, хотя и не такими гибкими.

Пример программы, в которой показана разность в размерах занимаемой памяти списками и кортежами, приведен в листинге 2.80.

Листинг 2.80. Размеры занимаемой памяти списками и кортежами

```
# Листинг 2.80
# Импортируем модуль sys
import sys

# Создаем и список и кортеж
list_1 = [100, 200, 300, 400, 500, 600, 700, 800, 900]
tpl_1 = (100, 200, 300, 400, 500, 600, 700, 800, 900)

# Получаем размеры
print('Размер list_1->', sys.getsizeof(list_1))
print('Размер tpl_1->', sys.getsizeof(tpl_1))
```

В этой программе был импортирован модуль sys из стандартной библиотеки Python, который позволяет получить объем памяти, занимаемой объектом.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.80. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.86).

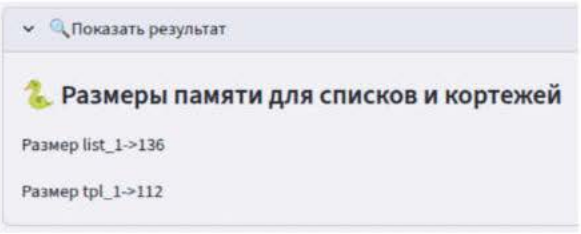


Рис. 2.86. Размеры занимаемой памяти списками и кортежами

2.5.6. Краткие выводы к разделу 2.5

1. В Python кортежи представляют собой упорядоченные, но неизменяемые коллекции объектов произвольного типа с доступом по индексу.
2. Чтобы создать кортеж достаточно перечислить требуемые объекты (элементы кортежа) через запятую внутри круглых скобок (не стоит забывать, что если создается кортеж с одним элементом, то после него следует ставить запятую), также кортежи могут быть созданы при помощи функции tuple().
3. После создания кортежа доступ к любому элементу кортежа можно получить по его индексу, то есть порядковому номеру элемента в кортеже (при этом следует помнить, что индексация элементов в кортежах начинается с нуля).
4. Из кортежей можно получать срезы, при этом срезы нельзя использовать для замены, удаления или добавления элементов в кортеж.
5. Для кортежей можно применять арифметические операторы «+», «+=», «*» и «*=» для объединения кортежей и для многократного дублирования картежа.
6. Инструкция «in» обеспечивает поиск требуемого элемента в кортеже, а инструкция «for» позволяет поочередно перебрать все его элементы.
7. Для кортежей доступны некоторые методы и функции.
8. Благодаря своей неизменяемости кортежи занимают в памяти меньше места, чем списки, а операции с ними являются более производительными.
9. Инструкция удаления «del» в отношении элементов кортежа неприменима, однако с использованием инструкции «del» можно удалить кортеж и освободить занимаемую им память.

2.5.77. Вопросы для самоконтроля к разделу 2.5



- Какие из литералов относятся к кортежу:
  - ➲ ('start', 'stopt');
  - ➲ '(3, 4)';
  - ➲ 5.5;
  - ➲ 'Python'.

- Задан кортеж: tpl_1=(1, 2, 3, 4, 5, 6, 7). Какое значение элемента кортежа будет получено по индексу [2]:
  - tpl_1[2] -> 3;
  - tpl_1[2] -> 2;
  - tpl_1[2] -> 1;
  - tpl_1[2] -> 4.
- Задан кортеж: tpl_1=(1, 2, 3, 4, 5, 6, 7). Какое значение элемента кортежа будет получено по индексу [-1]:
  - tpl_1[-1] -> 7,
  - tpl_1[-1] -> 6,
  - tpl_1[-1] -> 5,
  - tpl_1[-1] -> 4.
- Задан кортеж: tpl_1=(1, 2, 3, 4, 5, 6, 7). Какое значение элемента кортежа будет получено по индексу [:2]:
  - tpl_1[:2] -> (1, 2);
  - tpl_1[:2] -> (2, 3);
  - tpl_1[:2] -> (3, 4);
  - tpl_1[:2] -> (5, 6).
- Задан кортеж: tpl_1=(1, 2, 3, 4, 5, 6, 7). Какое значение элемента кортежа будет получено по индексу [2:5]:
  - tpl_1[2:5]-> (3, 4, 5);
  - tpl_1[2:5]-> (1, 2, 3);
  - tpl_1[2:5]-> (2, 3, 4);
  - tpl_1[2:5]-> (5, 6, 7).
- Какое значение нельзя поменять в кортеже (1, [2, 3, 4]):
  - 1;
  - 2;
  - 3;
  - 4.

2.6. Словари в Python

Словарь (от англ. dictionary) – это именованная, изменяемая, неупорядоченная коллекция объектов произвольного типа. Сами объекты называются элементами словаря, а доступ к ним может быть получен при помощи ключей словаря. Иногда словари еще называют ассоциативными массивами или хешами.

Так же как и списки, словари нужны для хранения произвольных объектов. Однако если в списках и картежах для доступа к элементам используются индексы, которые определяют позицию элемента в последовательности, то в словарях это делается при помощи ключей. Ключи представляют собой неизменяемые значения (числа, строки или кортежи), которые позволяют однозначно идентифицировать элементы словаря.

2.6.1. Создание словарей

Словари представляют собой конструкцию:

{Ключ1: Значение1, Ключ2: Значение2, Ключ3: Значение3, …}

Это последовательность, состоящая из элементов, заключенных в фигурные скобки. Каждый элемент последовательности состоит из уникального ключа, за которым через двоеточие следует значение этого ключа. Последний элемент словаря разрешается завершить запятой. Это никак не повлияет на количество элементов словаря, а значит и его длину.

Пример программы, в которой показано формирование словарей, приведен в листинге 2.81.

Листинг 2.81. Формирование словарей

```
# Листинг 2.81
# Словарь для хранения чисел
d_num = {'key1': 111, 'key2': 222, 'key3': 222}
print('Словарь d_num->', d_num)

# Словарь для хранения текста
d_txt = {1: 'Иванов', 2: 'Петров', 3: 'Сидоров'}
print('Словарь d_txt->', d_txt)

# Создаем кортеж
tpl_1 = (1, 2, 3)
# Словарь для разных объектов
d = {'str': 'Программа', 'num': 123, 'tuple': tpl_1}
print('Словарь d ->', d)

# Словарь с вложенными объектами
dict_1 = {'list': ['a', {0: (False, True), 1: None}], 'tpl': (1, 2, 3)}
print('Словарь dict_1 ->', dict_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.81. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.87).



Рис. 2.87. Формирование словарей

Как видно из данного примера, словари допускают возможность хранения объектов произвольного типа и любого уровня вложенности. Однако словари – это неупорядоченные коллекции, поэтому к ним не применимы операции, основанные на использовании порядка следования элементов или извлечение среза.

Благодаря наличию в синтаксисе фигурных скобок, которые обрамляют словарь, сложные и объемные элементы словарей в программном коде можно разносить на несколько строк, придавая коду удобочитаемый вид.

Создать словарь можно также при помощи встроенного конструктора dict(). Конструктор позволяет создавать словари разными способами:

1. Если использовать конструктор dict() без аргументов, то он вернет пустой словарь {}.
2. Если передать конструктору через запятую пары key_1 = val_1, key_2 = val_2, ..., то на выходе получим словарь с целевыми элементами {'key_1': val_1, 'key_': val_2, ...}.
3. Если передать конструктору список с кортежами пар значений [(key_1, val_1), (key_2, val_2), ...], то первое значение станет ключом, а второе элементом словаря.

Пример программы, в которой показано формирование словарей конструктором dict(), приведен в листинге 2.82.

Листинг 2.82. Формирование словарей конструктором dict()

```
# Листинг 2.82
# Пустой словарь
dict_1 = dict()
print('Пустой словарь ->', dict_1)

# Словарь из пар: «id = значение_id»
dict_2 = dict(one=1, two=2, str_1='Иванов', str_2='Петров')
print('Словарь из пар: "id = значение_id" ->', dict_2)

# Словарь из списока с кортежами пар
dict_3 = dict([('one', 1), ('two', 2), ('str_1', 'Иванов'), ('str_2',
'Петров')])
print('Словарь из списока с кортежами пар ->', dict_3)

# Словарь из допустимых ключей
dict_4 = dict([(1, 'one'), (2, 'two'), (3, 'Иванов'), (4, 'Петров')])
print('Словарь из допустимых ключей ->', dict_4)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.82. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.88).



Рис. 2.88. Формирование словарей конструктором dict()

Обратите внимание, что при передаче конструктору пар «имя_ключа = значение» в качестве ключей используются допустимые в Python идентификаторы, а не строки. Поэтому их в кавычки брать не нужно. Во втором же варианте можно использовать любые допустимые ключи, то есть не только строки, но и числа или кортежи.

Для создания словарей в Python могут использоваться генераторы словарей, которые применяют выражение к каждому элементу передаваемых итерируемых объектов и на основе их вычисления возвращают готовые словари с элементами.

Пример программы, в которой показано использование генераторов словарей, приведен в листинге 2.83.

Листинг 2.83. Использование генераторов словарей

```
# Листинг 2.83
# Словарь с квадратами чисел 2,3,4,5,6
print('Словарь с квадратами чисел 2,3,4,5,6')
d_1 = {x: x**2 for x in range(2, 7)}
print(d_1)
print('-'*40)

# Цикл с двумя переменными
print('Словарь цикле с двумя переменными')
d_2 = {k: v for k, v in [('a', 11), ('b', 22), ('c', 33), ('d', 44)]}
print(d_2)
print('-'*40)

# Словарь с ключами и значениями по умолчанию
print('Словарь с ключами и значениями по умолчанию')
d_3 = {k.lower(): None for k in 'ABC'}
print(d_3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.83. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.89).

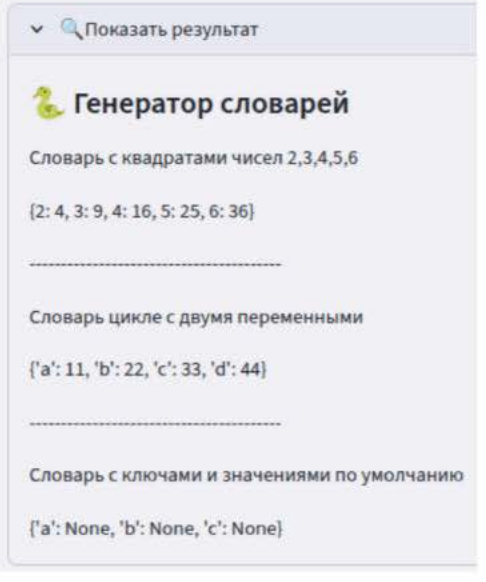


Рис. 2.89. Использование генераторов словарей

Таким образом, в простейшем случае для генерации словаря нужно в фигурных скобках через двоеточие написать выражения, которые будут применяться к каждому генерируемому ключу и/или элементу, и через пробел прописать соответствующий заголовок цикла «for» без завершающего двоеточия.

Для создания словарей в Python можно использовать специальный метод fromkeys(), который создает словарь из заданной последовательности ключей и значений.

Метод fromkeys(key, value) может принимать два параметра:

- key - ключи, которые могут быть любыми итерируемыми объектами, такими как строка, набор, список и т. д.
- value - значение ключей (это значение, которое может быть любого типа или любых итераций, таких как строка, список и т. д.).

Второй параметр value можно не указывать, в этом случае по умолчанию все элементы словаря получать значение None.

Пример программы, в которой показано использование метода fromkeys(), приведен в листинге 2.84.

Листинг 2.84. Создание словарей методом fromkeys()

```
# Листинг 2.84
# Ключи
keys = {'А', 'Б', 'В', 'Г', 'Д'}
# Значение ключей
value = 'Символ'
# Создаем словарь из ключей и значения
d_0 = dict.fromkeys(keys, value)
print('Словарь из ключей и значения->', d_0)

# Словарь с 3-мя элементами со значением None
d_1 = dict.fromkeys([1, 2, 3])
print('Словарь с 3-мя элементами со значением None->', d_1)

# Указываем значение для ключей в виде строки
d_2 = dict.fromkeys(range(0, 3), 'Строка')
print('Значение для ключей в виде строки->', d_2)

# Используем для ключей символы строки
d_3 = dict.fromkeys('ABCD', 345)
print('Используем для ключей символы строки->', d_3)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.84. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.90).

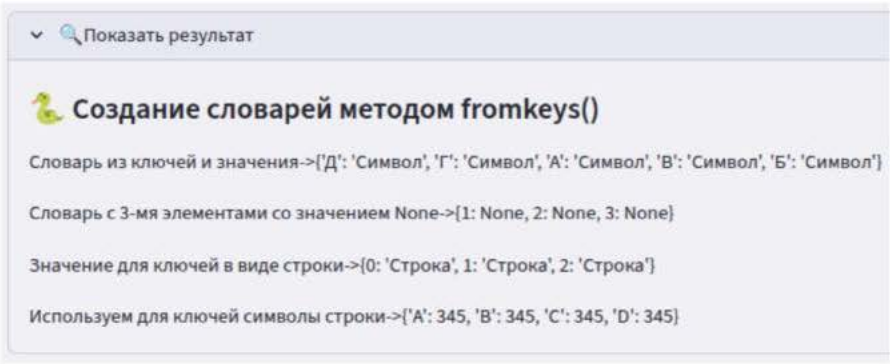


Рис. 2.90. Создание словарей методом fromkeys()

2.6.2. Ключи словаря в Python

В качестве ключей словаря в Python могут использоваться только объекты неизменяемых типов данных, так как они позволяют однозначно идентифицировать элементы словаря. Сюда относятся числа, строки, кортежи с неизменяемыми элементами, фиксированные множества, None, логические значения и другие неизменяемые типы данных.

Пример программы, в которой показано создание ключей словаря из данных различных типов, приведен в листинге 2.85.

Листинг 2.85. Создание ключей словаря из данных различных типов

```
# Листинг 2.85
# Импортируем модуль для создания чисел типа Decimal
from decimal import Decimal

# Используем для ключей строки
dict_1 = {'Первый': 1, 'Второй': 'Алфавит', 'Третий': 'А Б В Г Д ...'}
print('Ключи из строк->', dict_1)

# Используем для ключей числа
dict_2 = {1: 'Целое', -0.2: 'Отрицательное', 3j: 'Комплексное'}
print('Ключи из чисел->', dict_2)

# Используем числа типа Decimal
dict_5 = {Decimal(-0.5): 'Здесь ключ (-0.5)', Decimal(0.5): 'Здесь ключ
(0.5)'}
print('Ключи из чисел типа Decimal->', dict_5)

# Используем логические значения и None
dict_6 = {True: 0, False: 1, None: 'Не число'}
print('Ключи из логических значений и None->', dict_6)

# Кортежи с неизменяемыми элементами
```

```
dict_7 = {(True, 'a'): 1, (-0.7, (None, 5j)): 3}
print('Ключи из кортежей->', dict_7)

# Создаем фиксированное множество из элементов списка
fr_set1 = frozenset([-3.1, None, 'a'])
fr_set2 = frozenset([-3.1, None, 'd'])
print('--------- Ключи из фиксированного множества--------')
dict_8 = {fr_set1: '<-Это первый ключ', fr_set2: '<-Это второй ключ'}
print(dict_8)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.85. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.91).

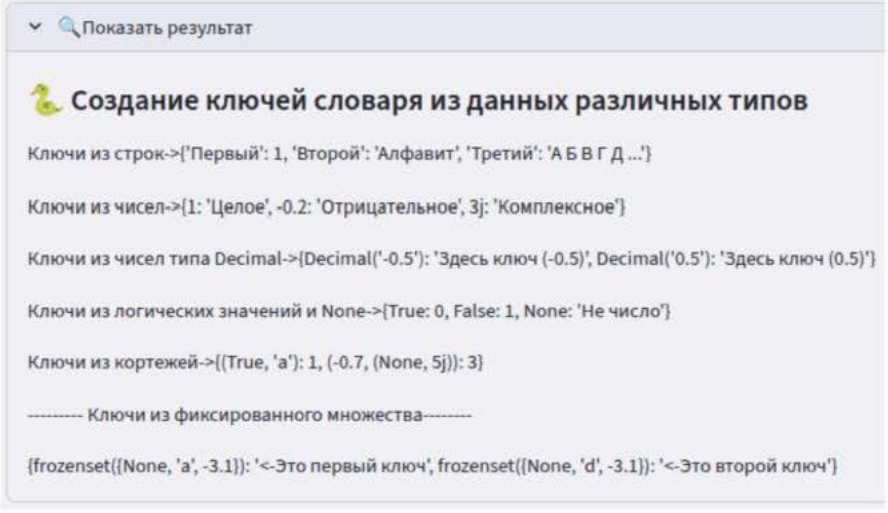


Рис. 2.91. Создание ключей словаря из данных различных типов

Конечно, формировать ключи из фиксированного множества это некий перебор, но вполне вероятно, что такая необходимость может возникнуть. Главным условием использования объектов в качестве ключей словаря является их неизменяемость. Использовать, например, списки или кортежи с изменяемыми элементами не получится.

В словаре нельзя использовать один и тот же ключ для нескольких элементов. В этом случае интерпретатор не выдаст ошибки, но по ключу будет доступен только один из таких элементов. В то же время один и тот же объект вполне может быть получен по разным ключам. Это продемонстрированно в программном модуле, который приведен в листинге 2.86.

Листинг 2.86. Ключи словаря и значения ключей

```
# Листинг 2.86
# Используем одинаковые ключи
d_1 = {'ok': 'Один',  'ok': 'Два'}
print('При одинаковом ключе "ok" получим только одно значение')
print('->', d_1['ok'], d_1['ok'])
print('-'*30)

# Используем одновременно 1 и True (для Python это одно значение -
"истина")
print('Для Python ключи 1 и True это одно значение - "1- истина"')
d_2 = {1: 'Это ключ 1', True: 'Это ключ True'}
print(d_2[1], d_2[True])
print('-'*30)

# Используем одновременно 0 и False (для Python это одно значение -
"ложь")
d_3 = {0: 'Это ключ 1', False: 'Это ключ 2'}
print('Для Python ключи 0 и False это одно значение - "0 - ложь"')
print(d_3[0], d_3[False])
print('-'*30)

# Сохраняем в словаре одно значение под разными ключами
d_3 = {'Да': 111, 'Нет': 111}
print('Разные ключи, одно значение')
print('Это элемент словаря с ключом "Да"->', d_3['Да'])
print('Это элемент словаря с ключом "Нет"->', d_3['Нет'])
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.86. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.92).

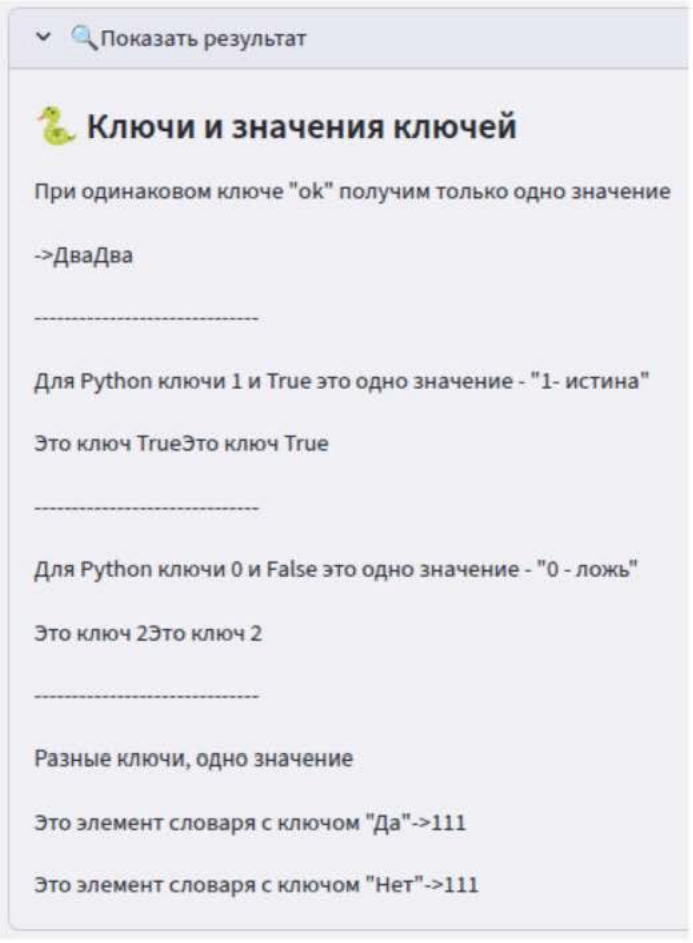


Рис. 2.92. Ключи словаря и значения ключей

Из данного примера видно, что в качестве ключей нельзя использовать значения «0» и «False» (интерпретатор считает это значение нулем), а также «1» и «True» (интерпретатор считает его единицей). К этой особенности мы еще вернемся, когда будем рассматривать логический тип данных.

2.6.3. Доступ к элементам словаря в Python

После того, как словарь будет создан, доступ к его элементам можно получать при помощи соответствующих ключей, которые нужно указывать в квадратных скобках после имени или литерала словаря.

Пример программы, в которой показано доступ к элементам словаря с использованием ключей, приведен в листинге 2.87.

Листинг 2.87. Доступ к элементам словаря с использованием ключей

```
# Листинг 2.87
# Формируем простой словарь
d_1 = {1: 'Один', 'Два': 'Символьная строка', 3: 555}
print('Исходный словарь - >', d_1)
print("Это значение по ключу - 1 ->", d_1[1])
print("Это значение по ключу - 'Два' ->", d_1['Два'])
print("Это значение по ключу - 3 ->", d_1[3])
print('-'*60)

# Используем вложенные списки и словари
d_1 = {'list': [{'Да': 1, 'Нет': 2}, 3], 'stroka': 'abc'}
print('Исходный словарь - >', d_1)
print("Это значение по ключу - d_1['list'][0]['Нет'] ->",
d_1['list'][0]['Нет'])
print("Это значение по ключу - d_1['list'][0]['Да'] ->",
d_1['list'][0]['Да'])
print("Это значение по ключу - d_1['list'][1] ->", d_1['list'][1])
print("Это значение по ключу - d_1['stroka'][1] ->", d_1['stroka'][1])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.87. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.93).

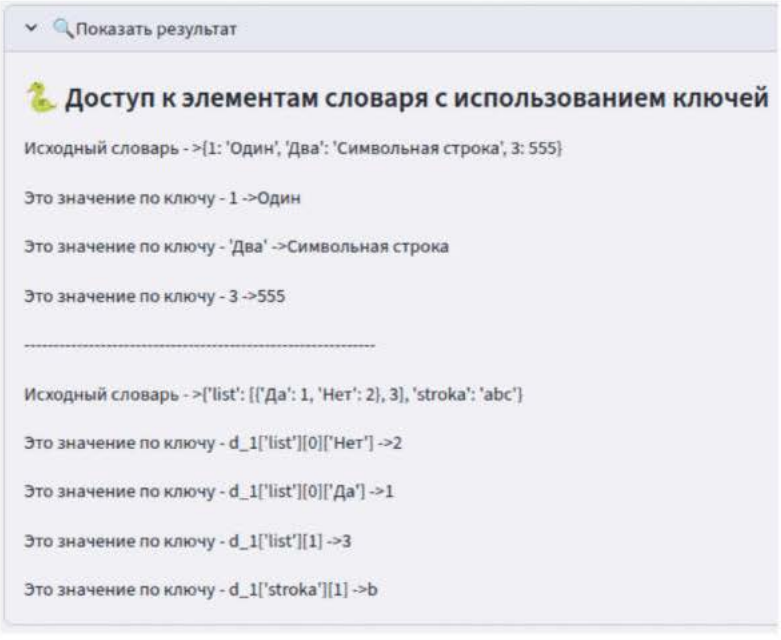


Рис. 2.93. Доступ к элементам словаря с использованием ключей

Как видно из данного примера, цепочка доступа к конкретному элементу словаря может быть довольно длинной. Более того, она может включать не только квадратные скобки с ключами для доступа к элементам словарей, но и квадратные скобки с индексами для доступа к элементам списков, строк, кортежей или даже точки для доступа к атрибутам объектов.

Что касается извлечения не одного, а сразу целого диапазона элементов, то для словарей такая операция не предусмотрена, так как словари являются неупорядоченными коллекциями объектов.

Изменяемость словарей означает, что мы можем не только получать доступ к элементам словарей, но также изменять их значения, добавлять в словарь новые элементы или удалять их. Изменить значение элемента в словаре можно следующим образом: найти элемент в словаре по ключу, а затем присвоить ему новое значение. При этом если указанный ключ в словаре будет отсутствовать, интерпретатор добавит в словарь новый элемент с этим ключом.

Пример программы, в которой показано изменение и добавление элемента словаря, приведен в листинге 2.88.

Листинг 2.88. Изменение и добавление элемента в словарь

```
# Листинг 2.88
# Формируем словарь
d_1 = {1: 1, 2: 2, 3: 333}
print('Исходный словарь - >', d_1)

# Изменяем значение элемента с ключом "3"
d_1[3] = 555
print('Значение элемента с ключом "3" изменено с 333 на 555 - >', d_1)

# Добавляем новый элемент
d_1[4] = 777
print('Добавлен элемент с ключом "4" и значением  777- >', d_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.88. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.94).

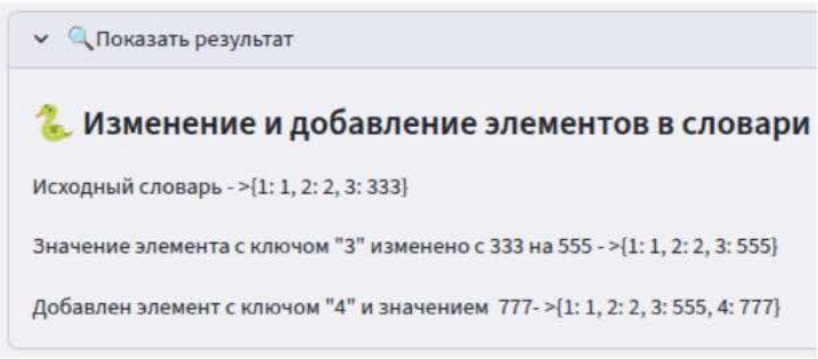


Рис. 2.94. Изменение и добавление элемента в словарь

2.6.4. Инструкции для обработки словарей

Поскольку словари относятся к неупорядоченным коллекциям объектов, операторы «+», «+=», «*» и «*=» с ними не используются. Что касается инструкции проверки на вхождение «in», то применительно к словарям она совершает поиск переданного ключа (левый операнд) среди ключей целевого словаря (правый операнд). В случае успеха эта операция возвращает «True», иначе – «False».

Пример программы, в которой показано применение для словарей инструкции проверки на вхождение «in», приведен в листинге 2.89.

Листинг 2.89. Применение для словарей инструкции «in»

```
# Листинг 2.89
# Формируем простой словарь.
d_1 = {1: 'Один', 2: 'Два', 3: 'Три'}
print('Исходный словарь - >', d_1)

# Наличие ключа - 2
print("Есть ли в словаре ключ -2 ? ->", 2 in d_1)
print("Есть ли в словаре ключ -4 ? ->", 4 in d_1)
print("Отсутствует ли в словаре ключ -4 ? ->", 4 not in d_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.89. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.95).

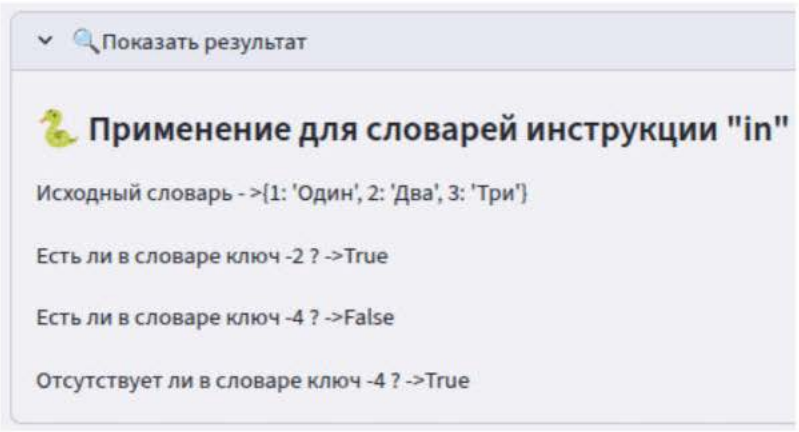


Рис. 2.95. Применение для словарей инструкции «in»

Со словарями можно использовать и инструкцию цикла «for», при этом перебирать словарь можно как по ключам словаря, так и по значениям ключей.

Пример программы, в которой показано, как можно использовать инструкцию цикла «for» для словарей, приведен в листинге 2.90.

Листинг 2.90. Применение для словарей инструкцию цикла «for»

```
# Листинг 2.90
# Формируем словарь.
d_1 = {1: 'Один', 2: 'Два', 3: 'Три'}
print('Исходный словарь - >', d_1)

# Цикл по ключам словаря
for key in d_1.keys():
    print('Ключи словаря ->', key)

# Цикл по значениям ключей словаря
for i in d_1:
    print('Значения ключей словаря ->', d_1[i])
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.90. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.96).

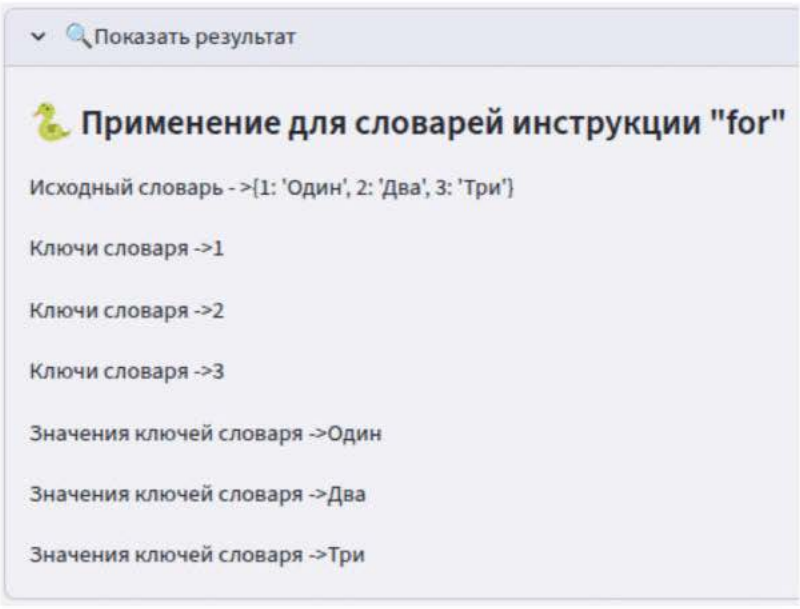


Рис. 2.96. Применение для словарей инструкцию цикла «for»

Поскольку словари относятся к изменяемым коллекциям объектов произвольного типа, их элементы мы можем, как добавлять, так и удалять. Последнюю операцию довольно просто выполнять при помощи знакомой нам инструкции удаления «del».

Пример программы, в которой показано, как можно добавить элемент в словарь и как можно удалить элемент из словаря с использованием инструкции «del», приведен в листинге 2.91.

Листинг 2.91. Добавление (удаление) элемента из словаря

```
# Листинг 2.91
# Формируем исходный словарь.
d_1 = {1: 'Один', 2: 'Два', 3: 'Три', 4: 'Четыре'}
print('Исходный словарь:', d_1)

# Добавляем новый элемент
d_1[5] = 'Пять'
print('Словарь с добавленным элементом (ключ - 5):', d_1)

# Удаляем элемент с ключом - 2
del d_1[2]
print('Словарь с удаленным элементом (ключ - 2):', d_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.91. Результаты работы модуля можно увидеть, открыв контейнер «Показать

результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.97).

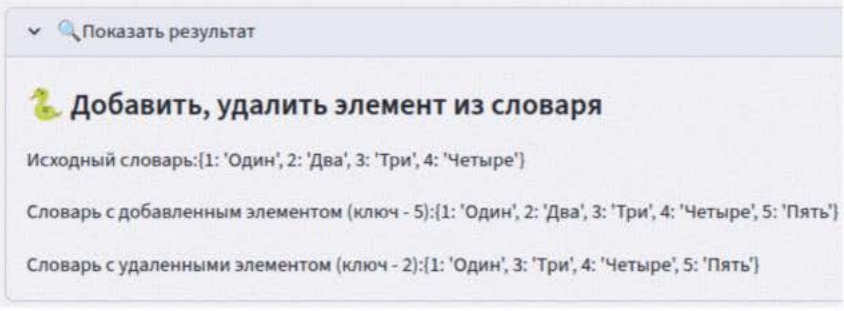


Рис. 2.97. Добавление (удаление) элемента из словаря

2.6.5. Функции и методы словарей в Python

Давайте рассмотрим имеющиеся у словарей методы. При этом не будем забывать, что словари относятся к изменяемым коллекциям объектов, поэтому многие методы словарей изменяют их непосредственно, а не возвращают измененную копию.

- get(key[,value]) – возвращает значение по ключу key. Второй параметр «value» не обязателен. Если ключа «key» в словаре не окажется, то метод вернет то значение, которое задано в параметре «value» (при этом значение «value» не будет добавлено в словарь). Этот параметр чаще используют в тех случаях, когда нужно вывести сообщение о том, что заданный ключ в словаре отсутствует. Если «value» не указать, а ключа «key» в словаре не окажется, то данный метод вернет значение «None».
- setdefault(key[,value]) – этот метод также возвращает значение элемента по ключу «key», но если элемента с заданным ключом не окажется, то в словарь будет добавлен новый элемент со значением «value». Если же значение «value» будет отсутствовать, то метод присвоит добавляемому элементу значение «None» (при этом значение «value» будет добавлено в словарь).
- fromkeys(iterable[, value]) – этот метод создает словарь с ключами из последовательности «iterable» и одинаковыми значениями «value». Если значение «value» не указать, то элементам будет присвоено значение по умолчанию – «None».

Пример программы, в которой показано использование этих методов, приведен в листинге 2.92.

Листинг 2.92. Методы обработки словарей

```
# Листинг 2.92
# Формируем словарь
d_1 = {'a': 1, 'b': 2, 'c': 3}
print('Исходный словарь ->', d_1)
```

```
print('---------- Пробуем метод get --------------')
# Метод get,получаем значение по ключу ['a']
print("Метод get, значение по ключу ['a'] ->", d_1.get('a'))

# Метод get,получаем значение по ключу ['b']
print("Метод get, значение по ключу ['b'] ->", d_1.get('b'))

# Метод get,получаем значение по ключу ['c']
print("Метод get, значение по ключу ['c'] ->", d_1.get('c', 'Такого
ключа нет'))

# Метод get,получаем значение по ключу ['d']
print("Метод get, значение по ключу ['d'] ->", d_1.get('d'))

# Метод get,получаем значение по ключу ['d']
print("Метод get, значение по ключу ['d'] ->", d_1.get('d', 'Такого
ключа нет'))

print('---------- Пробуем метод setdefaul --------------')
print('Исходный словарь ->', d_1)

# Получаем значение d_1['a']
v_1 = d_1.setdefault('a')
print("Метод setdefault, значение по ключу ['a'] ->", v_1)

# Получаем значение d_1['b']
v_1 = d_1.setdefault('b')
print("Метод setdefault, значение по ключу ['b'] ->", v_1)

# Получаем значение d_1['c']
v_1 = d_1.setdefault('c', 'Такого ключа нет')
print("Метод setdefault, значение по ключу ['c'] ->", v_1)

# Добавляем новый элемент с ключом - d и значением 444
v_2 = d_1.setdefault('d', 444)
print("Новый элемент с ключом - d и значением 444 ->", v_2)

# Добавляем новый элемент с ключом - e и значением по умолчанию (None)
v_3 = d_1.setdefault('e')
print("Новый элемент с ключом - e и значением по умолчанию - None ->",
v_3)

# Выводим итоговый словарь
print('---- Итоговый словарь с добавленными элементами ------')
print(d_1)

print('---------- Пробуем метод fromkeys --------------')
print('Создан словарь со значениями по умолчанию - None')
```

```
v_4 = d_1.fromkeys(['Один', 'Два', 'Три', 'Четыре'])
print(v_4)

print('Создан словарь со значениями по умолчанию - 555')
v_4 = d_1.fromkeys(['Один', 'Два', 'Три', 'Четыре'], 555)
print(v_4)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.92. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
Исходный словарь -> {'a': 1, 'b': 2, 'c': 3}
---------- Пробуем метод get --------------
Метод get, значение по ключу ['a'] -> 1
Метод get, значение по ключу ['b'] -> 2
Метод get, значение по ключу ['c'] -> 3
Метод get, значение по ключу ['d'] -> None
Метод get, значение по ключу ['d'] -> Такого ключа нет
---------- Пробуем метод setdefaul --------------
Исходный словарь -> {'a': 1, 'b': 2, 'c': 3}
Метод setdefault, значение по ключу ['a'] -> 1
Метод setdefault, значение по ключу ['b'] -> 2
Метод setdefault, значение по ключу ['c'] -> 3
Новый элемент с ключом - d и значением 444 -> 444
Новый элемент с ключом - e и значением по умолчанию - None -> None
---- Итоговый словарь с добавленными элементами ------
{'a': 1, 'b': 2, 'c': 3, 'd': 444, 'e': None}
---------- Пробуем метод fromkeys --------------
Создан словарь со значениями по умолчанию - None
{'Один': None, 'Два': None, 'Три': None, 'Четыре': None}
Создан словарь со значениями по умолчанию - 555
{'Один': 555, 'Два': 555, 'Три': 555, 'Четыре': 555}
```

Рассмотрим следующую группу методов обработки словарей:

- dict.keys() – возвращает список-представление всех ключей, содержащихся в словаре;
- dict.values() – возвращает список-представление всех значений, содержащихся в словаре;
- dict.items() – возвращает список-представление всех элементов словаря в виде кортежей с парами (key, value), где «key» это ключи словаря, а «value» – значения элементов словаря.

Объекты, возвращаемые методами словарей dict.keys(), dict.values() и dict.items() являются объектами-представлениями. Они обеспечивают динамическое представление записей

словаря. Другими словами, при изменении словаря dict соответствующие представление будет отражать эти изменения. Подробнее о представлениях будет сказано немного позже.

Пример программы, в которой показано использование методов dict.keys(), dict.values() и dict.items(), приведен в листинге 2.93.

Листинг 2.93. Методы обработки словарей

```
# Листинг 2.93
# Формируем словарь.
d_1 = {'a': 1, 'b': 2, 'c': 3}
print('Исходный словарь ->', d_1)
print('-'*30)

# Выведет список ключей
print("Список ключей ->", d_1.keys())
print("Список ключей ->", list(d_1.keys()))
print('-'*30)

# Выведет список значений
print("Список значений ->", d_1.values())
print("Список значений ->", list(d_1.values()))
print('-'*30)

# Выведет список ключей и значений
print("Список ключей и значений ->", d_1.items())
print("Список ключей и значений ->", list(d_1.items()))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.93. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.98).

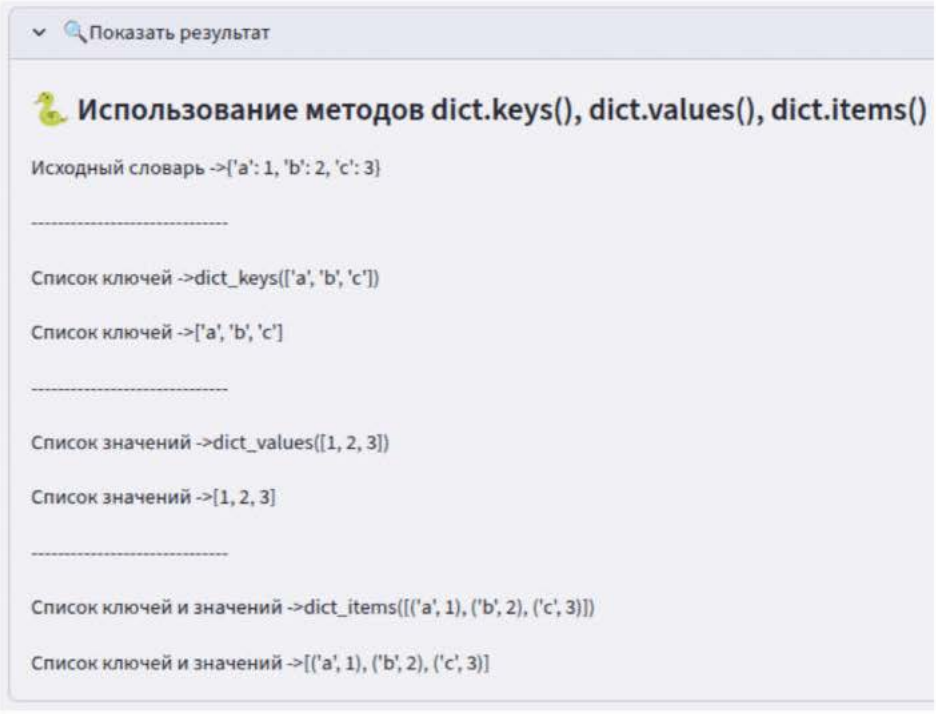


Рис. 2.98. Методы обработки словарей

Следующая группа методов позволяет изменять содержание словарей.

- dict.update([other]) – этот метод добавляет в словарь новые элементы, или обновляет значения уже существующих элементов. Данный метод только меняет элементы словаря и ничего не возвращает. В качестве источника новых ключей и значений выступает аргумент «other». На месте этого аргумента может быть:
    - другой словарь;
    - итерируемый объект с кортежами пар [(key_1, value_1), (key_2, value_2), ...], где первое значение будет ключом будущего элемента, а второе – соответствующим значением;
    - обычным набором пар key_1=value_1, key_2=value_2, ... (здесь, как и при создании словарей, в качестве ключей должны передаваться допустимые имена идентификаторов, которые в кавычки брать не нужно).
- dict.pop(key[, default]) – этот метод вернет значение ключа key, а также удалит его из словаря dict. Если ключ не найден, то вернет значение, заданное аргументом «default».
- dict.popitem() – удаляет из словаря последний добавленный элемент и возвращает его в виде кортежа (key, value). Этот метод удаляет элементы по принципу «последний пришел – первый ушел».

Пример программы, в которой показано использование методов dict.update(), dict.pop() и dict.popitem(), приведен в листинге 2.94.

Листинг 2.94. Методы обработки словарей

```
# Листинг 2.94
# Формируем пустой словарь
d_1 = {}
print("Исходный пустой словарь ->", d_1)

# Заполняем словарь списком пар (ключ:значение)
d_1.update([('a', 1), ('b', 2), ('c', 3)])
print('Заполняем словарь списком пар (ключ:значение) методом update')
print(d_1)
print('-'*30)

# Перечисляем пары через запятую
print("Обновляем словарь значениями  (a=11, b=22, c=33) методом update")
d_1.update(a=11, b=22, c=33)
print(d_1)
print('-'*30)

# Используем другой словарь
print("Обновляем словарь другим словарем {'c': 333, 'd': 444} методом
update")
d_1.update({'c': 333, 'd': 444})
print(d_1)
print('-'*30)

# Удаляем элемент d_1['c']
print("Удаляем из словаря элемент с ключом 'c' методом pop")
print("Удаленный элемент с ключом 'c' ->", d_1.pop('c', 'Нет такого
ключа'))
print("Словарь после удаления элемента с ключом 'c'")
print(d_1)
print('-'*30)

# Попытка удалить элемент с несуществующим ключом d_1['f']
print("Попытка удалить элемент с несуществующим ключом d_1['f'] методом
pop")
print("Удаленный элемент с ключом 'f' ->", d_1.pop('f', 'Нет такого
ключа'))
print(d_1)
print('-'*30)

# Удаляем элемент, вставленные последними
print("Удаляем элемент, вставленный последним")
print("Этот элемент был вставлен последним, он будет удален ->",
d_1.popitem())
print(d_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.94. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
Исходный пустой словарь -> {}
Заполняем словарь списком пар (ключ:значение) методом update
{'a': 1, 'b': 2, 'c': 3}
------------------------------
Обновляем словарь значениями  (a=11, b=22, c=33) методом update
{'a': 11, 'b': 22, 'c': 33}
------------------------------
Обновляем словарь другим словарем {'c': 333, 'd': 444} методом update
{'a': 11, 'b': 22, 'c': 333, 'd': 444}
------------------------------
Удаляем из словаря элемент с ключом 'c' методом pop
Удаленный элемент с ключом 'c' -> 333
Словарь после удаления элемента с ключом 'c'
{'a': 11, 'b': 22, 'd': 444}
------------------------------
Попытка удалить элемент с несуществующим ключом d_1['f'] методом pop
Удаленный элемент с ключом 'f' -> Нет такого ключа
{'a': 11, 'b': 22, 'd': 444}
------------------------------
Удаляем элемент, вставленный последним
Этот элемент был вставлен последним, он будет удален -> ('d', 444)
{'a': 11, 'b': 22}
```

В завершении рассмотрим еще два метода, которые позволяют обрабатывать словари:

- dict.copy() – возвращает поверхностную копию словаря;
- dict.clear() – удаляет все элементы из словаря и возвращает None.

Пример программы, в которой показано использование методов dict.copy() и dict.clear(), приведен в листинге 2.95.

Листинг 2.95. Методы обработки словарей

```
# Листинг 2.95
# Формируем словарь
d_1 = {'a': 1, 'b': [2, 3], 'c': 'Иванов А.В.'}
print("---- Исходный словарь (d_1) ----")
print(d_1)
print('-'*40)

# Создаем копию словаря
d_2 = d_1.copy()
```

```
print("---- Копия словаря (d_2) ----")
print(d_2)
print('-'*40)

# Очищаем словарь (метод вернет None)
print("Очищаем словарь (метод вернет None) ->", d_2.clear())
# Выведет пустой словарь {}.
print("Словарь d2 после очистки ->", d_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.95. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.99).

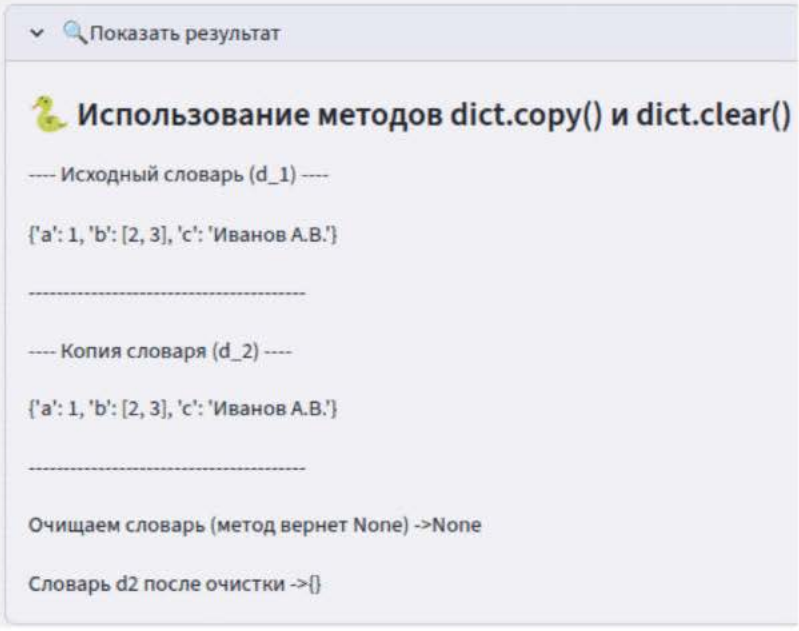


Рис. 2.99. Методы обработки словарей

Теперь рассмотрим некоторые встроенным функциям, которые применимы к словарям:

- len(dict) – возвращает количество элементов в словаре;
- list(dict) – возвращает список ключей словаря;
- iter(dict) – возвращает итератор по ключам словаря (метод является более короткой записью инструкции «iter(dict.keys())»).

В отличие от представления dict.keys(), итератор является статичным набором ключей, который определяется состоянием словаря в момент вызова метода.

Пример программы, в которой показано использование встроенных функций для обработки словарей, приведен в листинге 2.96.

Листинг 2.96. Функции для обработки словарей

```
# Листинг 2.96
# Формируем словарь
d_1 = {'a': 1, 'b': 2, 'c': 3}
print("Исходный словарь ->", d_1)

# Количество элементов в словаре
print("Количество элементов в словаре ->", len(d_1))

# Список ключей словаря
print("Список ключей словаря ->", list(d_1))

# Итератор по ключам словаря
print("Итератор по ключам словаря ->", iter(d_1))
a = iter(d_1)

# Преобразуем итератор словаря в список для вывода на печать
li_iter = list(iter(d_1))
print("Итератор словаря в виде списка -> ->", li_iter)

li_iter = list(iter(d_1.keys()))
print("Итератор словаря в виде списка -> ->", li_iter)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.96. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.100).

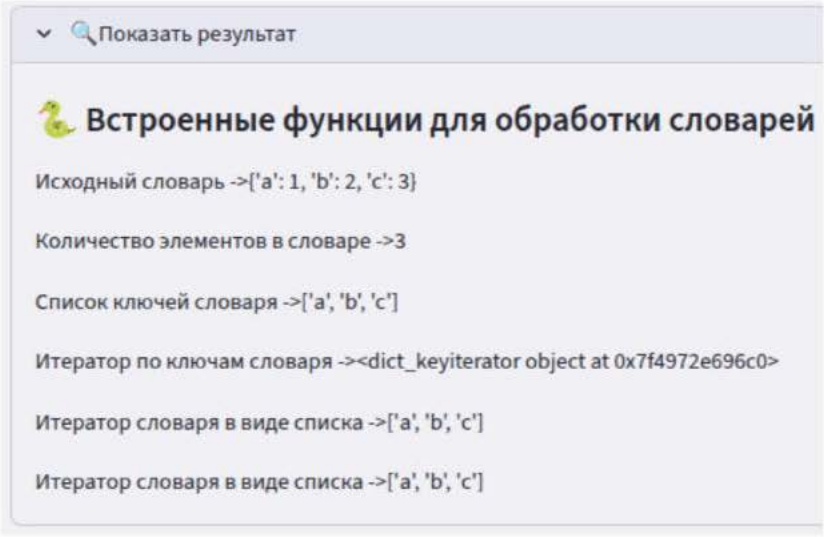


Рис. 2.100. Функции для обработки словарей

2.6.6. Объекты - представления словарей в Python

Объекты, возвращаемые методами словарей dict.keys(), dict.values() и dict.items() являются объектами-представлениями. Они обеспечивают динамическое представление записей словаря. Другими словами, при изменении словаря dict соответствующие представление будет отражать эти изменения.

Представления словарей – это еще одни специальные итерируемые объекты, которые обеспечивают динамическое представление записей словаря. Это значит, что при изменении какого либо элемента словаря (например, при удалении элемента) соответствующие представления то же будет изменяться.

Пример программы, в которой показано использование объектов-представлений, приведен в листинге 2.97.

Листинг 2.97. Использование объектов-представлений для словарей

```
# Листинг 2.97
# Формируем исходный словарь
d_1 = {'a': 1, 'b': 2, 'c': 3}
# Выводим его на экран.
print('Исходный словарь ->:', d_1)
print('-'*60)

# Сохраняем представления словаря в переменных
keys = d_1.keys()
values = d_1.values()
i = d_1.items()
# Для примера получим один итератор
```

```
i_keys = list(iter(d_1.keys()))

# Выводим представления исходного словаря
print('Представления исходного словаря')
print('Представления ключей ->', keys)
print('Итератор ключей ->', i_keys)
print('Значения элементов словаря ->', values)
print('Все элементы словаря ->', i)
print('-'*60)

# Изменим словарь
del d_1['a']                        # удаляем элемент из словаря
d_1.update(b=222, c=333)            # обновляем элементы словаря
d_1['d'] = "Новый элемент"          # добавляем элемент в словарь

# Выводим все на экран
print('Представления после изменения словаря')
print('Представления ключей ->', keys)
print('Итератор ключей ->', i_keys)
print('Значения элементов словаря ->', values)
print('Все элементы словаря ->', i)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.97. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.101).

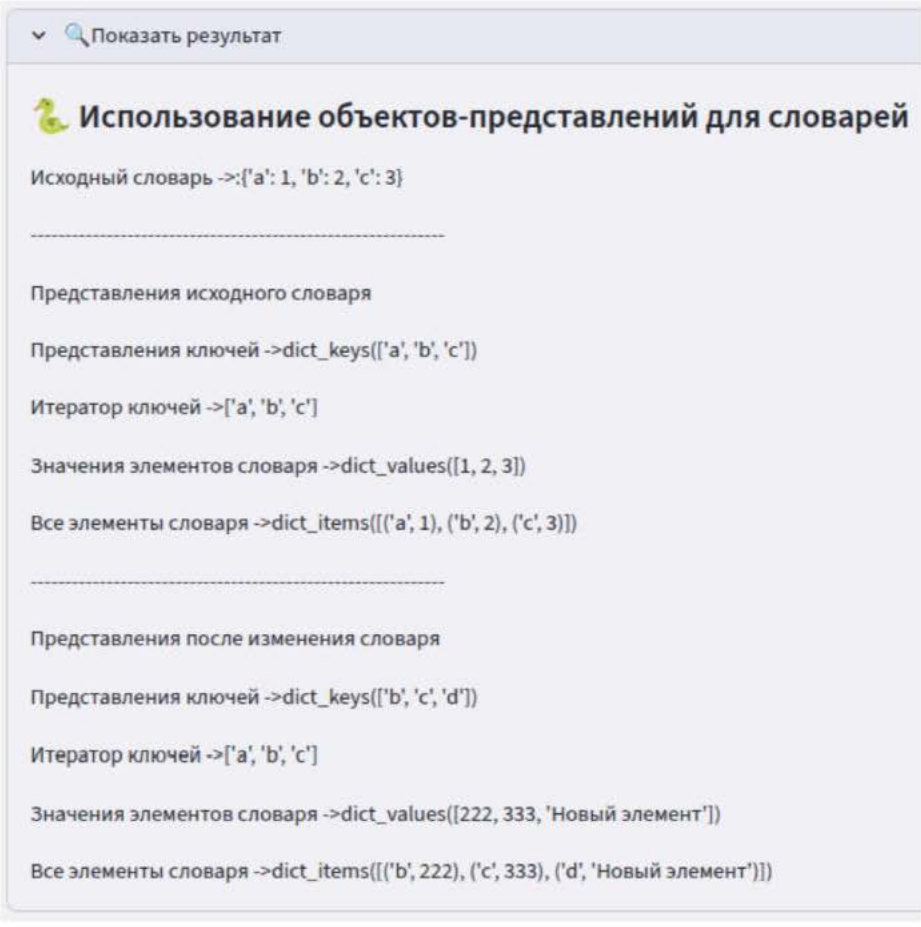


Рис. 2.101. Использование объектов-представлений для словарей

Как видно из данного примера, при изменении словаря его представления автоматически изменились. А вот итератор (преобразованный для наглядности в список) остался прежним.

Если по какой-то причине вместо представления тех же ключей словаря понадобится их список или кортеж, то представление можно преобразовать с помощью соответствующего конструктора, например, list() или tuple(). Точно также, с использованием встроенной функции iter(), можно преобразовать представление в итератор.

Добавим, что для представлений также доступен ряд известных нам операций: функция len(), функция iter(), инструкция проверки на вхождение «in», а также обход в цикле с использованием инструкции «for».

Пример программы, в которой показано использование операций обработки объектов-представлений, приведен в листинге 2.98.

Листинг 2.98. Использование операций обработки объектов-представлений для словарей

```
# Листинг 2.98
# Формируем словарь.
d_1 = {'a': 1, 'b': 2, 'c': 3}
print('Исходный словарь ->:', d_1)
print('-'*45)

# Получаем представление ключей словаря
keys = d_1.keys()
print("Количество элементов в словаре ->", len(keys))

# Проверяем наличие ключей в словаре
key_a = 'a' in keys
key_d = 'd' in keys
print("Наличие ключа a в словаре ->", key_a)
print("Наличие ключа d в словаре ->", key_d)
print('-'*35)

print("Перечень ключей в словаре")
for key in keys:
    print('Ключ словаря ->', key)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.98. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.102).

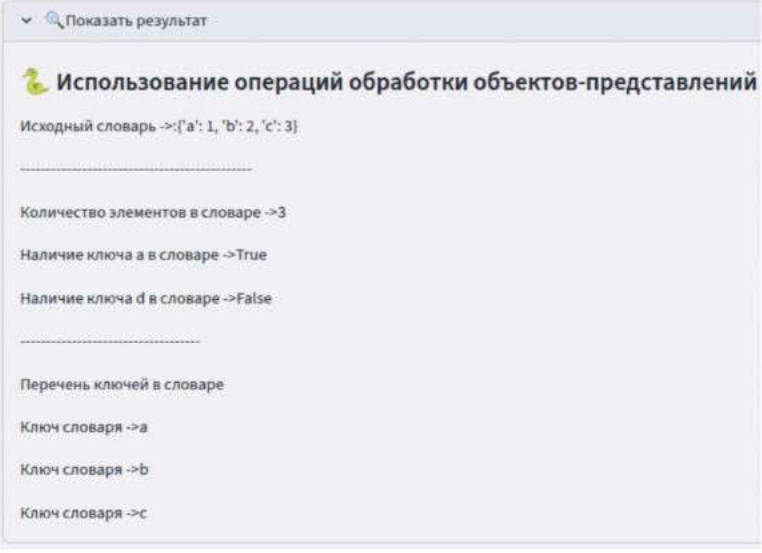


Рис. 2.102. Использование операций обработки объектов-представлений для словарей

Поскольку представления не являются списками, они не поддерживают такие операции, как обращение к элементам по индексам или метод «sort» списков, а также не отображают значения своих элементов при выводе.

Представления в некоторой степени похожи на множества – «set», которые будут рассмотрены в следующем разделе (множество – это коллекции уникальных и неизменяемых объектов). В Python для представления ключей dict.keys() и представления элементов dict.items() стали доступны операции, определенные для множеств.

Пример программы, использование операций обработки множеств (пересечение, объединение) для объектов-представлений, приведен в листинге 2.99.

Листинг 2.99. Использование операций обработки множеств для объектов-представлений

```
# Листинг 2.99
# Формируем два словаря
d_1 = {'a': 1, 'b': 2, 'c': 3}
d_2 = {'b': 2, 'c': 3, 'd': 4}
print('----- Исходные словари -----')
print('d_1 ->', d_1)
print('d_2 ->', d_2)
print('-' * 35)

# Получаем представления ключей словарей
k_1 = d_1.keys()
k_2 = d_2.keys()

# Находим их пересечение (общие ключи)
k = k_1 & k_2
print("Пересечение (общие ключи словарей) ->", k)

# Объединяем представления словарей
k = k_1 | k_2
print("Объединяем представления словарей ->", k)

# Объединяем результат со множеством
m = {'e', 'f'}  # Создаем множество
print("Множество ->", m)

k = k | m  # Объединяем представления словарей с множеством
print("Представления словарей + множество ->", k)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.99. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.103).

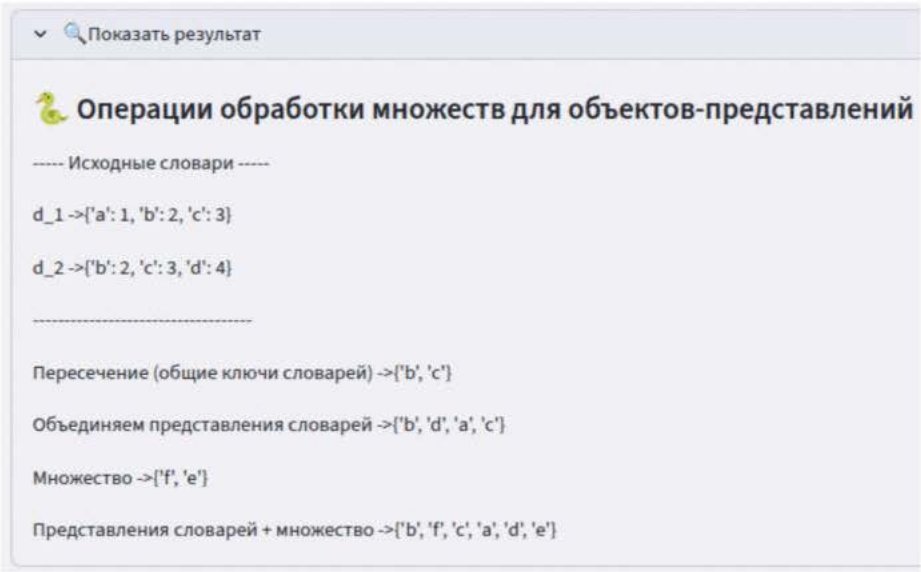


Рис. 2.103. Использование операций обработки множеств для объектов-представлений

Что касается аналогичных операций в отношении представлений элементов словарей dict.items(), то здесь важно понимать, что они должны быть неизменяемыми, иначе интерпретатор выдаст ошибку, это демонстрирует следующий программный код (листинг 2.100).

Листинг 2.100. Использование метода dict.items()

```
# Листинг 2.100
# Формируем словари
d_1 = {'a': 1, 'b': 2}
d_2 = {'b': 2, 'c': 3}
d_3 = {'a': [1, 2], 'b': 2}  # Здесь есть изменяемый элемент

# Получаем представления элементов
itm_1 = d_1.items()
itm_2 = d_2.items()
itm_3 = d_3.items()

itm = itm_1 & itm_2  # Находим пересечение (тут все нормально)
print("Результат инструкции (itm_1 & itm_2) ->", itm)
print("Тип созданного объекта type(itm) ->", type(itm))  # Узнаем тип
результата

# Та же операция с изменяемым элементом d_3[0]
itm = itm_1 & itm_3  # Получим ошибку, та как в d_3 присутсвует
изменяемый элемент
print("itm_1 & itm_3 ->", itm)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.100. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.104).

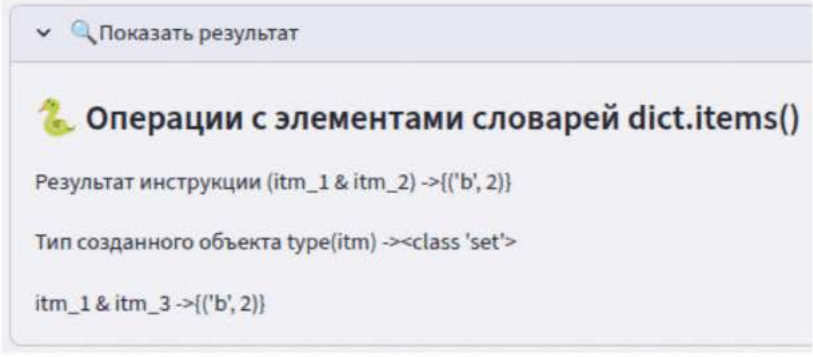


Рис. 2.104. Использование метода dict.items()

Здесь словарь d_3 в качестве одного из элементов содержит список ([1, 2]), а списки относятся к изменяемым данным. Поэтому при попытке выполнения операции itm = itm_1 & itm_3 программа выдала ошибку.

2.6.7. Краткие итоги раздела 2.6

1. В Python словари представляют собой неупорядоченные изменяемые коллекции объектов произвольного типа с доступом по ключу.
2. Чтобы создать словарь, нужно в фигурных скобках через запятую перечислить пары «ключ: значение», в которых ключи отделить от значений двоеточием. В случае необходимости последний элемент словаря можно завершить запятой. Это никак не повлияет на количество элементов словаря, а значит и его длину.
3. Словари могут быть созданы при помощи конструктора dict(). Этот конструктор можно использовать со следующими видами аргументов:
   - если аргументы отсутствуют, то конструктор вернет пустой словарь {};
   - если передать конструктору через запятую пары key_name_1 = val_1, key_name_2 = val_2, ..., то на выходе получим словарь с целевыми элементами {'key_name_1': val_1, 'key_name_2': val_2, ...}, в котором ключами будут строки с переданными именами, то есть допустимыми в Python идентификаторами.;
   - если конструктору передавать список с кортежами пар значений [(key_1, val_1), (key_2, val_2), ...], первое из которых станет ключом, а второе элементом словаря (например, инструкции dict(one = 1, two = 2) и dict([('one', 1), ('two', 2)]) вернут словарь {'one': 1, 'two': 2}).

4. Для создания словарей в Python могут использоваться генераторы словарей, которые применяют выражение к каждому элементу передаваемых последовательностей и на основе их вычисления возвращают готовые словари с элементами. Например, генератор {x: y for x, y in [('one', 1), ('two', 2)]} вернет нам словарь {'one': 1, 'two': 2}.
5. Для словарей доступен специальный метод класса dict.fromkeys(iterable[, value]), который возвращает словарь, использующий в качестве своих ключей элементы итерируемого объекта iterable, а в качестве их значений второй необязательный параметр value, который по умолчанию принимается за None. Например, инструкция dict.fromkeys('abcd', 0) вернет словарь {'a': 0, 'b': 0, 'c': 0, 'd': 0}.
6. Доступ к элементам словаря можно получать при помощи ключей, которые нужно указывать в квадратных скобках после имени словаря. Например, для словаря d = {'one': 1, 'two': 2} инструкция d['two'] вернет 2.
7. Словари могут содержать объекты любого типа и любого уровня вложенности, поэтому цепочка доступа к конкретному элементу словаря может быть довольно длинной. Такая цепочка может включать не только квадратные скобки с ключами для доступа к элементам словарей, но и квадратные скобки с индексами для доступа к элементам списков, строк, кортежей или даже точки для доступа к атрибутам объектов. Например, для словаря d = {'a': [{'one': 1}, {'two': 2}], 'b': 2} инструкция d['a'][0]['one'] вернет 1.
8. В качестве ключей словаря могут использоваться только объекты неизменяемых типов данных, так как они должны однозначно идентифицировать элементы словаря. Сюда относятся числа, строки, кортежи с неизменяемыми элементами, фиксированные множества, None, логические значения True и False, а также другие неизменяемые типы данных.
9. В словаре нельзя использовать один и тот же ключ для нескольких элементов. Ошибки интерпретатор не выдаст, но доступен будет только один из таких элементов. В то же время один и тот же объект вполне может быть доступен по разным ключам. Кроме того, не стоит использовать в качестве ключей одновременно 0 и False (интерпретатор считает это значение нулем), а также 1 и True (интерпретатор считает его единицей).
10. Поскольку словари являются изменяемыми коллекциями объектов, их можно изменять: удалять элементы, добавлять новые, изменять значения уже существующих элементов.
11. К словарям применимы инструкции: «in» (проверяет наличие в словаре объекта с указанным ключом) и цикл «for» (позволяет перебрать все элементы словаря по их ключам).
12. Словари поддерживают ряд методов, которые дают возможность выполнять широкий спектр операций над словарями. При этом для их использования не нужно импортировать никаких модулей, так как они уже встроены в сам интерпретатор. Однако не стоит забывать, что словари изменяемы, поэтому их методы изменяют сам словарь, а не возвращают его измененную копию.
13. У словарей есть специальные объекты - представления, которые обеспечивают динамическое представление записей словаря. При изменении словаря соответствующее представление будет изменяться. Например, получив один раз представление ключей словаря, мы в дальнейшем можем обращаться к представлению за актуальной информацией о состоянии ключей (даже после добавления в словарь новых или удаления существующих ключей).

2.6.8. Вопросы для самоконтроля к разделу 2.6



1.К какому типу данных относятся словари:

- неупорядоченные, изменяемые;
- упорядоченные, изменяемые;
- упорядоченные, неизменяемые;
- неупорядоченные, неизменяемые.

2.Какие из литералов относятся к словарю:

5. {1: 'один', 2: 'два'};
6. 5.0;
7. [3, 4];
8. "{'a': 1, 'b': 2}".

3.Дан словарь d = {'a': [{'1': 'Иванов'}, 'Петров'], 'b': 3}. Какое значение будет выдано при значении ключа d['b'] = ?:

- 3;
- Иванов;
- Петров;
- 1;

4.Дан словарь d = {'a': [{'1': 'Иванов'}, 'Петров'], 'b': 3}. Какое значение будет выдано при значении ключа d['a'] = ?:

- [{'1': 'Иванов'}, 'Петров'];
- Иванов;
- Петров;
- 3.

5.Дан словарь d = {'a': [{'1': 'Иванов'}, 'Петров'], 'b': 3}. Какое значение будет выдано при значении ключа d['a'][1] = ?:

- Петров;
- [{'1': 'Иванов'}, 'Петров'];
- Иванов;
- 3.

6.Дан словарь d = {'a': [{'1': 'Иванов'}, 'Петров'], 'b': 3}. Какое значение будет выдано при значении ключа d['a'][0] = ?:

- {'1': 'Иванов'};
- [{'1': 'Иванов'}, 'Петров'];
- Петров;
- 3.

7.Сколько элементов в словаре d = {1: 'a', 2: ['ab', {'a': 1, 'b': 2}], (1, 2): ('a', 'b')}:

- 3;
- 2;
- 1;
- 5.

8.Что из этого ключ словаря d = {1: 'a', 2: ['ab', {'a': 1, 'b': 2}], (1, 2): ('a', 'b')}:

- (1, 2);
- 'a';
- 'a': 1;
- 'b': 2.

9.Что из этого значение ключа словаря d = {1: 'a', 2: ['ab', {'a': 1, 'b': 2}], (1, 2): ('a', 'b')}:

- ('a', 'b');
- 1;
- 2.

2.7. Множества в Python

Множества (от англ. set) – это неупорядоченные коллекции уникальных и неизменяемых объектов, называемых элементами множества.

Как следует из определения, множества не являются последовательностями, как списки или кортежи, и в тоже время не являются отображениями, как словари. Это просто коллекции уникальных и неизменяемых объектов, во многом похожие на математические множества. Множества можно объединять, искать пересечение, находить разность или выполнять над ними какие-нибудь другие типичные математические операции. Множество можно представить в виде некого облака, в котором хаотично расположились совершенно разные и непохожие друг на друга данные (рис. 2.105).



Рис. 2.105. Образное представление понятия «множества»

2.7.1.Создание множеств в Python

Для того чтобы создать множество, нужно просто перечислить элементы множества через запятую внутри фигурных скобок. При этом последний элемент множества разрешается завершать запятой, поскольку это никак не повлияет на количество его элементов.

Пример программы с примерами создания множества приведен в листинге 2.101.

Листинг 2.101. Создание множества

```
# Листинг 2.101
# Множество из одного элемента
s_1 = {1}
print('Множество из одного элемента ->', s_1)

# Множество из 4-х элементов
s_2 = {'Иванов', 0.1, ('Иванов', 2), 567, }
print('Множество из 4-х элементов ->', s_2)

# Множество с одинаковыми значениями элементоа
s_3 = {'a', 1, 1, 1, 1}
print('Слияние одинаковых элементов множества ->', s_3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.101. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.106).

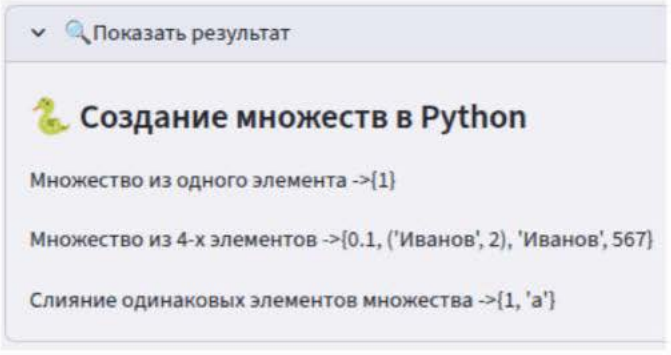


Рис. 2.106. Создание множества

Как видно из данного примера, множества действительно состоят из наборов уникальных значений. Попытка создания множества с несколькими одинаковыми элементами ошибки не вызовет, но интерпретатор преобразует множество так, что в нем останется только один из одинаковых элементов. Также при помощи фигурных скобок не получится создать пустое множество, ведь конструкция {} зарезервирована под пустой словарь. Если попытаться добавить в множество изменяемый элемент, интерпретатор выдаст ошибку. Например, следующее множество s_3 = {'a', 4, [1, 2, 3]} не будет создано, так как оно содержит список, а списки относятся к изменяемым типам данных.

Создать пустое множество можно и при помощи встроенного конструктора set([iterable]), который в качестве аргумента может принимать итерируемый объект, например, строку или список. Если использовать конструктор без аргумента, то он вернет пустое множество. Если же использовать конструктор с аргументом, он вернет нам множество с элементами переданного итерируемого объекта.

Пример программы с примерами создания множеств с использованием конструктора set() приведен в листинге 2.102.

Листинг 2.102. Создание множества с использованием конструктора set()

```
# Листинг 2.102
# Пустое множество
s_0 = set()
print('Пустое множество ->', s_0)

# Множество из списка
s_1 = set([1, 2, 3])
```

```
print('Множество из списка ->', s_1)

# Множество из кортежа
s_3 = set(('a', 1, 2))
print('Множество из кортежа ->', s_3)

# Множество из вложенных кортежей
s_4 = set(('a', 0.1, ('b', 2)))
print('Множество из вложенных кортежей ->', s_4)

# Множество из строки символов
s_5 = set('Строка')
print('Множество из строки символов ->', s_5)

# Копия множества
s_6 = set(s_5)
print('Копия множества s_6 из s_5 ->', s_6)

# Множество из одного элемента
s_7 = set((333,))
print('Множество из одного элемента ->', s_7)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.102. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.107).

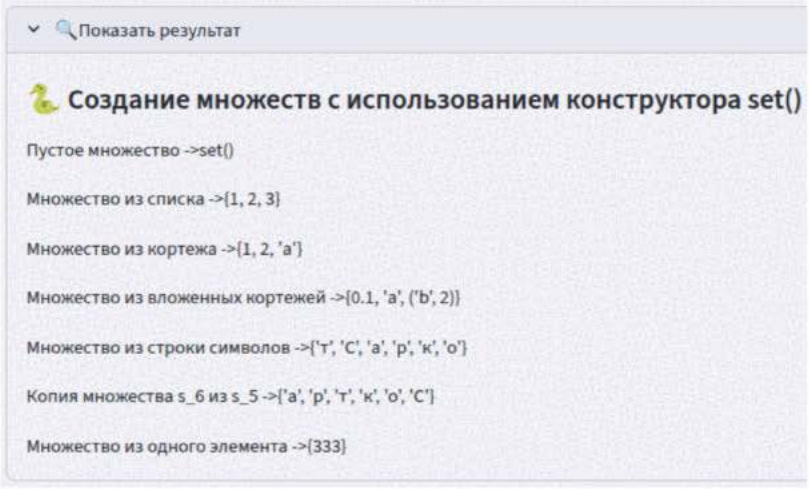


Рис. 2.107. Создание множества с использованием конструктора set()

Обратите внимание, что если передать конструктору другое множество, он вернет его поверхностную копию. Также имейте в виду, что при передаче конструктору множества из одного элемента, например числа, то нужно использовать завершающую запятую.

Для создания множеств в можно использовать и генераторы множеств, которые применяют выражение к каждому элементу передаваемого итерируемого объекта и на основе их вычисления возвращают новое множество.

Пример программы использования генератора для создания множества приведен в листинге 2.103.

Листинг 2.103. Генератор для создания множества

```
# Листинг 2.103
# Генерируем множество из символьной строки
s_1 = {sim_str for sim_str in '123'}
print('Множество из символьной строки ->', s_1)

# Генерируем множество из арифметического выражения
s_2 = {int(num)*5 for num in '123'}
print('Множество из арифметического выражения ->', s_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.103. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.108).

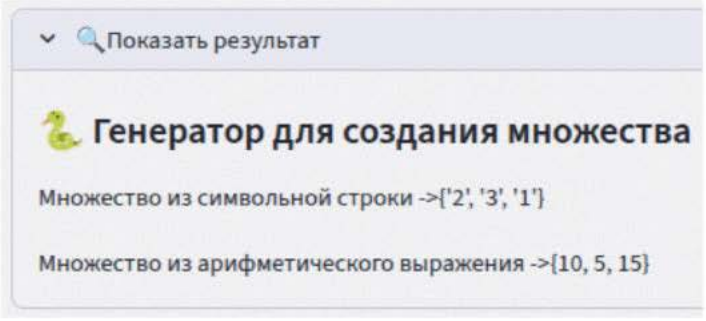


Рис. 2.108. Генератор для создания множества

Таким образом, в простейшем случае для генерации множества нужно в фигурных скобках написать выражение, которое будет применяться к каждому элементу передаваемого итерируемого объекта, и через пробел прописать соответствующий заголовок цикла «for» без завершающего двоеточия.

При использовании для создания множеств любого из перечисленных методов важно помнить, что будущие элементы множества должны быть неизменяемыми. Можно,

например, использовать числа, строки, логические значения True и False, объект None или кортеж, состоящий из неизменяемых элементов.

2.7.2.Операторы и инструкции для обработки множеств

Поскольку множества, как и словари, относятся к неупорядоченным коллекциям объектов, операторы «+», «+=», «*» и «*=» с ними не используются. Однако множествам доступен целый ряд других математических операторов, которые можно разбить на две группы: для сравнения множеств, для обработки множеств.

В Python используются следующие операторы сравнения: «==», «<», «>», «<=», «>=». С помощью этих операторов можно проверить, является ли набор подмножеством, надмножеством или эквивалентом другого набора. Эти операторы возвращают следующие значения:

- s_1 == s_2 – возвращает True, если множество s_1 является подмножеством s_2 и при этом они равны;
- s_1 < s_2 – возвращает True, если множество s_1 является подмножеством s_2 и при этом они не равны;
- s_1 > s_2 – возвращает True, если множество s_1 является надмножеством s_2 и при этом они не равны;
- s_1 >= s_2 – возвращает True, если множество s_1 является надмножеством s_2, т.е. каждый элемент второго множества есть в первом множестве;
- s_1 <= s_2 – возвращает True, если множество s_1 является подмножеством s_2, т.е. каждый элемент первого множества есть во втором множестве.

Пример программы использования операторов сравнения для множества приведен в листинге 2.104.

Листинг 2.104. Использования операторов сравнения для множества

```
# Листинг 2.104
# Исходные множества
Days1 = {"Понедельник",  "Вторник", "Среда", "Четверг"}
Days2 = {"Понедельник",  "Вторник"}
Days3 = {"Понедельник", "Вторник", "Пятница"}
Days4 = {"Вторник", "Понедельник"}
print('    Исходные множества')
print('Days1 - >', Days1)
print('Days2 - >', Days2)
print('Days3 - >', Days3)
print('Days4 - >', Days4)
print('-'*30)

# Результаты сравнения
print('    Результаты сравнения')
print('Days2 == Days4 ->', Days2 == Days4)
print('Days1 == Days2 ->', Days1 == Days2)
print('Days1 > Days2 ->', Days1 > Days2)
```

```
print('Days1 < Days2 ->', Days1 < Days2)
print('Days1 >= Days2 ->', Days1 >= Days2)
print('Days1 <= Days2 ->', Days1 <= Days2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.104. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.109).



Рис. 2.109. Использования операторов сравнения для множества

Обработка множеств может выполняться с использованием следующих операторов:

- объединение двух множеств - «|»;
- пересечение двух множеств - «&»;
- разница двух множеств - «-»;

- симметричная разность двух множеств - «^».

Объединение двух множеств вычисляется с помощью оператора в виде вертикальной черты «|». Объединенный набор будет содержать все элементы, которые присутствуют в обоих наборах. В результате операции «U=Days_1 | Days_2» – будет получено новое множество «U», которое объединит исходные множеств. Новое множество будет включать все элементы, которые есть хотя бы в одном из множеств, участвующих в операции (рис. 2.110).

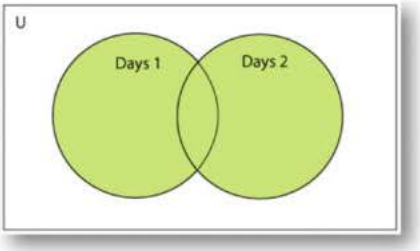


Рис. 2.110. Объединение двух множеств (оператор «|»)

Пересечение двух множеств может быть выполнено с помощью оператора «&». Пересечение двух множеств это набор элементов, которые содержаться в обоих множествах. В результате операции «U=Days_1 & Days_2» – будет получено новое множество «U», образуемое через пересечение множеств. Такое множество будет состоять только из тех элементов, которые есть в обоих множествах, участвующих в операции (рис. 2.111).

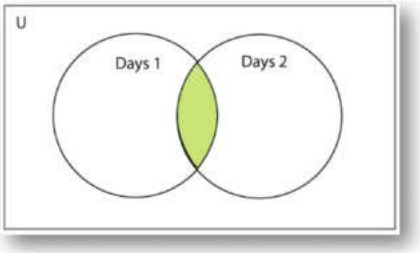


Рис. 2.111. Пересечение двух множеств (оператор «&»)

Разницу двух множеств можно вычислить с помощью оператора вычитания «-». В результате операции «U=Days_1 - Days_2» во множество «U» будут включены те элементы из Days_1, которых нет во множестве Days_2. Иными словами новое множество будет состоять из всех элементов, которые есть в уменьшаемом множестве (левый операнд), но нет в вычитаемом множестве (правый операнд) – рис. 2.112.

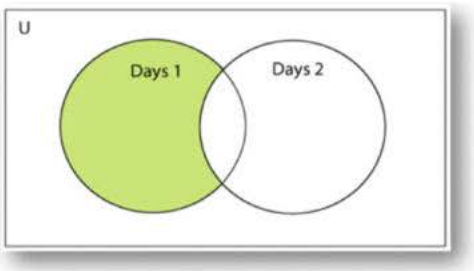


Рис. 2.112. Разница двух множеств (оператор «-»)

Симметричная разность двух множеств вычисляется с помощью оператора «^». Симметричная разность множеств удаляет те элементы, которые присутствуют в обоих множествах. В результате операции «U=Days_1 ^ Days_2» во множество «U» будут включены все элементы обоих множеств, но при этом удалены те элементы, которые есть и во множестве Days_1, и во множестве Days_2 (рис. 2.113).

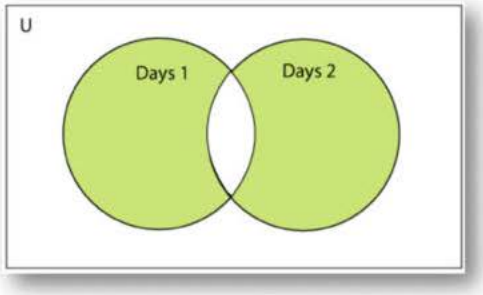


Рис. 2.113. Симметричная разность двух множеств (оператор «^»)

Описанные выше действия могут быть выполнены и с использованием составных операторов:

- s_1 |= s_2 – обновляет множество s_1, добавляя элементы из множества s_2;
- s_1 &= s_2 – обновляет множество s_1, оставляя в нем только элементы, которые есть сразу в обоих множествах, участвующих в операции;
- s_1 -= s_2 – обновляет множество s_1, оставляя в нем только элементы, которых нет во множестве s_2;
- s_1 ^= s_2 – обновляет множество s_1, оставляя в нем только элементы, которые есть в одном из множеств, участвующих в операции, но не в обоих сразу.

Пример программы обработки множеств приведен в листинге 2.105.

Листинг 2.105. Пример программы обработки множеств

```
# Листинг 2.105
# Исходные множества
Days1 = {"Понедельник", "Вторник", "Среда", "Четверг"}
Days2 = {"Пятница", "Суббота", "Воскресенье"}
Days3 = {"Вторник", "Среда"}
```

```
print('    Исходные множества')
print('Days1 - >', Days1)
print('Days2 - >', Days2)
print('Days3 - >', Days3)

print('----- Объединение множеств Days1 | (Days2   ----------')
print(Days1 | Days2)
print(' ')

print('----- Пересечение множеств Days1 & (Days3   ----------')
print(Days1 & Days3)
print(' ')

print('----- Разница множеств Days1 - Days3   ----------')
print(Days1 - Days3)
print(' ')

print('----- Симметричная разность двух множеств  Days1 ^ (Days3   -----
-----')
print(Days1 ^ Days3)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.105. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.114).

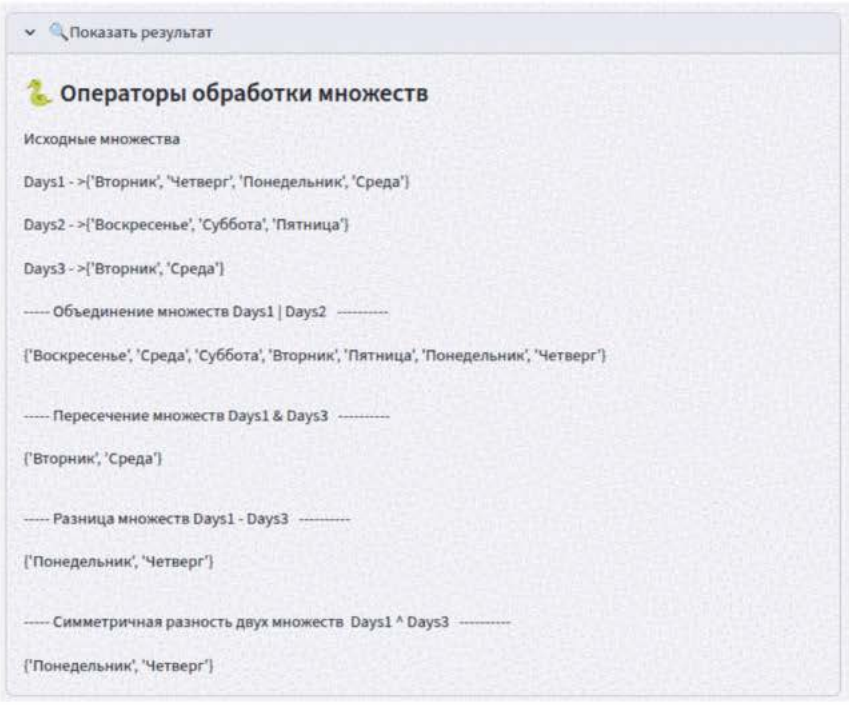


Рис. 2.114. Пример программы обработки множеств

Следует помнить, что математические операторы «>», «>=», «<» и «<=» не сравнивают множества по количеству элементов, а проверяют на вхождение одного множества в другое. Поэтому, например, операция {1, 2, 3} > {4, 5} вернет False, хотя элементов в первом множестве больше.

Что касается инструкции проверки на вхождение «in», то применительно к множествам она совершает поиск переданного объекта (левый операнд) среди элементов целевого множества (правый операнд). В случае успеха операция возвращает «True».

Пример программы использования инструкции «in» (проверки на вхождение), приведен в листинге 2.106.

Листинг 2.106. Использование инструкции «in» для обработки множеств

```
# Листинг 2.106
# Создаем множество и сохраняем его в переменной
s_1 = {1, 2, 3, 4, 5}
print('Исходное множество ->', s_1)
print('Есть ли 2 в множестве ->', 2 in s_1)
print('Есть ли 10 в множестве ->', 10 in s_1)
print("Отсутсвует ли 10 в множестве ->", 10 not in s_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел

«Листинг 2.106. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.115).

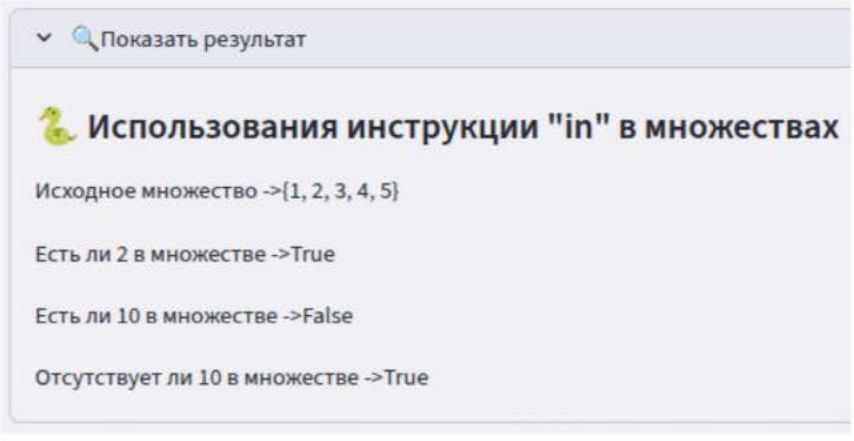


Рис. 2.115. Использование инструкции «in» для обработки множеств

С множествами можно использовать инструкцию цикла «for», которая позволяет сделать перебор элементов множества.

Пример программы использования инструкции «for» приведен в листинге 2.107.

Листинг 2.107. Использование инструкции «for» для обработки множеств

```
# Листинг 2.107
# Создаем множество в переменной
s_1 = {1, 2, 0.3, 'Иванов'}
print('Исходное множество ->', s_1)

# Цикл по значениям множества
for val in s_1:
    print('Элемент множества ->', val)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.107. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.116).

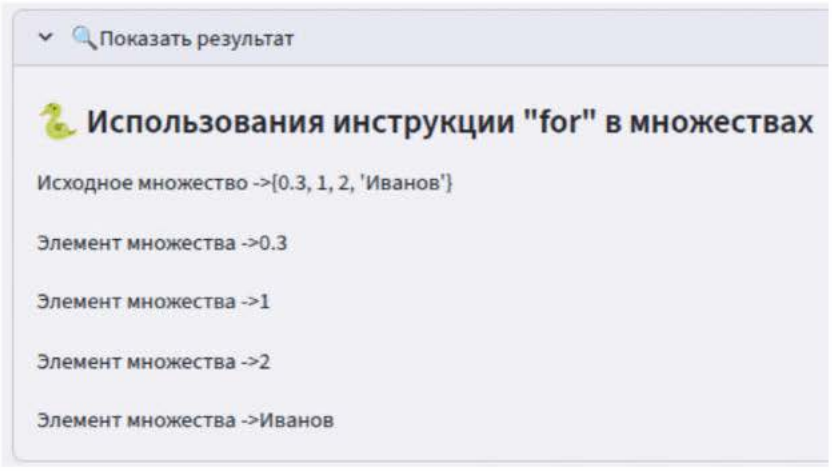


Рис. 2.116. Использование инструкции «for» для обработки множеств

2.7.3. Функции и методы множеств в Python

Множества имеют встроенные методы и функции, при использовании которых учитывать, что это изменяемые объекты, поэтому многие методы меняют содержимое множеств, а не возвращают их измененную копию.

Многие методы дублируют действия соответствующих математических операторов, перечисленных выше:

- set.isdisjoint(other) – метод возвращает True, если у множества set нет общих элементов с указанным множеством other;
- set.issubset(other) – метод делает тоже, что и оператор «set <= othe»;
- set.issuperset(other) – метод делает тоже, что и оператор «set >= other»;
- set.union(*others) – метод делает тоже, что и оператор «set | other | ....»;
- set.intersection(*others) – метод делает тоже, что и оператор «set & other & ....»;
- set.difference(*others) – метод делает тоже, что и оператор «set - other - ....»;
- set.symmetric_difference(other) – метод делает тоже, что и оператор «set ^ other».

Пример программы использования методов, описанных выше, приведен в листинге 2.108.

Листинг 2.108. Использование методов для обработки множеств

```
# Листинг 2.108
# Сохраняем множества в переменных
s_1 = {"Понедельник",  "Вторник", "Среда"}
s_2 = {"Вторник", "Среда", "Четверг"}
s_3 = {"Четверг", "Пятница", "Суббота"}

print('    Исходные множества')
print('Days1 - >', s_1)
```

```
print('Days2 - >', s_2)
print('Days3 - >', s_3)


print('     Метод issubset()')
res = s_1.issubset(s_2)
print('s_1.issubset(s_2) ->', res)
print('     Метод isdisjoint()')
res = s_1.isdisjoint(s_2)
print('s_1.isdisjoint(s_2) ->', res)
res = s_1.isdisjoint(s_3)
print('s_1.isdisjoint(s_3) ->', res)

# Методы получения новых множеств

print('      Объединение: (s_1, s_2, s_3) методом union()')
res = set.union(s_1, s_2, s_3)
print(res)
print('-'*65)

print('      Пересечение (s_2, s_3) методом intersection()')
res = s_1.intersection(s_2, s_3)
print(res)
print('-'*65)

print('      Разность (s_1, s_2, s_3) методом difference()')
res = set.difference(s_1, s_2, s_3)
print('set.difference(s_1, s_2, s_3) ->', res)
print('-'*65)

print('      Разность (s_1, s_2 методом symmetric_difference()')
res = s_1.symmetric_difference(s_2)
print(res)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.108. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
   Исходные множества
Days1 - > {'Среда', 'Понедельник', 'Вторник'}
Days2 - > {'Четверг', 'Среда', 'Вторник'}
Days3 - > {'Четверг', 'Пятница', 'Суббота'}
    Метод issubset()
s_1.issubset(s_2) -> False
```

```
    Метод isdisjoint()
s_1.isdisjoint(s_2) -> False
s_1.isdisjoint(s_3) -> True
    Объединение: (s_1, s_2, s_3) методом union()
{'Понедельник', 'Вторник', 'Четверг', 'Пятница', 'Суббота', 'Среда'}
----------------------------------------------------------------
    Пересечение (s_2, s_3) методом intersection()
set()
----------------------------------------------------------------
    Разность (s_1, s_2, s_3) методом difference()
set.difference(s_1, s_2, s_3) -> {'Понедельник'}
----------------------------------------------------------------
    Разность (s_1, s_2 методом symmetric_difference()
{'Четверг', 'Понедельник'}
```

Обратите внимание, что вышеперечисленные методы не изменяют исходные множества. Кроме того, как видно из примера, многие методы можно использовать как в форме общего метода класса, так и в форме метода конкретного объекта. Например, инструкции set.union(s_1, s_2, s_3) и s_1.union(s_2, s_3) в обоих случаях вернули множество:

```
{'Суббота', 'Пятница', 'Среда', 'Вторник', 'Четверг', 'Понедельник'}
```

Методы, которые меняют сами множества, должны использоваться в форме метода конкретного объекта. Они обновляют само множество. К этим методам относятся:

1. set.update(*others) – метод делает тоже, что и «set |= other | ....»
2. set.intersection_update(*others) – метод делает тоже, что и «set &= other & ....»
3. set.difference_update(*others) – метод делает тоже, что и «set -= other | ....»
4. set.symmetric_difference_update(other) – метод делает тоже, что и «set ^= other».

Пример программы использования методов, описанных выше, приведен в листинге 2.109.

Листинг 2.109. Использование методов для обработки множеств

```
Листинг 2.109
# Сохраняем множества в переменных
s_1 = {1, 2}
s_2 = {2, 3}
s_3 = {3, 4}

print('    Исходные множества')
print('s_1 - >', s_1)
print('s_2 - >', s_2)
print('s_3 - >', s_3)
print('-'*30)

# Здесь изменяется множество s_1.
s_1.update(s_2, s_3)
print('Итог: s_1.update(s_2, s_3) ->', s_1)
print('-'*30)
```

```
s_1.intersection_update(s_2, s_3)
print('Итог: s_1.intersection_update(s_2, s_3) ->', s_1)
print('-'*30)

s_1.difference_update(s_2, s_3)
print('Итог: s_1.difference(s_2, s_3) ->', s_1)
print('-'*30)

s_1.symmetric_difference_update(s_2)
print('Итог: s_1.symmetric_difference_update(s_2) ->', s_1)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.109. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.117).

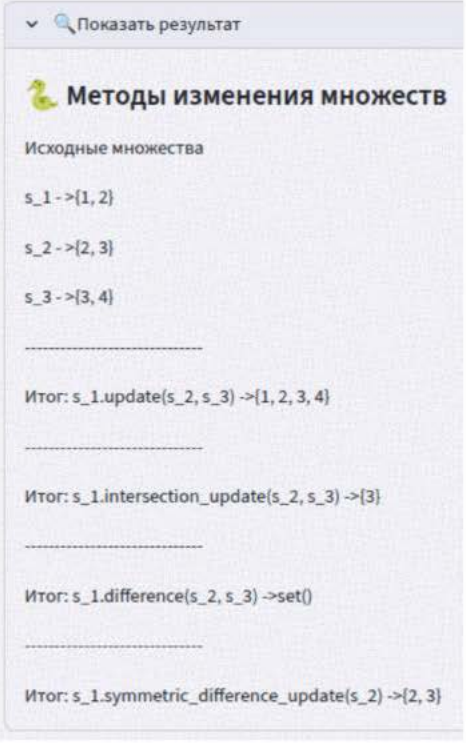


Рис. 2.117. Использование методов для обработки множеств

Следующая группа методов позволяет изменить само множество (добавить элемент, удалить элемент, скопировать или очистить множество). К этим методам относятся:

- set.add(elem) – добавляет во множество указанный элемент elem;
- set.discard(elem) – удаляет указанный элемент elem, если он присутствует в множестве;
- set.remove(elem) – также удаляет указанный элемент elem, если он присутствует в множестве.
- set.pop() – удаляет произвольный элемент множества и запоминает его.
- set.copy() – возвращает поверхностную копию множества.
- set.clear() – очищает множество, удаляя все его элементы.

Пример программы использования методов, описанных выше, приведен в листинге 2.110.

Листинг 2.110. Использование методов для обработки множеств

```
# Листинг 2.110
# Сохраняем множество в переменной.
s_1 = {"Понедельник",  "Вторник", "Среда", "Четверг"}
print('    Исходное множество')
print(s_1)
print('-'*45)

print('Добавим элемент со значением 444')
s_1.add(444)
print('s_1.add(444) ->', s_1)
print('-'*45)

print('Удалим элемент со значением 444')
s_1.discard(444)
print('s_1.discard(444) ->', s_1)
print('-'*45)

print('Удалим элемент со значением "Четверг"')
s_1.remove("Четверг")
print('s_1.remove("Четверг") ->', s_1)

print('Удалим произвольный элемент и запомним его в del_elem')
del_elem = s_1.pop()
print('s_1.pop() ->', s_1)
print('Удаленный элемент ->', del_elem)

print('Копируем множество s_1 в s_2')
s_2 = s_1.copy()
print('Множество s_2 ->', s_2)

print('Очищаем множество s_2')
s_2.clear()
print('Пустое множество s_2 ->', s_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.110. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
   Исходное множество
{'Среда', 'Вторник', 'Понедельник', 'Четверг'}
-----------------------------------------------
Добавим элемент со значением 444
s_1.add(444) -> {'Среда', 'Вторник', 'Понедельник', 444, 'Четверг'}
-----------------------------------------------
Удалим элемент со значением 444
s_1.discard(444) -> {'Среда', 'Вторник', 'Понедельник', 'Четверг'}
-----------------------------------------------
Удалим элемент со значением "Четверг"
s_1.remove("Четверг") -> {'Среда', 'Вторник', 'Понедельник'}
Удалим произвольный элемент и запомним его в del_elem
s_1.pop() -> {'Вторник', 'Понедельник'}
Удаленный элемент -> Среда
Копируем множество s_1 в s_2
Множество s_2 -> {'Вторник', 'Понедельник'}
Очищаем множество s_2
Пустое множество s_2 -> set()
```

Помимо собственных методов к множествам применим и ряд встроенных функций, например:

- len() - возвращает количество элементов во множестве;
- min() - возвращает элемент с минимальным значением, при этом элементы должны быть одного типа с возможностью сравнения;
- max() - возвращает элемент с максимальным значением, при этом элементы должны быть одного типа с возможностью сравнения;
- sum() возвращает сумму элементов множества, при этом элементы должны быть числами.

Пример программы использования функций, описанных выше, приведен в листинге 2.111.

Листинг 2.111. Использование функций для обработки множеств

```
# Листинг 2.111
# Сохраняем множество в переменной
s_1 = {1,  22, 33, 44}
print('Исходное множество ->', s_1)
print('Количество элементов множества ->', len(s_1))
print('Минимальный элемент множества ->', min(s_1))
```

```
print('Максимальный элемент множества ->', max(s_1))
print('Сумма элементов множества ->', sum(s_1))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.111. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.118).



Рис. 2.118. Использование функций для обработки множеств

2.7.4. Фиксированные множества в Python

Довольно часто возникают ситуации, когда нужно обеспечить неизменяемость используемого множества. В таких случаях при создании или преобразовании множества необходимо вместо конструктора set([iterable]) использовать конструктор frozenset([iterable]), который создает *фиксированное множество*. Единственное отличие фиксированного множества от обычного заключается в том, что оно неизменяемо. Соответственно для фиксированных множеств становятся недоступны методы, которые пытаются их изменить. Ниже приведен пример программного кода, который создает фиксированное множество:

```
s_1 = frozenset({1, 2, 3})
```

2.7.5.Краткие итоги раздела 2.7

1. В Python множества представляют собой неупорядоченные изменяемые коллекции объектов неизменяемого типа.

2. Чтобы создать литерал множества, нужно просто перечислить элементы множества через запятую внутри фигурных скобок (например, s = {'a', 'b', 'c'}). В случае необходимости последний элемент множества можно завершить запятой. Это никак не повлияет на количество элементов множества, а значит и его длину. При этом следует помнить, что пустое множество при помощи фигурных скобок создать не получится, так как конструкция {} зарезервирована под пустой словарь.
3. Множества могут быть созданы при помощи конструктора типа - set([iterable]), который в качестве аргумента может принимать итерируемый объект, например, строку или список. Если использовать конструктор без аргумента, он вернет пустое множество set(). Если использовать конструктор с аргументом, он вернет множество с элементами переданного итерируемого объекта. Например, инструкция set('abcd') вернет множество {'a', 'b', 'c', 'd'}.
4. Для создания множеств могут использоваться генераторы множеств, которые применяют выражение к каждому элементу передаваемого итерируемого объекта и на основе их вычисления возвращают новое множество. Например, генератор {x: for x in 'abcd'} вернет нам все то же множество {'a', 'b', 'c', 'd'}.
5. Что касается операторов «+», «+=», «*» и «*=», то они с множествами напрямую не используются. Однако к множествам модно применять инструкцию проверки на вхождение «in» (проверяет наличие в множестве целевого объекта) и инструкцию организации цикла «for» (позволяет совершить обход всех элементов множества).
6. Множествам доступен целый ряд математических операторов, которые позволяют выполнять над множествами операции объединения - «|», пересечения – «&»,«-».
7. Множества имеют ряд методов, которые дают возможность выполнять операции над множествами. При этом нужно помнить, что методы изменяют само множество, а не возвращают его измененную копию.
8. Множества могут быть фиксированными, они отличаются от обычных множеств тем, что их нельзя изменять.

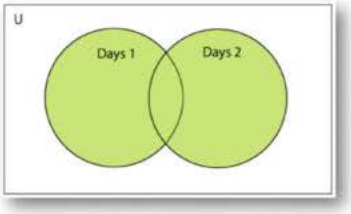
2.7.6.Вопросы для самоконтроля к разделу 2.7



1. Какое из утверждений о множествах верно:
   - неупорядоченные элементы, не индексированные;
   - упорядоченные элементы, не индексированные;
   - неупорядоченные элементы, индексированные;
   - упорядоченные элементы, индексированные.
2. Можно ли удалять элементы из множества или добавлять в него новые?
   - Да;
   - Нет;
   - Только добавлять;
   - Только удалять.
3. Какие из представленных литералов относятся к множеству:
   - {3, 4};
   - {1: 'один', 2: 'два'};
   - [3, 4];
   - "{5, 6}".

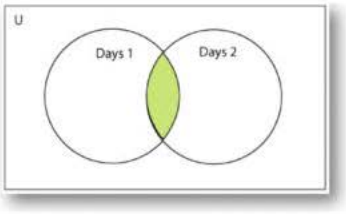
4. Какой оператор позволяет объединить два множества:

- оператор «|»;
- оператор «&»;
- оператор «-»;
- оператор «^».



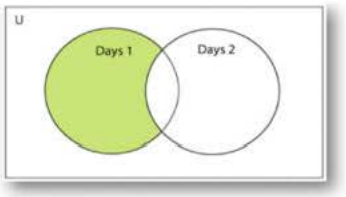
5. Какой оператор позволяет найти пересечение двух множеств:

1. оператор «&»;
2. оператор «-»;
3. оператор «^»;
4. оператор «|»;



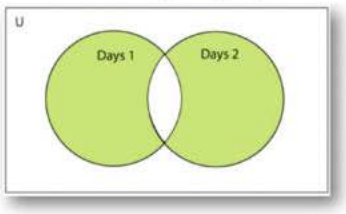
6. Какой оператор позволяет найти разницу двух множеств:

- оператор «-»;
- оператор «|»;
- оператор «&»;
- оператор «^».



7. Какой оператор позволяет найти симметричную разность двух множеств.

- оператор «^»;
- оператор «|»;
- оператор «&»;
- оператор «-».



2.8. Логический тип данных (bool), логические операторы и выражения и NoneType в Python

Часто в жизни приходиться соглашаться с каким-либо утверждением или отрицать его. Например, если вас спросят: число 5 больше числа 3, вы согласитесь, скажете - «Да» (утверждение истинно). А если спросят: число 5 меньше числа 3, вы скажете - «Нет» (утверждение ложно).

В Python, как и в других языках программирования, используется логический тип данных (bool). Выразить истинность или ложность утверждения можно всего двумя объектами (или, по сути, константами) True и False. Логические константы обычно являются результатом операций сравнения, особенно в выражениях, которые определяют ход дальнейших действий программы (чаще в инструкции разветвления программы «if»). С инструкцией «if» мы еще не знакомы, о ней пойдет речь в отдельном разделе, а пока рассмотрим использование констант True и False на простых логических выражениях. Для создания логических выражений используются следующие операторы сравнения:

- «==» - равно;
- «!=» - не равно;
- «>» - больше;
- «<» - меньше;
- «>=» - больше или равно;
- «<=» - меньше или равно;

С использованием операторов сравнения можно создавать логические выражения, в которых сравнивается значения переменных или выражений.

Пример программы, в которой используются операторы сравнения, приведен в листинге 2.112.

Листинг 2.112. Использование операторов сравнения

```
# Листинг 2.112
print('Начальные значения переменным x и y')
x = 10
y = 20
print('x =', x)
print('y =', y)

z = (x == 10)
print('x == 10 ->', z)

z = (x != 10)
print('x != 10 ->', z)

z = (x > y)
print('x > y ->', z)
```

```
z = (x < y)
print('x < y ->', z)

z = (x >= 10)
print('x >= 10 ->', z)

z = (y <= 20)
print('y <= 20 ->', z)

str_1 = "Привет"
print('str_1 ->', str_1)
res = (str_1 == "Привет")
print('(str_1 == "Привет") ->', res)
res = (str_1 == "Ответ")
print('(str_1 == "Ответ") ->', res)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.112. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
Начальные значения переменным x и y
x = 10
y = 20
x == 10 -> True
x != 10 -> False
x > y -> False
x < y -> True
x >= 10 -> True
y <= 20 -> True
str_1 -> Привет
(str_1 == "Привет") -> True
(str_1 == "Ответ") -> False
```

Из данного примера видно, что сравнивать между собой можно не только числа, но и строки. Обратите внимание, что в Python имена булевых констант True и False зависят от регистра, поэтому использовать их нужно именно в указанном здесь виде (нельзя использовать написание констант в виде true и false).

Стоит добавить, что True и False, по сути, являются обычными целыми числами 1 и 0, пусть и с некоторыми особенностями отображения на экране.

Пример программы, в которой показано, что с логическими константами True и False можно оперировать как с обычными числами, приведен в листинге 2.113.

Листинг 2.113. Логические константы True и False

```
# Листинг 2.113
res = True + True
print('True + True =', res)

res = False + False
print('False + False =', res)

res = True + False
print('True + False =', res)

res = (1 == True)
print('(1 == True) ->', res)

res = (0 == False)
print('(0 == False) ->', res)
print('-'*20)

print('Выведем тип данных')
print('type(True) ->', type(True))
print('type(False) ->', type(False))
print('type(1) ->', type(1))
print('type(0) ->', type(0))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.113. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.119).

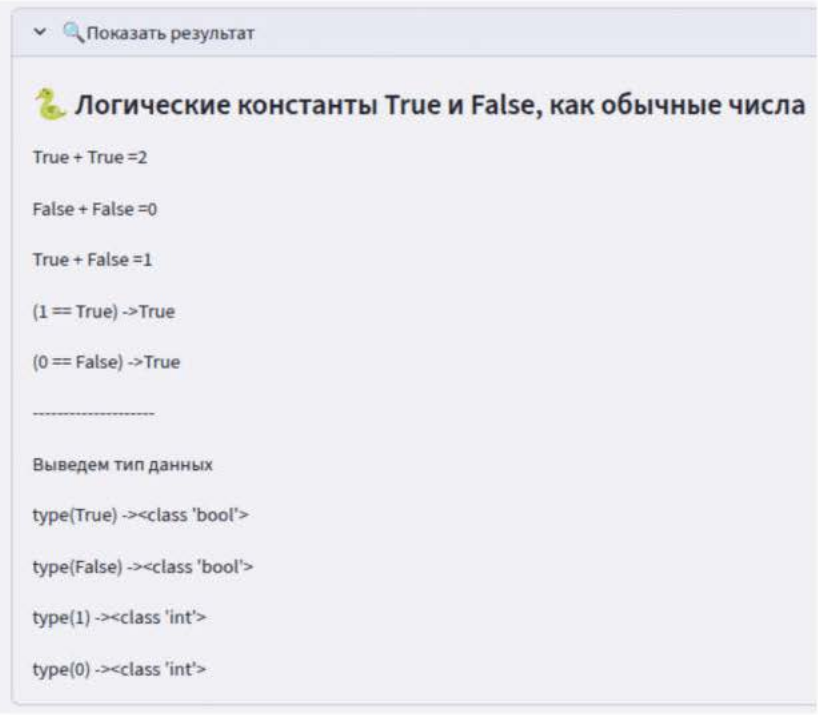


Рис. 2.119. Логические константы True и False

Как видно из данного примера, булевы константы действительно можно использовать в математических выражениях наравне с числами. Точно также и в операциях сравнения можно использовать вместо констант True и False числа 1 и 0. Поэтому для себя их можно считать просто именованными «ноликами» и «единичками» не забывая при этом, что мы имеем дело с отдельным типом данных.

Для составления более сложных логических выражений Python поддерживает и стандартные логические операторы: «and» (логическое И), «or» (логическое ИЛИ), «not» (логическое НЕ). Все они используются в логических выражениях для определения того или иного хода выполнения программы. В зависимости от возвращаемого выражением результата логические операторы будут возвращать следующие значения:

- s and f - логическое «И» (True, если оба значения s и f равны, в остальных случаях False);
- s or f - логическое «ИЛИ», если хотя бы один из операндов истинный, то и все выражение будет истинным (True or True равно True, True or False равно True, False or True равно True, False or False равно False);
- not s - логическое «НЕ», изменяет логическое значение операнда на противоположное (True, если значение s равно False, False, если значение s равно True).

Важно понимать, как интерпретатор обрабатывает логические выражения. Если в выражении с оператором «or» первый (левый) операнд будет иметь значение True или в выражении с оператором «and» первый операнд будет иметь значение False, то второй (правый) операнд уже вычисляться не будет. Это связано с тем, что конечный результат в

таких случаях уже не изменится, следовательно, отпадает и необходимость обработки кода второго операнда. Однако при этом нужно быть внимательным и не помещать в правый операнд код, от которого может зависеть правильная работа программы.

Пример программы, в которой показано использование сложных логических выражений на основе логических операторов «and», «or» и «not», приведен в листинге 2.114.

Листинг 2.114. Сложные логические выражения

```
# Листинг 2.114
print('Начальные значения переменным x и y')
x = 10
y = 20
print('x =', x)
print('y =', y)

z = (x == 10 and y == 20)
print('(x == 10 and y == 20) ->', z)

z = (x == 10 and y == 10)
print('(x == 10 and y == 10) ->', z)

z = (x == 10 or y == 10)
print('(x == 10 or y == 10) ->', z)

z = (x == 20 or y == 30)
print('(x == 20 or y == 30) ->', z)

z = (x == 10)
print('(x == 10) ->', z)

z = (not x == 10)
print('(not x == 10) ->', z)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.114. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.120).

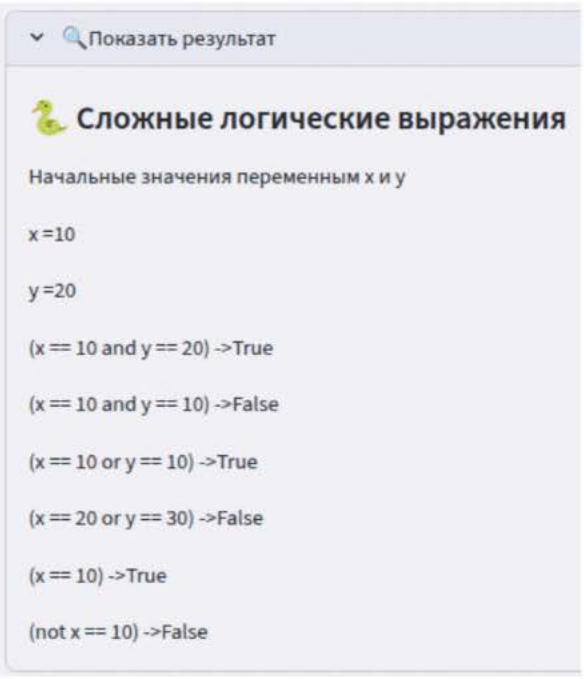


Рис. 2.120. Сложные логические выражения

2.8.1.Тип данных NoneType в Python

В Python единственным представителем данных NoneType является специальная константа None, означающая отсутствие значения. Обычно ее используют для инициализации переменных и объектов, которым значения будет присвоено позже в процессе выполнения программы. Чаще такой тип данных используется при инициализации классов, когда нужно заранее объявить общую переменную, которая будет использоваться в разных методах класса, но о классах мы поговорим позже. Кроме того, None неявно возвращается функциями, у которых отсутствует оператор return.

Пример программы, в которой мы резервируем переменную, а затем присваиваем ей данные разных типов, приведен в листинге 2.115.

Листинг 2.115. Тип данных NoneType в Python

```
Листинг 2.115
# Инициализируем переменную
n = None
print('n ->', n)
print('type(n) ->', type(n))
is_none = n is None
print('is_none ->', is_none)
print('-'*30)
```

```
# Создаем список
n = [1, 2, 3, 4]
print('n ->', n)
print('type(n) ->', type(n))

# Создаем кортеж
n = (1, 2, 3, 4)
print('n ->', n)
print('type(n) ->', type(n))

# Создаем множество
n = {1, 2, 3, 4}
print('n ->', n)
print('type(n) ->', type(n))

is_none = n is None
print('is_none ->', is_none)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.115. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
n -> None
type(n) -> <class 'NoneType'>
is_none -> True
------------------------------
n -> [1, 2, 3, 4]
type(n) -> <class 'list'>
n -> (1, 2, 3, 4)
type(n) -> <class 'tuple'>
n -> {1, 2, 3, 4}
type(n) -> <class 'set'>
is_none -> False
```

Как и в случае с логическими константами, имя None чувствительно к регистру символов, поэтому использовать его нужно именно в указанном здесь виде.

Что касается проверки переменной на значение None, то сделать это надежнее всего при помощи инструкции «is». Например, для инструкции n = None проверка n is None вернет True.

2.8.2. Вопросы для самоконтроля к разделу 2.8



1. Какая константа написана без ошибки:
   - True;
   - NONE;

- none
- false;
- true.

2. Какая константа написана без ошибки:
   - False;
   - NONE;
   - none
   - true;
   - false.
3. Какая константа написана без ошибки:
   - 1 None
   - 2 NONE;
   - 3 false;
   - 4 true.
4. Какое выражение имеет результат True:
   1. 5 < 7;
   2. 5 >= 7;
   3. 5 > 9;
   4. 9 <= 7.
5. Какое выражение имеет результат False:
   - 5 > 7
   - 5 >= 5;
   - 5 < 7;
   - 9 >= 7.
6. Какое выражение имеет результат True:
   - 7 >= 5 and 3 < 9;
   - 7 <= 5 and 3 < 9;
   - 7 == 5 and 3 < 9;
   - 7 > 5 and 3 > 9.
7. Какое выражение имеет результат False:
   - 7 < 5 or 3 > 9;
   - 7 < 5 or 3 > 9;
   - 7 <= 5 or 3 > 9;
   - 7 == 5 or 3 > 99.
8. Какое выражение имеет результат True:
   - not 10 == 20;
   - not 10 < 20;
   - not 10 <= 20;
   - not 20 > 10.

2.9. Приведение типов данных в Python

Поскольку Python относится к языкам программирования с динамической типизацией данных, ответственность за предоставление данных нужного типа практически полностью ложится на плечи программистов. В операциях с различными типами чисел или же в условных инструкциях и выражениях интерпретатор автоматически преобразует объекты к общему типу. Однако при работе с текстовыми и числовыми данными следует внимательно следить за текущим типом переменной и, при необходимости, самостоятельно выполнять приведение данных к нужному типу.

Приведением типа называется процесс преобразования значения одного типа данных в значение другого типа.

Для согласования различных типов данных между собой в Python доступен целый ряд встроенных функций (или конструкторов типов):

- bool() – приведение к логическому типу;
- int() – приведение к целочисленному типу;
- float() – приведение к вещественному типу;
- complex() – приведение к комплексному числу;
- str() – приведение к строковому типу;
- tuple() – приведение к кортежу;
- list() – приведение к списку;
- dict() – приведение к словарю;
- set() – приведение к множеству.

Использование перечисленных выше функций требует знания правил, по которым происходит преобразование типов данных. Например, не каждую строку можно преобразовать в число, хотя обратное преобразование возможно всегда. И таких нюансов можно перечислить довольно много. Поэтому будет не лишним рассмотреть порядок приведения типов более детально.

Ниже описаны общие правила приведения типов для конструкторов без дополнительных параметров. Если вдруг возникнет необходимость в более экзотических приемах преобразования, следует обращаться к описаниям конструкторов в справочнике встроенных функций стандартной библиотеки.

2.9.1. Приведение данных к типу bool (логический тип данных)

Во время преобразования к типу bool (логический тип данных) следующие значения рассматриваются как False:

- само значение False,
- значение None,
- нулевые числовые значения 0, 0.0 и 0+0j,
- пустая строка "",

- пустые объекты: [], tuple(), {}, dict().

Все остальные значения преобразуются в значение True, включая число -1, поскольку оно представляет собой обычное отрицательное число, отличное от нуля. То же самое касается коллекций с одним нулевым значением, например, [0], {0.0}, (0,), а также непустых строк, например, '0' или '0.0'.

Пример программы, в которой показано приведение различных данных к логическому типу функцией bool(), приведен в листинге 2.116.

Листинг 2.116. Приведение различных данных к логическому типу

```
# Листинг 2.116
print('Эти данные будут иметь значение - False')
print('bool(None):', bool(None))
print('bool(0):', bool(0))
print('bool(0.0):', bool(0.0))
print('bool(0+0j):', bool(0+0j))
print('bool(""):', bool(''))
print('bool([]):', bool([]))
print('bool({}):', bool({}))
print('bool(tuple()):', bool(tuple()))
print('bool(dict()):', bool(dict()))

print()
print('Эти данные будут иметь значение - True')
print('bool("0"):', bool('0'))
print('bool(-1):', bool(-1))
print('bool([False]):', bool([False]))
print('bool({0,}):', bool({0, }))
print('bool((0,)):', bool((0, )))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.116. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в среде Python, то будет получен следующий результат:

```
Эти данные будут иметь значение - False
bool(None): False
bool(0): False
bool(0.0): False
bool(0+0j): False
bool(""): False
bool([]): False
bool({}): False
bool(tuple()): False
bool(dict()): False
```

```
Эти данные будут иметь значение - True
bool("0"): True
bool(-1): True
bool([False]): True
bool({0,}): True
bool((0,)): True
```

2.9.2.Приведение данных к типу int (целые числа)

В случае приведения к типу int (целые числа) будут соблюдаться следующие правила.

- логическое значение True преобразуется в 1, False – в 0.
- вещественные числа просто усекаются до целой части. Например, в случае приведения вещественного числа -37.9 к целочисленному типу оно будет преобразовано в число -37.
- если строка содержит верную запись целого числа в десятичной системе счисления, то интерпретатор в процессе приведения строки к целочисленному типу вернет это число.
- если строка num_str содержит верное представлением целого числа в системе счисления с указанным основанием base, то преобразовать ее можно в целое число при помощи конструктора типа с двумя аргументами int(num_str, base=10).

Для других типов данных приведение к целочисленному типу не определено, поэтому попытка такого преобразования приведет к ошибке.

Пример программы, в которой показано приведение различных данных к целому типу функцией int(), приведен в листинге 2.117.

Листинг 2.117. Приведение различных данных к типу – целое число

```
# Листинг 2.117
print('Эти данные будут иметь тип - целое число')
print('int(True): ', int(True))
print('int(False): ', int(False))
print('int("+55"): ', int('+55'))
print('int(-17.33): ', int(-17.33))
print('int(.44): ', int(.44))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.117. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.121).

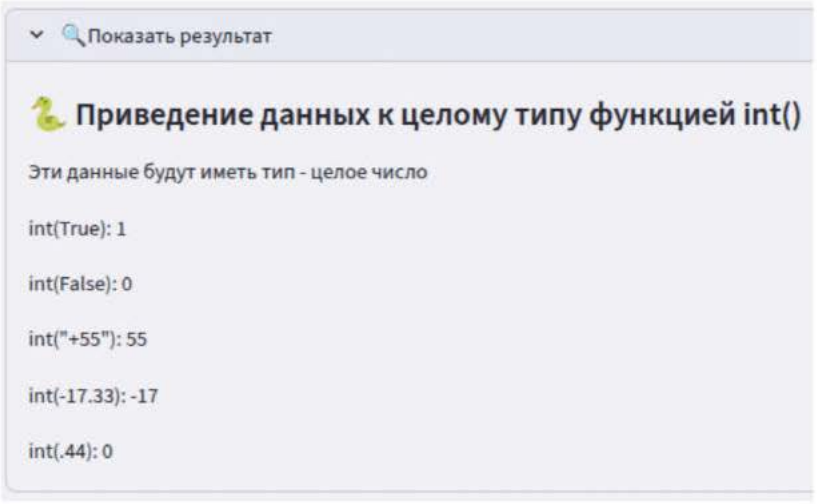


Рис. 2.121. Приведение различных данных к типу – целое число

2.9.3.Приведение данных к типу float (вещественное число)

В случае приведения к типу float (вещественные числа) будут соблюдаться следующие правила.

- логическое значение True преобразуется в 1.0, False – в 0.0.
- Целые числа преобразуются в вещественные числа простым добавлением нулевой дробной части. Например, целое отрицательное число -37 будет преобразовано в число -37.0.
- если строка содержит верное представление целого числа в десятичной системе счисления, то интерпретатор в процессе приведения строки к вещественному типу вернет это число, добавив к нему нулевую дробную часть. При этом допускаются начальные и конечные пробельные символы, а также знаки плюса и минуса. Например, строка, содержащая запись целого числа ‘105’ будет преобразовано в число 105.0.
- если строка содержит верное представление вещественного числа в десятичной системе счисления, то интерпретатор в процессе приведения строки к вещественному типу вернет это число. При этом допускаются начальные и конечные пробельные символы, а также знаки плюса и минуса. Например, строка, содержащая запись вещественного отрицательного числа с пробелами ' -1.5 ' будет преобразовано в число -1.5.
- также строка может представлять значения бесконечности ('NaN', 'Infinity' или '-Infinity').

Для других типов данных приведение к вещественному типу не определено, поэтому попытка такого преобразования приведет к ошибке.

Пример программы, в которой показано приведение различных данных к вещественому типу функцией float(), приведен в листинге 2.118.

Листинг 2.118. Приведение различных данных к типу – вещественное число

```
# Листинг 2.118
print('Эти данные будут иметь тип - вещественное число')
print('float(True): ', float(True))
print('float(False): ', float(False))
print('float(" +55"): ', float(' +55'))
print('float("-7.44"): ', float('-7.44'))
print('float("1.4e-2 "): ', float('1.4e-2 '))
print('float("NaN"): ', float('NaN'))
print('float("-Infinity"): ', float('-Infinity'))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.118. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.122).

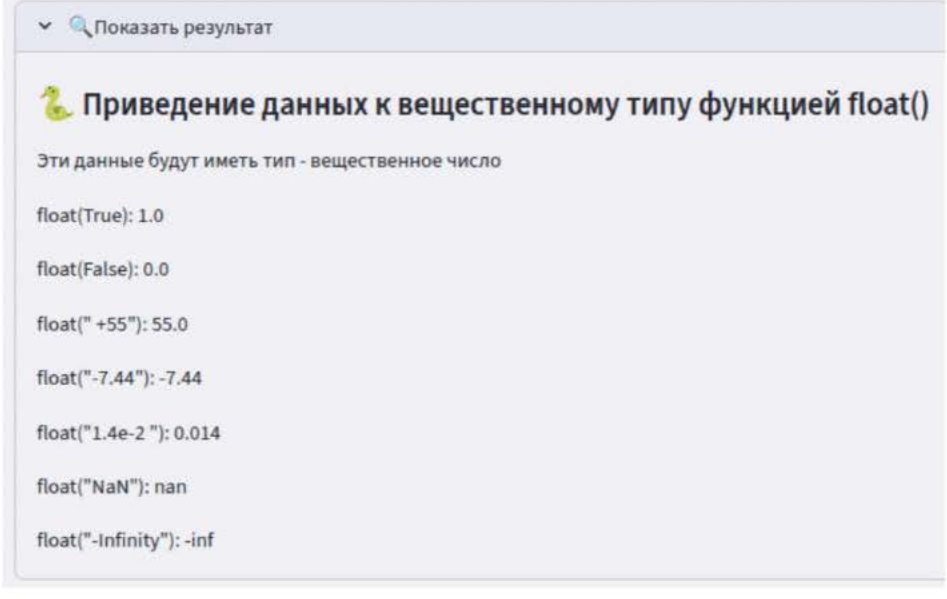


Рис. 2.122. Приведение различных данных к типу – вещественное число

2.9.4.Приведение данных к типу complex (комплексные числа)

В случае приведения к типу complex (комплексные числа) будут соблюдаться следующие правила.

- логическое значение True преобразуется в (1+0j), False – в 0j.
- целые и вещественные числа преобразуются в комплексные числа простым добавлением нулевой мнимой части «+0j». Например, отрицательное вещественное число -3.7 будет преобразовано в комплексное число (-3.7+0j).
- если строка содержит верное представление комплексного числа, то интерпретатор в процессе приведения строки к комплексному типу вернет это число. При этом допускаются начальные и конечные пробельные символы, а также знаки плюса и минуса, как перед действительной, так и мнимой частью. Например, строка, содержащая запись комплексного числа с пробелами и знаком ' -5.5+0.2j ' будет преобразовано в число (-5.5+0.2j). Однако пробелов между действительной частью и мнимой частью быть не должно!

Для других типов данных приведение к комплексному типу не определено, поэтому попытка такого преобразования приведет к ошибке.

Пример программы, в которой показано приведение различных данных к вещественному типу функцией complex(), приведен в листинге 2.119.

Листинг 2.119. Приведение различных данных к типу – комплексное число

```
# Листинг 2.119
print('Эти данные будут иметь тип - комплексное число')
print('complex(True):', complex(True))
print('complex(False):', complex(False))
print('complex(" +4.9-3.2j"):', complex(' +4.9-3.2j'))
print('complex(" 55"):', complex(' 55'))
print('complex(" -7.44 "):', complex(' -7.44 '))
print('complex("1.2e-2 "):', complex('1.2e-2 '))
print('complex("NaN"):', complex('NaN'))
print('complex(" -Infinity "):', complex(' -Infinity '))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.119. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.123).

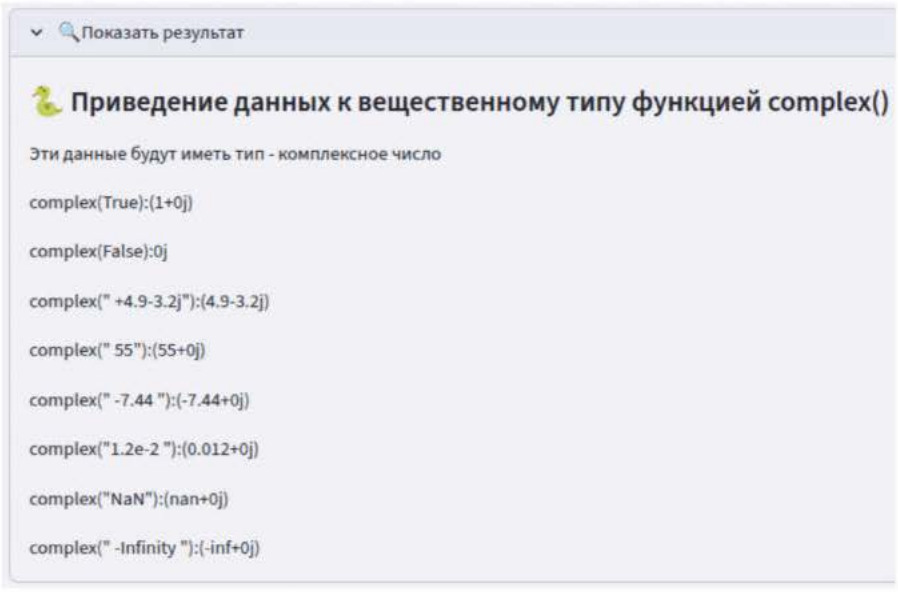


Рис. 2.123. Приведение различных данных к типу – комплексное число

2.9.5. Приведение данных к типу str (строка)

В случае приведения к типу str (строка) интерпретатор просто вернет строковое представление объекта. В строку может быть преобразован любой тип данных.

Пример программы, в которой показано приведение различных данных к типу «строка» функцией str(), приведен в листинге 2.120.

Листинг 2.120. Приведение различных данных к типу – строка

```
# Листинг 2.120
print('Эти данные будут иметь тип - строка')
print('str(True):', str(True))
print('str(False):', str(False))
print('str+6.9-3.2j:', str(+6.9-3.2j))
print('str(55):', str(+55))
print('str(-7.44):', str(-7.44))
print('str(1.2e-2):', str(1.2e-2))
print('str(None):', str(None))
print('str([0.15, 1,5]):', str([0.15, 1, 5]))
print("str({'Один', 'Два'}):", str({'Один', 'Два'}))
print("str((5, '6')):", str((5, '6')))
print('str({11: [22, True]}):', str({11: [22, True]}))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел

«Листинг 2.120. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.124).

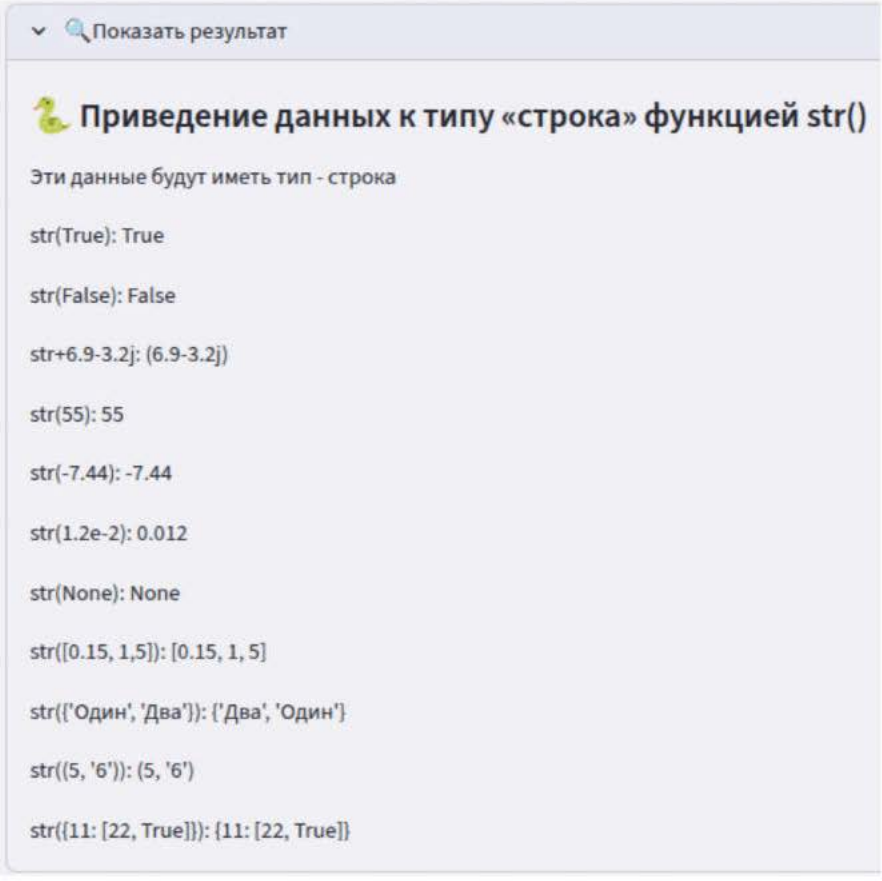


Рис. 2.124. Приведение различных данных к типу – строка

2.9.6.Приведение данных к типу tuple (кортеж)

К типу данных tuple (кортежи) могут быть преобразованы только итерируемые объекты. При этом словари, которые преобразуются в кортежи, теряют значения элементов (в кортеж будут перенесены только ключи словаря).

Пример программы, в которой показано приведение различных данных к типу «кортеж» функцией tuple(), приведен в листинге 2.121.

Листинг 2.121. Приведение различных данных к типу – кортеж

```
# Листинг 2.121
print('Эти данные будут иметь тип - кортежи')
print("tuple('ABC') ->", tuple('ABC'))
print('tuple([1, 2, 3]) ->', tuple([1, 2, 3]))
print("tuple({'Один', 'Два'}) ->", tuple({'Один', 'Два'}))
print("tuple({1: 'Один', 2: 'Два'}) ->", tuple({1: 'Один', 2: 'Два'}))
print("tuple({'Один': 1, 'Два': 2}) ->", tuple({'Один': 1, 'Два': 2}))
print("tuple(range(1, 4)) ->", tuple(range(1, 4)))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.121. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.125).

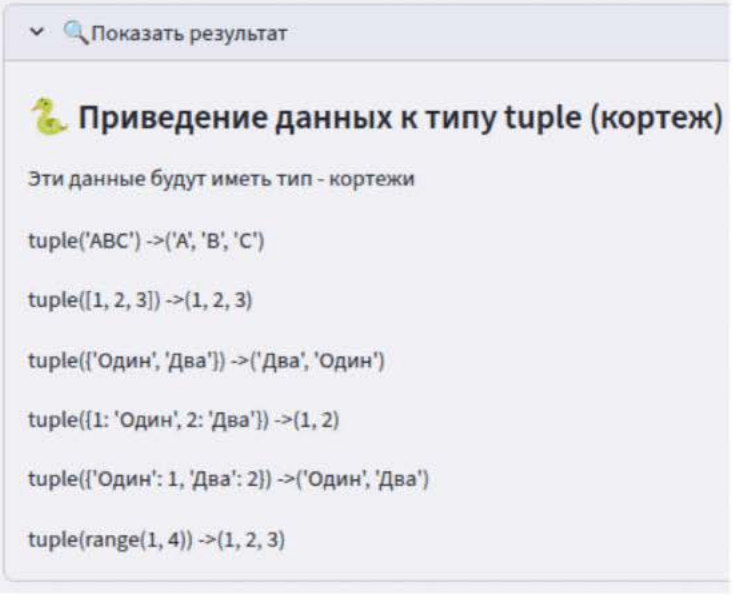


Рис. 2.251. Приведение различных данных к типу – кортеж

2.9.7. Приведение данных к типу list (список)

К типу данных list (списки) могут быть преобразованы только итерируемые объекты. При этом словари, которые преобразуются в список, теряют значения элементов (в список будут перенесены только ключи словаря).

Пример программы, в которой показано приведение различных данных к типу «список» функцией list(), приведен в листинге 2.122.

Листинг 2.122. Приведение различных данных к типу – список

```
# Листинг 2.122
print('Эти данные будут иметь тип - списки')
print("list('ABC') ->", list('ABC'))
print('list((1, 2, 3)) ->', list((1, 2, 3)))
print("list({'Один', 'Два'}) ->", list({'Один', 'Два'}))
print("list({1: 'Один', 2: 'Два'}) ->", list({1: 'Один', 2: 'Два'}))
print("list({'Один': 1, 'Два': 2}) ->", list({'Один': 1, 'Два': 2}))
print("list(range(1, 5)) ->", list(range(1, 5)))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.122. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.126).

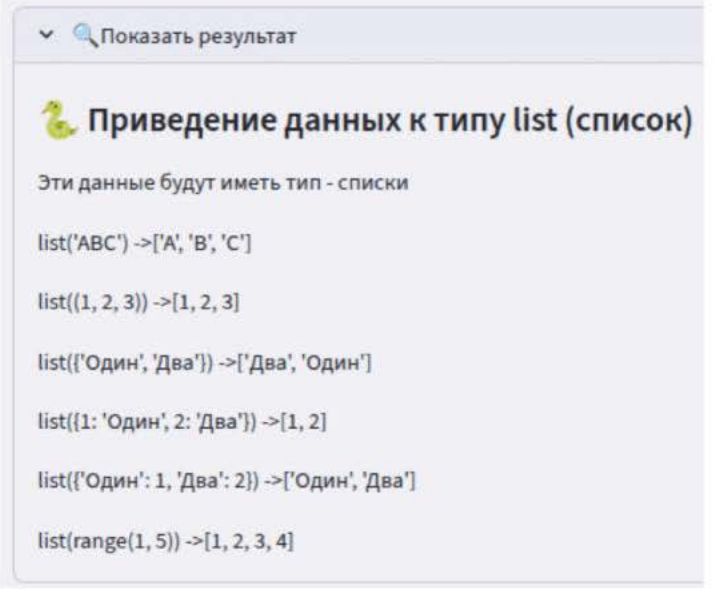


Рис. 2.126. Приведение различных данных к типу – список

2.9.8. Приведение данных к типу set (множества)

К типу данных set (множества) могут быть преобразованы только итерируемые объекты. При этом словари, которые преобразуются во множества, теряют значения элементов (во множества будут перенесены только ключи словаря).

Пример программы, в которой показано приведение различных данных к типу «множества» функцией set(), приведен в листинге 2.123.

Листинг 2.123. Приведение различных данных к типу – множества

```
# Листинг 2.123
print('Эти данные будут иметь тип - множество')
print("set('ABC') ->", set('abc'))
print('set((1, 2, 3)) ->', set((1, 2, 3)))
print("set({1: 'Один', 2: 'Два'}) ->", set({1: 'Один', 2: 'Два'}))
print("set({'Один': 1, 'Два': 2}) ->", set({'Один': 1, 'Два': 2}))
print("set(range(1, 4)) ->", set(range(1, 4)))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.123. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.127).

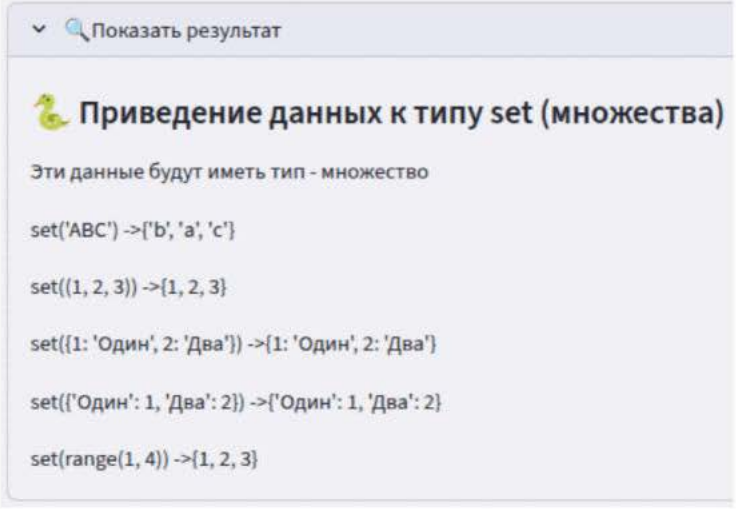


Рис. 2.127. Приведение различных данных к типу – множества

2.9.9. Приведение данных к типу dict (словари)

К типу данных dict (словари) напрямую могут быть преобразованы только итерируемые объекты, содержащие пары значений, в которых первые элементы становятся ключами словаря, а вторые – его значениями. Например, это могут быть списки, кортежи или множества, содержащие в качестве своих элементов другие списки, кортежи или множества

из двух элементов. При этом следует помнить, что только хешируемые (неизменяемые) объекты могут быть ключами словаря или элементами множества.

Пример программы, в которой показано приведение различных данных к типу «словарь» функцией dict(), приведен в листинге 2.124.

Листинг 2.124. Приведение различных данных к типу – словарь

```
# Листинг 2.124
print('Эти данные будут иметь тип - словарь')

print('Словари из списков')
print("dict([[1, 'Один'], (2, 'Два')]) ->", dict([[1, 'Один'], (2,
'Два')]))
print("dict([(1, 'Один'), {2, 'Два'}])->", dict([(1, 'Один'), {2,
'Два'}]))

print('Словари из кортежей')
print("dict(((1, 'Один'), {2, 'Два'}))->", dict(((1, 'Один'), {2,
'Два'})))
print("dict(({1, 'Один'}, [2, 'Два']))->", dict(({1, 'Один'}, [2,
'Два'])))
print("dict({(1, 'Один'), (2, 'Два')})->", dict({(1, 'Один'), (2,
'Два')}))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 1» выбрать опцию «Листинги главы 2», и в выпадающем списке боковой панели выбрать раздел «Листинг 2.124. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 2.128).

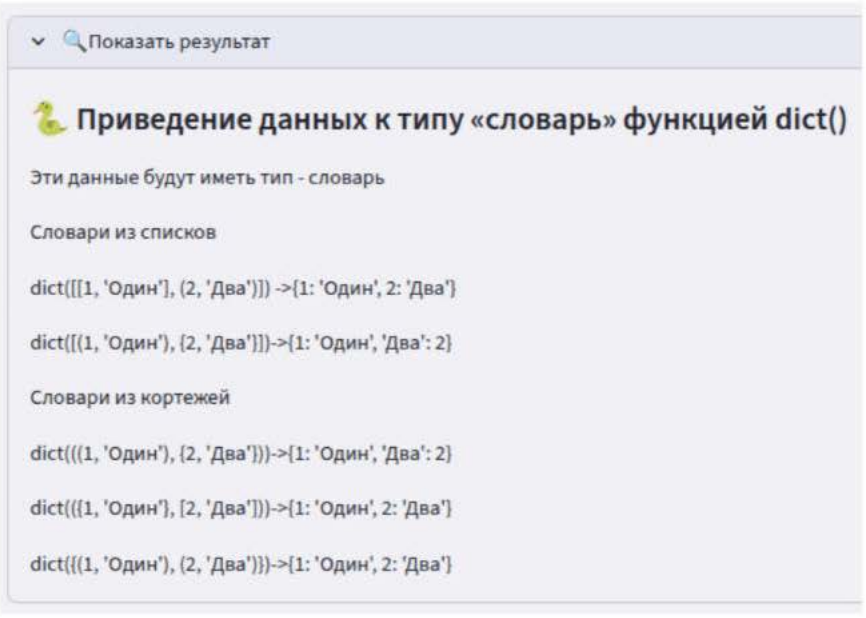


Рис. 2.128. Приведение различных данных к типу – словарь

2.9.10. Краткие итоги раздела 2.9

Для преобразования различных типов данных в Python доступен целый ряд встроенных функций (конструкторов типов):

- bool() – приведение к логическому типу;
- int() – приведение к целочисленному типу;
- float() – приведение к вещественному типу;
- complex() – приведение к комплексному числу;
- str() – приведение к строковому типу;
- tuple() – приведение к кортежу;
- list() – приведение к списку;
- dict() – приведение к словарю;
- set() – приведение к множеству.

В операциях с различными типами чисел или же в условных инструкциях и выражениях интерпретатор неявно (автоматически) преобразует объекты к общему типу. Однако в большинстве случаев это не так, поэтому следует внимательно следить за текущим типом переменной и, при необходимости, самостоятельно осуществлять явное приведение к нужному типу.

2.9.11. Вопросы для самоконтроля к разделу 2.9



1. Какое значение вернет функция int(True):

- 1;
- 0;
- None;
- True.

2. Какое значение вернет функция float(False):
    1. 0.0;
    2. 0;
    3. 1.0
    4. False.
3. Какое значение вернет функция complex(True):
    - 1+0j;
    - 1.0+0j;
    - 0+1j;
    - True.
4. Какое значение вернет функция str(True):
    - True;
    - 1;
    - 1+1j;
    - False.
5. Какое значение вернет функция tuple('ABC'):
    - ('A', 'B', 'C');
    - ['A', 'B', 'C'];
    - {'A', 'B', 'C'};
    - ('ABC').
6. Какое значение вернет функция list('ABC'):
    - ['A', 'B', 'C']
    - ('A', 'B', 'C');
    - {'A', 'B', 'C'};
    - ('ABC').
7. Какое значение вернет функция set('ABC'):
    - {'C', 'B', 'A'}
    - ['C', 'B', 'A'];
    - ('1', '2', '3');
    - ('ABC').
8. Какое значение вернет функция dict([[1, 'Один'], (2, 'Два')])):
    - {1: 'Один', 2: 'Два'};
    - [1: 'Один', 2: 'Два'];
    - (1: 'Один', 2: 'Два');
    - {'1': 'Один', '2': 'Два'}.

Глава 3.Базовые компоненты языка программирования Python



Любая книга состоит глав, главы можно разбить на разделы, разделы на абзацы, абзацы на слова, слова на буквы. В итоге мы остановились на букве – это тот базовый элемент, из которого можно сформировать любую книгу. Если провести параллель с программированием, то любое приложение состоит из модулей, модули из строк программного кода, строки из инструкций, выражений, операторов и ключевых слов. Для того чтобы написать приложение, нужно знать представление о базовых элементах, из которых оно будет состоять. Из ключевых слов, операторов и выражений составляются инструкции, которые затем и выполняет интерпретатор. Вот этим элементам языка программирования Python и посвящена данная глава. Из материалов этой главы вы узнаете:

- общие сведения об инструкциях, выражениях и операторах;
- что такое инструкция присваивания и как она работает с различными типами данных;
- как инструкция «if» позволяет направить ход выполнения программы по разным веткам;
- какие инструкции позволяют организовать циклическое выполнение фрагментов программного кода;
- что такое итераторы и генераторы в Python;
- как можно обработать ошибки, которые могут возникнуть в ходе выполнения программ, с использованием инструкции «try»;
- как и где используется менеджер контекста (инструкция with/as);
- какие инструкции обеспечиваются ввод и вывод данных.

3.1.Инструкции, выражения и операторы в Python

Инструкция в Python - это логически законченная конструкция, которую интерпретатор Python может прочесть и на ее основе выполнить какие либо действия.

Выражение — один из типов инструкции, оно содержит логическую последовательность чисел, строк, объектов и операторов. Значение и переменная являются выражениями сами по себе. С помощью выражений можно выполнять такие операции, как сложение, вычитание, конкатенация и другие. Они также могут представлять собой вызовы функций, которые возвращают результат.

Оператор в Python — это символ, который указывает, что нужно сделать с одним или несколькими операндами. В качестве операторов используются специальные символы, например: «+» - сложение, «-» - вычитание, «*» - умножение, «/» - деление. Операндом может быть константа, переменная, функция и т.п., то есть некие значения, над которыми выполняется операция.

Для новичков в программном коде бывает достаточно сложно определить, что является инструкций, а что выражением. Однако они отличаются как по целям использования, так и по и смысловому содержанию. Вот их принципиальное отличие:

- 13выражения – это все, что состоит из объектов (или ссылок на них) и операторов (выражение может быть вычислено интерпретатором и получить конкретное значение);
- 14инструкции – это указания на выполнение программой каких-либо действий, которые не возвращают значений.

Выражения можно считать частью инструкций, в результате выполнения которых будет получены значения. Выражения практически всегда входят в состав каких-либо инструкций или сложных (смешанных) выражений. А вот использование инструкций в выражениях недопустимо. Например, в выражении «d = 5 + (def my_func())» присутствует инструкция «def my_func()», которая не возвращает значение, а дает указание создать функцию. Такая конструкция приведет к ошибке. А вот выражение «d = 5 + my_func()», к ошибке не приведет, поскольку «my_func()» вернет значение, вычисленное в этой функции.

В данной главе мы более подробно познакомимся:

- с инструкциями и правилами работы с ними;
- с правилами написания выражений;
- с операторами и правилами использования операторов в выражениях.

Также подведем краткие итоги главы и попробуем ответить на контрольные вопросы.

3.1.1.Инструкции в Python

Инструкция языка программирования (от англ. statement) – это команда, посредством которой компьютеру сообщается о необходимости выполнения одного определенного шага по обработке порции информации.

Для того чтобы наша программа могла делать что-то полезное, мы должны предоставить интерпретатору Python верные инструкции, то есть посредством программного кода сообщить ему логическую последовательность действий, выполнение которой и приведет к желаемому результату.

По сути, любой python-скрипт (программный код) представляет собой не что иное, как некоторую последовательность инструкций, которая может состоять из десятков, сотен, тысяч или даже сотен тысяч и миллионов инструкций. Все зависит от того, сколько инструкций потребуется для того, чтобы описать все шаги, необходимые для достижения целей программы.

Пример программы, в которой показаны различные виды инструкций, приведен в листинге 3.1.

Листинг 3.1. Примеры различных видов инструкций

```
# Листинг 3.1
from math import pi  # Инструкция импортирует переменную pi из модуля
math

x, y, z = [1, 2, 3]  # Позиционное присваивание значений переменным
```

```
# Эта инструкция объявляет функцию со значением по умолчанию
def my_func(z=10):
    s_1 = z ** 2

    for i in range(25):  # Инструкция запускает цикл
        if i < 5:  # Условная инструкция
            continue  # Инструкция перехода в начало цикла
        elif 5 <= i < 10:
            s_1 += i  # Увеличиваем z на знаначение из i
        else:
            break  # Инструкция прерывания цикла
    return s_1  # Инструкция возврата значения функции


try:                                    # Инструкция будет обрабатывать
исключения (ошибки)
    a_1 = my_func(z)                    # инструкция обращения к функции
    res = a_1 + int(pi) * 10 + x + y  # инструкция выполнения
арифметических операций
    print("Результат вычислений res=", res)    # Инструкция выводит
результат

except Exception as err:  # Блок выполняется, когда есть ошибка
    print(err)  # Выводим сообщение об ошибке
else:  # Если ошибок нет
    pass  # Ничего не делаем (пустая инструкция-заполнитель)
finally:  # Этот блок выполняется всегда, даже при наличии ошибки
    print('Конец программы')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.1. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.1).

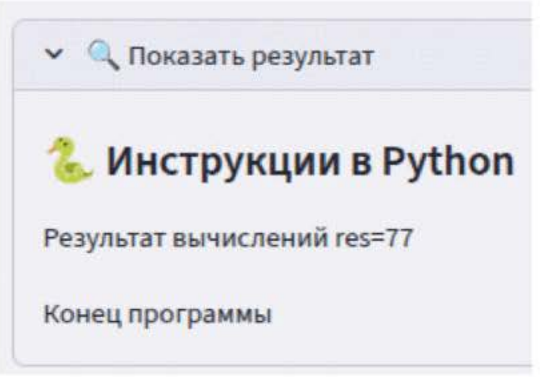


Рис. 3.1. Примеры различных видов инструкций

Как видно из данного примера, к инструкциям можно отнести практически все команды, которые мы пишем в коде программы:

- импорт модулей;
- присваивание или удаление ссылок на объекты;
- определение функций;
- циклы;
- условные конструкции;
- обработка исключений и т.д.

В Python инструкции посредством отступов могут быть объединены в блоки. Каждый из таких блоков может содержать собственные вложенные блоки инструкций. В последующих разделах все виды инструкций будут рассмотрены достаточно подробно. Сейчас же мы просто перечислим основные инструкции, дав краткие пояснения, для чего они используются.

Инструкция a = b – это инструкция присваивания (например, a = 8**2 - 6 или b = len([1, 2, 3])). Помимо этого имеются и другие формы присваивания:

5. комбинированные инструкции присваивания (например, a += 5 или b **= 2),
6. позиционное присваивание (например, a, b = 5, 9 или [a, b] = 'AB'),
7. операция распаковывания последовательностей (например, a, *b = 1, 2, 3 или a, *b, c = 'ABCDEF'),
8. а также групповое присваивание одного значения (например, a = b = c = 9).

Алгоритм присваивания у всех этих форм несколько отличается, но главная цель у них одна – это создание ссылок на объекты и сохранение их в переменных. При этом стоит добавить, что присваивание может быть и неявным, например, в вызовах функций или в заголовках циклов, но в любом случае результатом будут ссылки на объекты, сохраненные в переменных.

Инструкция def – определение функций и методов в классах. Например:

```
def my_func(a, b, c=0, *d):
return a + b + c + d[0].
```

Инструкция return – возврат результата из функций и методов.

Инструкция yield – инструкция возврата результата из функции-генератора с сохранением состояния ее локальных переменных. При повторном вызове такой функции выполнение, в отличие от инструкции return, продолжается с ключевого слова yield, на котором ее работа была прервана ранее.

Инструкция class – определение классов. Например:

```
class MyClass:
pass
```

Инструкция pass – пустая инструкция-заполнитель.

Инструкция global – инструкция объявления глобальных переменных (если переменных несколько, то они перечисляются через запятую).

Инструкция nonlocal – инструкция используется во вложенных функциях и объявляет переменные нелокальными. В результате интерпретатор будет использовать переменные с указанными именами из ближайшей области видимости (т.е. объемлющей функции), исключая глобальную.

Инструкция if/elif/else – условная инструкция (разветвление программы).

Инструкция for/else – инструкция обхода последовательности в цикле.

Инструкция while/else – цикл общего назначения.

Инструкция break – инструкция выхода из цикла.

Инструкция continue – инструкция перехода в начало цикла.

Инструкция import – инструкция импорта всех доступных имен из указанных модулей (модули перечисляются через запятую).

Инструкция from/import – инструкция импорта из модуля только указанных имен (имена перечисляются через запятую). Например:

```
from math import pi as p, tau as t
```

Инструкция try/except/finally – инструкция обработки исключений (обход ошибок).

Инструкция raise – инструкция возбуждения исключений.

Инструкция assert – инструкция отладки, которая позволяет производить проверку истинности утверждений и выводить сообщение об ошибке. Например:

```
assert x > y, 'Значение x недопустимо мало!'
```

Инструкция with/as – менеджер контекста.

Инструкция del – инструкция удаления ссылок на объекты (если ссылок несколько, то они перечисляются через запятую). Например:

```
del a, b[0], c['car']
```

Отметим, что список инструкций языка Python постоянно изменяется. Обычно это происходит после выхода очередных версий языка (появляются новые виды инструкций, а некоторые старые обновляются или вообще удаляются).

Что касается вызовов функций, генераторов и других выражений, то они тоже являются инструкциями, но при этом выделяются в особую группу за счет следующих особенностей:

7. они могут объединяться операторами в сколь угодно длинные конструкции, которые в итоге последовательных операций вычисляются интерпретатором и возвращаются в виде одного единственного значения (объекта некоторого типа);
8. они могут использоваться внутри других инструкций, являясь их составной частью.

Например, инструкция присваивания res = 5 + my_func(1, 2) содержит в правой части выражение, включающее вызов функции. Однако в результате вычислений в переменной сохранится ссылка лишь на один итоговый объект.

В отличие от выражений обычные инструкции в Python не могут использоваться там, где ожидается значение. Поэтому их нельзя использовать в качестве аргументов функций, правой части присваиваний, в качестве операций или возвращаемых значений. Например, нельзя в инструкции заменить вызов функции ее определением:

```
res = 5 + (def my_func(a, b): return a + b)
```

В этом случае интерпретатор возбудит исключение (выдаст ошибку).

По умолчанию интерпретатор Python выполняет инструкции последовательно одна за другой в порядке их следования в коде. Однако ряд специальных инструкций, называемых управляющими конструкциями, позволяют изменять ход выполнения программы. К ним инструкциям относятся:

- условные инструкции (if/else или if/elif/else), которые указывают интерпретатору на выполнение или пропуск других инструкций (в зависимости от значения условного выражения);
- инструкции циклов (for/in или while), которые указывают интерпретатору на многократное выполнение других инструкций;
- инструкции переходов (break, continue или return), которые указывают интерпретатору на выполнение перехода в другую часть программы;
- инструкции обработки исключений (try/except/else или try/except/finally), которые указывают интерпретатору на выполнение перехода в другую часть программы в случае возникновения ошибок.

3.1.2. Выражения в Python

Выражение (от англ. expression) – это любая единица исходного кода Python, которая может быть вычислена интерпретатором для получения значения. При этом литералы и переменные считаются выражениями сами по себе, а вызовы функций и некоторые другие конструкции вроде генераторов следует относить к выражениям из-за того, что они явно или неявно, но выдают результат. Выражения являются не менее важными элементами в Python, чем другие инструкции. Ведь большая часть того, что пишется в коде, является именно выражениями.

Выражения делятся на простые (первичные) и сложные. Простые выражения не могут быть разделены на еще более простые выражения. К ним относятся, например, литералы, инициализированные переменные, вызовы функций. Сложные выражения составляются из более простых при помощи различных операторов, которые позволяют выполнять такие операции, как сложение, вычитание, конкатенация и другие.

Пример программы, в которой показаны сложные выражения, приведен в листинге 3.2.

Листинг 3.2. Примеры сложных выражений

```
# Листинг 3.2
# Сложное выражение, включающее:
#   генератор списка,
#   арифметические выражения,
#   выражения сравнения,
#   логическое выражение
list_1 = [x * 10 for x in (0, 1, 2, 3) if x > 0 and x != 3]
print('Список ->', list)

# Анонимная функция и ее вызов в арифметическом выражении
res = (lambda a, b: a + b)(list_1[0], list_1[1]) ** 2

# Выражение с вызовом функции
if res in [900, 1000] and type(res) is int:
    print('Результат ->', res)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.2. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.2).

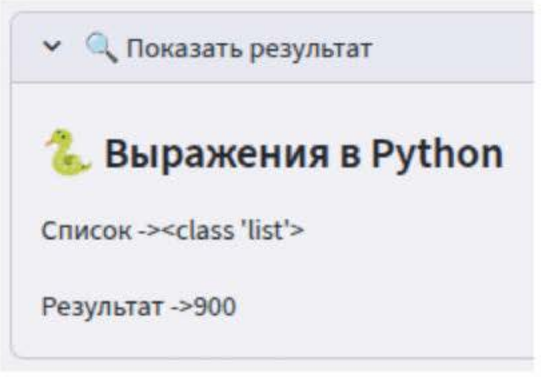


Рис. 3.2. Примеры сложных выражений

В данном примере присутствуют простые выражения:

- литерального значения -> 10, (0, 1, 2, 3);
- выражения переменных -> x, res;
- выражения вызовов функций -> type(res);
- вызов анонимной функции -> («lambda a, b: a + b)(li[0], li[1])»).

Посредством операторов эти простые выражения объединяются в более длинные смешанные выражения. Однако все эти примеры далеко не предел, так как выражения могут использоваться везде, где интерпретатор ожидает получить значение. Выражения могут выглядеть довольно длинными и сложными, но в результате вычислений на выходе будет получено одно результирующее значение.

В последующих разделах будут более подробно рассмотрены все виды выражений и особенности их использования. Сейчас же, просто перечислим основные выражения, дав краткие пояснения их назначения.

Выражения литерального значения – это литералы в коде программы, например:

- 'Иванов' -> строка,
- 10 -> число,
- ['Петров', 10] -> список,
- list -> объект.

Выражения переменных – обычные инициализированные переменные, например, «res» в выражении «res in [900, 1000]» (здесь переменная инициализирована, а значит, обращение к ней возвращает значение, сохраненное в памяти под данным именем).

Арифметические выражения – это выражения, в которых используются знакомые всем математические операторы (например, «+», «*», «-», «/») в выражении «a + b*3 – d / 8».

Условные выражения или выражения сравнения – используются операторы сравнения, например, a <= b.

Выражения с логическими операторами – это выражения, в которых используются логические операторы (например, «>», «and», «<» в выражении «a > b and a < c»).

Выражения с побитовыми операторами – это выражения, в которых используются побитовые операторы (например, «&» в выражении «a&b»).

Выражения с операторами принадлежности – это выражения, в которых используются операторы принадлежности (членства) «in» и «not in» (например, «in» в выражении «5 in [1, 2, 3]»).

Выражения с операторами идентичности – это выражения, в которых используются операторы идентичности «is» и «is not» (например, «is not» в выражении «5 is not '5'»).

Вызовы функций и методов – это тоже выражения, так как они явно или неявно возвращают результат (например, «len('Иванов')»).

Генераторные выражения – это выражения, которые способные генерировать данные: списки, словари, множества (например, выражение «{simbol for simbol in '123'}»).

lambda-выражения — это выражения, предназначенные для создания анонимных функций. Это особый синтаксис, который позволяет встроить функцию в строку программного кода (например, «5**2 + (lambda x, y: x - y)(5, 2)»).

Выражения присваивания – это выражения, в которых используется моржовый (walrus) оператор «:=», появившийся в Python начиная с версии 3.8. Этот оператор позволяет присваивать значения непосредственно в выражении (например, «x = 5 + (y:=5)»).

Трехместные выражения «if/else» – эти конструкции позволяют использовать в выражениях условную инструкцию «if/else» в рамках одно строки программного кода (например, «x = True if y > 0 else False»).

3.1.3. Операторы в Python

Операторы (от англ. operators) – это элементы программного кода, которые используются для указаний программе выполнять определенные действия в выражениях. Проще говоря, операторы – это символьные обозначения команд, с помощью которых выполняются операции.

Операции (от англ. operations) – это те действия, которые мы выполняем над переменными и объектами.

Как было отмечено в предыдущих разделах, для составления выражений используются операторы, то есть некие символы, которые позволяют сообщить интерпретатору о тех действиях, которые нужно выполнить над объектами (фактически над данными). Некоторые операторы и обозначающие их символы нам известны еще со школы: «+» - сложение, «-» - вычитание, «*» - умножение. С частью операций, используемых в программировании, еще предстоит познакомиться.

Примерами выражений с использованием операторов могут служить: «A + B», «a != 4», «z > d», «-b», «(b + 7)/(5 - 3**2)» и т.п. Здесь символы и комбинации символов «+», «!=», «**», «-», «/», «>» и есть операторы. При этом выражения или значения, расположенные по сторонам операторов, называют операндами.

Большинство операторов обрабатывают свои операнды в определенном направлении. Это свойство принято называть ассоциативностью. Одни операторы выполняют обработку своих операндов слева направо, обладая левой ассоциативностью, другие обладают правой ассоциативностью, выполняя обработку в обратном направлении, то есть справа налево. Также существуют операторы, не имеющие ассоциативности.

Кроме того, операторы могут отличаться своим приоритетом, который определяет порядок (очередность) их выполнения в ходе вычисления значения выражения. Первыми обрабатываются операторы с более высоким приоритетом, затем операторы с более низким приоритетом и так далее по мере убывания их приоритетности. Если приоритет операторов одинаков, они группируются и обрабатываются либо согласно их ассоциативности, либо в соответствии с установленным для них порядком (это касается операторов, у которых ассоциативность отсутствует). Также отметим, что для принудительного указания порядка выполнения операторов используются круглые скобки.

Пример программы, в которой показана последовательность действия операторов по умолчанию и, при наличии скобок, приведен в листинге 3.3.

Листинг 3.3. Примеры последовательности действия операторов по умолчанию

```
# Листинг 3.3
print("Приоритет оператора '*' выше, чем у '+'")
a = 6 + 9 * 2
print('6 + 9 * 2 ->', a)

print("Скобки сменили приоритет, теперь '+' выше, чем у '*'")
b = (6 + 9) * 2
print('(6 + 9) * 2 ->', b)
print('-'*50)

c = 2
```

```
print('Приоритет слева-> направо.')
print('a < b > c ->', a < b > c)
print("Скобки сменили приоритет, теперь налево <- справа.")
print('a < (b > c) ->', a < (b > c))
print('-'*50)

# Согласно принятому порядку это тоже самое, что и c = (a = b)
print("Здесь по умолчанию приоритет налево <- справа.")
print("Как будто присутствуют скобки c = (a = b).")
print('b=', b)
print('a=', a)
c = a = b
print('c = a = b ->', a)
print('-'*50)

print('Операторы "-" и "-" имеют равный приоритет.')
print('По умолчанию левая ассоциативность.')
print('Как будто присутствуют скобки  d = (5 - 6) - 7')
d = 5 - 6 - 7
print('5 - 6 - 7 ->', d)
print('-'*50)

print('По умолчанию левая ассоциативность.')
print('Как будто присутствуют скобки  d = (0 and 0) or 1')
d = 0 and 0 or 1
print('0 and 0 or 1 ->', d)
print('-'*50)

print('Теперь правая ассоциативность.')
print('Так как операция в скобках выполняется первой.')
d = 0 and (0 or 1)
print('0 and (0 or 1) ->', d)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.3. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Приоритет оператора '*' выше, чем у '+'
6 + 9 * 2 -> 24
Скобки сменили приоритет, теперь '+' выше, чем у '*'
(6 + 9) * 2 -> 30
--------------------------------------------------
Приоритет слева-> направо.
a < b > c -> True
Скобки сменили приоритет, теперь налево <- справа.
a < (b > c) -> False
```

```
-------------------------------------------------
Здесь по умолчанию приоритет налево <- справа.
Как будто присутствуют скобки c = (a = b).
b= 30
a= 24
c = a = b -> 30
-------------------------------------------------
Операторы "-" и "-" имеют равный приоритет.
По умолчанию левая ассоциативность.
Как будто присутствуют скобки  d = (5 - 6) - 7
5 - 6 - 7 -> -8
-------------------------------------------------
По умолчанию левая ассоциативность.
Как будто присутствуют скобки  d = (0 and 0) or 1
0 and 0 or 1 -> 1
-------------------------------------------------
Теперь правая ассоциативность.
Так как операция в скобках выполняется первой.
0 and (0 or 1) -> 0
```

Основные операторы языка Python, использующиеся в выражениях, в порядке убывания их приоритетов, представлены в табл.3.1.

Таблица 3.1

Основные операторы языка Python.

Оператор	Описание	Ассоциативность
**	Возведение в степень	Справа налево
~ + -	Побитовое отрицание (инверсия битов) Унарный плюс Унарный минус	Справа налево
* / % //	Умножение Деление Деление по модулю Целочисленное деление	Слева направо
+ -	Сложение Вычитание	Слева направо
<< >>	Побитовый сдвиг влево Побитовый сдвиг вправо	Слева направо
&	Побитовое and (И)	Слева направо
^	Побитовое xor (исключающее ИЛИ)	Слева направо
|	Побитовое or (ИЛИ)	Слева направо
< <= > >= == !=	Меньше Меньше или равно Больше Больше или равно Равно Не равно	Отсутствует

is, is not	Операторы тождественности	Отсутствует
in, not in	Операторы членства	Отсутствует
not	Логическое отрицание (НЕ)	Справа налево
and	Логическое И	Слева направо
or	Логическое ИЛИ	Слева направо
:=	Моржовый оператор (выражение присваивания)	Справа налево

В данной таблице, чем ниже приоритет, тем ниже находится строка с соответствующими операторами. Если операторы имеют одинаковый приоритет, они перечисляются в одной и той же строке таблицы.

У операторов Python есть еще одна особенность, связанная с операторами «*» и «**». Оператор «*» чаще всего ассоциируется с операцией умножения, а «**» - с возведение в степень. В этом случае они стоят между двумя переменными (например, f * h, или x**2). Но в Python звездочки могут иметь и друге назначение. Достаточно часто можно встретить инструкцию, когда звездочки являются префиксом, стоят рядом лишь с одной переменной и не отделяется от нее пробелом. Например, инструкция *args, **kwargs при работе с функциями, или инструкция *name, где звездочка стоит перед именем переменной. Более детально эта функция звездочек будет рассмотрена в следующем разделе.

Хорошей практикой программирования считается использование скобок, принудительно определяющих порядок выполнения операций. Это облегчит сопровождение кода и поиск ошибок, как при наличии сложных выражений, так и в случае изменения поведения операторов при переходе от одной версии Python к другой.

Большинство операторов, приведенных в таблице, являются бинарными, т.е. используются с двумя операндами (например, «+», «*», «!=»). Однако есть операторы, которые работают только с одним операндом, их называют унарными. Примером может служить - (оператор отрицания или унарный минус), который применяется для смены знака числа на противоположный.

Фрагменты кода с присваиванием значений переменным в Python принято считать инструкциями присваивания. Например, фрагменты кода «d = 5» или «s += 7**2» относятся к инструкциям присваивания. Однако одна форма выражения присваивания в Python 3.8 и последующих версиях все-таки имеется. Формируется она при помощи введенного моржового оператора «:=», который может использоваться не только в инструкциях, но и внутри обычных выражений. Например, инструкция присваивания «li = [a := 1, b := 2]» включает также и два выражения присваивания.

3.1.4.Особенности операторов * и ** в Python

Операторы «*» и «**» встречаются в питоне очень часто. Традиционно в языках программирования они обозначают операции умножения и возведения в степень. Однако иногда эти операторы встречаются в инструкциях, которые непонятны не только новичкам, но и опытным программистам, переходящим на Python с других языков программирования. В данном разделе мы будем говорить про операторы - префиксы «*» и «**», которые используются перед переменными.

В сферу применения рассматриваемых операторов входит:

- операторы «*» и «**» - передача аргументов в функцию;
- операторы «*» и «**» - захват аргументов, переданных в функцию;
- оператор «*» - принятие аргументов, содержащих только ключевые слова;
- оператор «*» - захват элементов во время распаковки кортежа (tuple);
- оператор «*» - распаковка итерируемых объектов в списке или кортеже;
- оператор «**» - распаковка словарей в других словарях.

Использование этих операторов в функциях будет рассмотрено позднее в соответствующем разделе, а здесь рассмотрим их возможности для работы с последовательностями.

Пример программы, в которой операторы «*» и «**» используются для распаковки словарей, кортежей и списков, приведен в листинге 3.4.

Листинг 3.4. Операторы «*» и «» для распаковки словарей, кортежей и списков**

```
# Листинг 3_4
print('Исходный список фруктов')
fruits = ['Лимоны', 'Апельсины', 'Мандарины', 'Яблоки']
print('Фрукты ->', fruits)
print('----- Распаковываем список фруктов -----')
f_1, f_2, *f_3 = fruits
print('f_1->', f_1)
print('f_2->', f_2)
print('f_3->', f_3)
print('Тип данных fruits->', type(fruits))
print('Тип данных f_3->', type(f_3))
print('=' * 30)

print('Исходный кортеж овощей')
vegetables = ('Огурцы', 'Помидоры', 'Капуста', 'Морковь')
print('Овощи ->', vegetables)
print('----- Распаковываем кортеж овощей -----')
ov_1, ov_2, *ov_3 = vegetables
print('ov_1->', ov_1)
print('ov_2->', ov_2)
print('ov_3->', ov_3)
print('Тип данных vegetables->', type(vegetables))
print('Тип данных ov_3->', type(ov_3))
print('=' * 30)

print('Исходные словари')
d_1 = {1: "Пн", 2: "Вт", '3': "Ср"}
d_2 = {4: "Чт", 5: "Пт", '6': "Сб", 7: "Вс"}
print('d_1 ->', d_1)
print('d_2 ->', d_2)
print('----- Объединенный словарь -----')
all_d = {**d_1, **d_2}
print(all_d)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.4. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Исходный список фруктов
Фрукты -> ['Лимоны', 'Апельсины', 'Мандарины', 'Яблоки']
----- Распаковываем список фруктов -----
f_1-> Лимоны
f_2-> Апельсины
f_3-> ['Мандарины', 'Яблоки']
Тип данных fruits-> <class 'list'>
Тип данных f_3-> <class 'list'>
==============================
Исходный кортеж овощей
Овощи -> ('Огурцы', 'Помидоры', 'Капуста', 'Морковь')
----- Распаковываем кортеж овощей -----
ov_1-> Огурцы
ov_2-> Помидоры
ov_3-> ['Капуста', 'Морковь']
Тип данных vegetables-> <class 'tuple'>
Тип данных ov_3-> <class 'list'>
==============================
Исходные словари
d_1 -> {1: 'Пн', 2: 'Вт', '3': 'Ср'}
d_2 -> {4: 'Чт', 5: 'Пт', '6': 'Сб', 7: 'Вс'}
----- Объединенный словарь -----
{1: 'Пн', 2: 'Вт', '3': 'Ср', 4: 'Чт', 5: 'Пт', '6': 'Сб', 7: 'Вс'}
```

Здесь на первом шаге был создан список из фруктов. При распаковке этого списка значения первых двух элементов списка присвоены переменным f_1 и f_2, а оставшиеся элементы списка распакованы в новый список f_3. Контрольный вывод типа данных показал, что при распаковке списка тип данных не изменился.

На втором шаге был создан кортеж из овощей. При распаковке этого кортежа значения первых двух элементов кортежа присвоены переменным ov_1 и ov_2, а оставшиеся элементы кортежа распакованы в новый список ov_3. Контрольный вывод типа данных показал, что действительно исходный тип данных являлся кортежем:

```
Тип данных vegetables-> <class 'tuple'>
```

А распакованная часть этого кортежа трансформировалась в список:

```
Тип данных ov_3-> <class 'list'>
```

На третьем шаге было создано два словаря с днями недели – d_1 и d_2. Затем эти словари были распакованы и объединены в один словарь – d_3.

Операторы «*» и «**» в Python это просто «синтаксический сахар». Конечно часть из того, что можно сделать с их помощью, можно достичь и другими способами, но все они, как

правило, более громоздкие и ресурсоемкие. А некоторые из возможностей, предоставляемых звездочками, просто нельзя реализовать иначе: например, принять переменное количество аргументов в функцию. Но использование звездочек в функциях будет рассмотрено позднее в одной из глав.

3.1.5. Краткие итоги раздела 3.1

7. Инструкции – это команды на языке Python, которые сообщают интерпретатору последовательность действий, необходимых для достижения поставленных целей. По умолчанию интерпретатор выполняет инструкции последовательно одна за другой в порядке их следования в коде. Если нужно изменить выполнение хода программы по умолчанию, используются управляющие конструкции: условные инструкции, циклы, инструкции переходов или инструкции обработки исключений. Инструкции только управляют ходом выполнения программы и не вычисляют значения.
8. Выражение – это любая единица программного кода Python, которая может быть использована интерпретатором для получения значения. Сложные выражения составляются из более простых при помощи различных операторов, которые позволяют выполнять над объектами такие операции как сложение, вычитание, сравнение и т.д. Но каким бы сложным не было выражение, в результате последовательных вычислений на выходе интерпретатор получит результирующе значение.
9. Выражения практически всегда входят в состав каких-либо инструкций, а использование инструкций в выражениях недопустимо.
10. Операторы представляют собой символьные обозначения команд, с помощью которых выполняются операции над объектами. Выражения или значения, расположенные по сторонам операторов, называют операндами. Если оператор имеет два операнда, его называют бинарным. Если же оператор имеет только один операнд, его называют унарным.
11. Большинство операторов обрабатывают свои операнды в определенном направлении. Это свойство называют ассоциативностью. Одни операторы выполняют обработку своих операндов слева направо (левая ассоциативностью), другие справа налево (правая ассоциативность). Существуют и операторы, не имеющие ассоциативности. Левой ассоциативностью, в основном обладают бинарные операторы, правая ассоциативность присуща унарным операторам, а операторы сравнения вообще не обладают ассоциативностью.
12. Важным является приоритет выполнения операторов. Операторы с большим приоритетом обрабатываются первыми.
13. Если приоритет операторов одинаков, то они выполняются либо в очередности согласно их ассоциативности, либо в заданном для них принудительным порядке, который задается скобками. Рекомендуется всегда использовать скобки, определяя принудительный порядок выполнения операций.
14. В Python операторы «*» и «**» в разных инструкциях имеют совершенно разное назначение. Это обстоятельство нужно учитывать при написании программ.

3.1.6. Вопросы для самоконтроля к разделу 3.1



- Какой из фрагментов программного кода относится к инструкциям:

- def my_func(a,b,c):;
- s = 2**3;
- d = 1 + True;
- A += 1.

- Какой из фрагментов программного кода относится к выражениям:
  - d = {1: 'Иванов', 2: 'Петров'};
  - def my_func(a,b,c):;
  - if a > b:;
  - for i in num:
- Какое значения вернет выражение: A = 5 + 5 * 2?
  - A = 15;
  - A = 10;
  - A = 25;
  - A = 20.
- Какое значения вернет выражение: A = (5 + 5) * 2?
  - A = 20;
  - A = 15;
  - A = 10;
  - A = 25.

3.2.Инструкции присваивания в Python

Операторы присваивания предназначены для связывания имен со значениями и для изменения атрибутов или элементов изменяемых объектов. Оператор присваивания связывает переменную с объектом и набором (списком) объектов. Оператор присваивания обозначается «=». В языке Python существуют следующие формы (разновидности) инструкции присваивания:

9. обычная (каноническая) форма;
10. комбинированные инструкции присваивания;
11. позиционное присваивание;
12. групповое присваивание одного значения;
13. выражения присваивания.

3.2.1. Каноническая форма присваивания

В Python, как и во многих других языках программирования, каноническая форма операции присваивания имеет вид a = b, где слева от оператора присваивания записывается целевое имя переменной или компонент объекта, а в качестве правого операнда выступает произвольное выражение, которое в результате вычислений дает объект.

Пример программы, в которой показано использование простейшей канонической формы присваивания, приведен в листинге 3.5.

Листинг 3.5. Использование простейшей канонической формы присваивания

```
# Листинг 3.5
a = 5 + 0.3 + 5**2
print('Присваивание из выражения ->', a)

b = [1, 1, 2]
print('Присваивание из списка ->', b)

c = ('A', 'B', 'C')
print('Присваивание из кортежа ->', c)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.5. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.3).



Рис. 3.3. Использование простейшей канонической формы присваивания

Как видим, присваивание действительно выполняется достаточно просто. Однако при этом нужно обратить внимание на следующие особенности присваивания в Python.

- Инструкции присваивания всегда сохраняют ссылки на объекты, например, в переменных или в элементах структур данных, и никогда не создают копии присваиваемых объектов. Как следствие, переменные в Python больше напоминают указатели, чем области хранения данных.
- Объявлять переменные заранее нежелательно, так как интерпретатор создает переменные в момент присваивания им значений (то есть ссылок на объекты). После инициализации, когда имя переменной будет встречаться в выражениях, оно будет автоматически замещаться объектом, на который ссылается данная переменная. Если же интерпретатор встретит в программе неинициализированную переменную, он возбудит исключение вместо того, чтобы вернуть какое-либо значение по умолчанию.
- Некоторые инструкции пусть и неявно, но тоже выполняют операцию присваивания. Примерами могут служить: импорт модуля, определение функции или класса, указание переменной в цикле «for», передача аргументов функции. В любом случае результатом явных или неявных инструкций присваивания будет связывание имен со ссылками на объекты.

Необходимо помнить, что в программировании знак равно (=) обычно означает не равенство двух величин, а присваивание значения переменной. Если же нужно указать на равенство двух значений, то используется дублирование знака равно (==).

3.2.2. Комбинированные инструкции присваивания

В дополнение к базовой инструкции присваивания в Python имеется и ряд комбинированных инструкций присваивания, которые объединяют операцию присваивания с другой операцией. Например, запись x + = y является сокращенной записью инструкции присваивания x = x + y , в которой к значению переменной «x» прибавляется значение переменной «y», а результат присваивается переменной «x».

Пример программы, в которой показано использование комбинированной формы присваивания, приведен в листинге 3.6.

Листинг 3.6. Использование комбинированной формы присваивания

```
# Листинг 3.6
# Присвоили начальное значение
a = 3
print('a = 3 ->', a)

a += 6  # Это равнозначно a = a + 6
print('a += 6 ->', a)

a *= 2  # Это равнозначно a = a*2
print('a *= 2 ->', a)

a /= 2  # Это равнозначно a = a/2
print('a /= 2 ->', a)

a **= 2  # Это равнозначно a = a**2
print('a **= 2 ->', a)

a %= 2  # Это равнозначно a = a%2
print('a %= 2 ->', a)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.6. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.4).

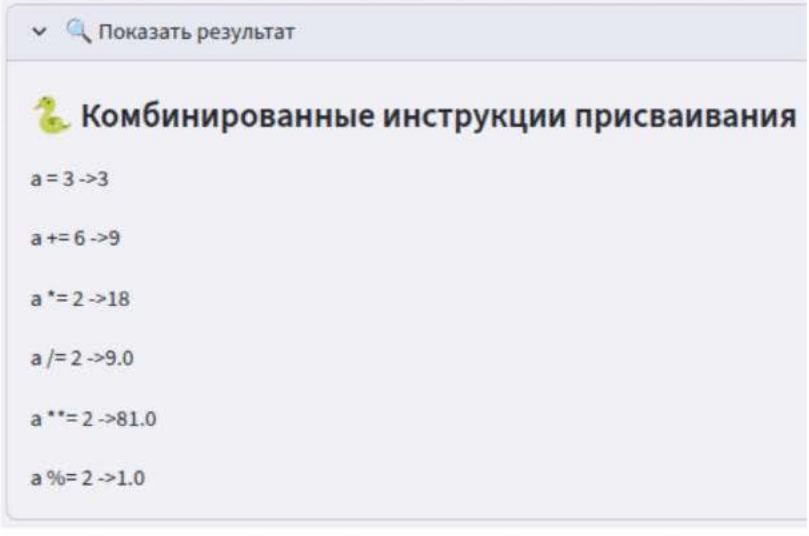


Рис. 3.4. Использование комбинированной формы присваивания

Таким образом, комбинированная инструкция присваивания объединяет в себе выражение и присваивание, являясь, по сути, краткой формой записи кода. И хотя инструкции num += 1

и num = num + 1 дадут один и тот же результат, первая из них выглядит явно компактнее. Кроме того, если объект, находящийся справа относится к категории изменяемых объектов и поддерживает указанную операцию, то комбинированная инструкция присваивания будет выполняться быстрее за счет изменения самого объекта вместо создания и изменения его копии.

Пример программы, в которой показана разность времени выполнения операций присваивания при использовании канонической и комбинированной формы присваивания, приведен в листинге 3.7.

Листинг 3.7. Использование канонической и комбинированной формы присваивания

```
# Листинг 3.7
# Импортируем модуль time
import time

# Получаем стартовое значение времени
t_1 = time.time()

# Запускаем цикл 10 млн. раз
for i in range(0, 10 ** 7):
    # Обычное присваивание
    li = [0, 1]
    li = li + [2, 3]
    li = li + [4, 5]
    del li

# Получаем конечное значение времени
t_2 = time.time()
t3 = round((t_2 - t_1), 13)
print('Каноническая форма присваивания ->', t3, 'сек.')

# Тоже самое, но для комбинированного присваивания

# Получаем стартовое значение времени
t_1 = time.time()

# Запускаем цикл 10 млн. раз
for i in range(0, 10 ** 7):
    # Комбинированное присваивание
    li = [0, 1]
    li += [2, 3]
    li += [4, 5]
    del li

# Получаем конечное значение времени
t_2 = time.time()
t3 = round((t_2 - t_1), 13)
print('Комбинированная форма присваивания ->', t3, 'сек.')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.7. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.5).

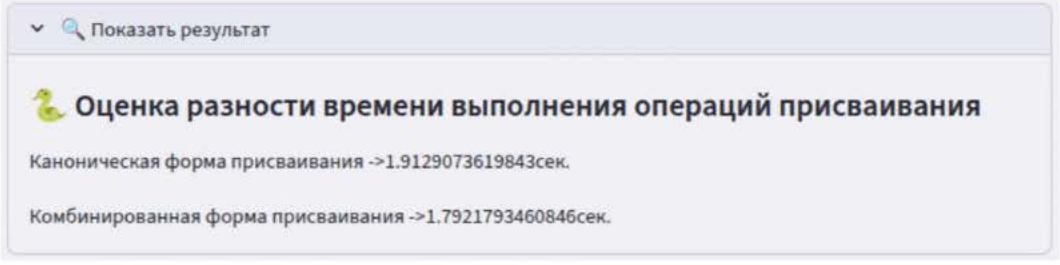


Рис. 3.5. Использование канонической и комбинированной формы присваивания

В этой программе используется метод time() одноименного модуля time из стандартной библиотеки Python. Данный метод возвращает время в секундах с начала эпохи. В операционных системах Unix за начало эпохи принимается 1 января 1970 года 00:00:00 (UTC). Но поскольку в программах в основном используются интервалы времени между двумя событиями, а не время, прошедшее с начала эпохи, знать дату начала эпохи какой-либо конкретной операционной системы необязательно, так как разница между двумя временными точками для всех платформ получается совершенно одинаковой.

Ниже перечислены основные инструкции присваивания, комбинированные с арифметическими операторами:

- «a += b» – тоже, что и a = a + b;
- «a -= b» – тоже, что и a = a - b;
- «a *= b» – тоже, что и a = a * b;
- «a **= b» – тоже, что и a = a**b;
- «a /= b» – тоже, что и a = a / b;
- «a //= b» – тоже, что и a = a // b;
- «a %= b» – тоже, что и a = a % b.

3.2.3. Позиционное присваивание

Еще одной формой присваивания в Python является позиционное присваивание. При этом группе объектов, которые находятся слева от оператора присваивания, присваиваются значения, расположенные справа от оператора присваивания. При этом присваивание выполняется слева направо согласно местоположениям объектов и присваиваемых значений.

Пример программы, в которой показано позиционное присваивание, приведен в листинге 3.8.

Листинг 3.8. Позиционное присваивание

```
# Листинг 3.8
print('Позиционное присваивание значений переменным "a" и "b"')
print(' ')
a, b = 1, 2
print('Выражение: a, b = 1, 2')
print('a ->', a, ', b ->', b)

# Позиционное присваивание из списка
a, b = [3, 4]
print('Выражение: a, b = [3, 4]')
print('a ->', a, ', b ->', b)

# Позиционное присваивание списку из списка
[a, b] = [4, 5]
print('Выражение: [a, b] = [4, 5]')
print('a ->', a, ', b ->', b)

# Позиционное присваивание строки списку
[a, b] = 'AB'
print("Выражение: [a, b] = 'AB'")
print('a ->', a, ', b ->', b)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.8. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.6).

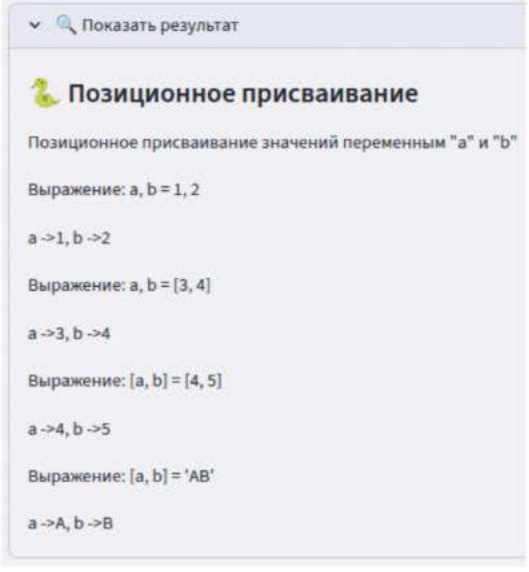


Рис. 3.6. Позиционное присваивание

Как видим, данная форма присваивания позволяет смешивать последовательности разных типов. Здесь главное, чтобы количество присваиваемых значений совпадало с количеством переменных, которым эти значения будут присваиваться. Но и эта проблема может быть решена за счет имеющейся возможности использования в присваивании расширенного синтаксиса распаковывания последовательностей (добавление символа «*»).

Пример программы, в которой показано использования в присваивании расширенного синтаксиса распаковывания последовательностей, приведен в листинге 3.9.

Листинг 3.9. Использование в присваивании распаковывания последовательностей

```
# Листинг 3.9
# Используем синтаксис распаковывания последовательности
a, b, *c = [1, 2, 3, 4, 5]
print('Выражение: a, b, *c = [1, 2, 3, 4, 5]')
print('Итог: a->', a, ', b ->', b, ', c ->', c)
print('-'*40)

*a, b, c = [1, 2, 3, 4, 5]
print('Выражение: *a, b, c = [1, 2, 3, 4, 5]')
print('Итог: a->', a, ', b ->', b, ', c ->', c)
print('-'*40)

a, *b, c = [1, 2, 3, 4, 5]
print('Выражение: a, *b, c = [1, 2, 3, 4, 5]')
print('Итог: a->', a, ', b ->', b, ', c ->', c)
print('-'*40)

# Граничный случай
a, b, *c = [1, 2, 3]
print('Выражение: a, b, *c = [1, 2, 3]')
print('Итог: a->', a, ', b ->', b, ', c ->', c)
print('-'*40)

# Еще один граничный случай
a, b, *c = [1, 2]
print('Выражение: a, b, *c = [1, 2]')
print('Итог: a->', a, ', b ->', b, ', c ->', c)
print('-'*40)

# Еще один граничный случай.
*c, = [1, 2]
print('Выражение: c, = [1, 2]')
print('Итог: c ==', c)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.9. Результаты работы модуля можно увидеть, открыв контейнер «Показать

результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Выражение: a, b, *c = [1, 2, 3, 4, 5]
Итог: a-> 1 , b -> 2 , c -> [3, 4, 5]
----------------------------------------
Выражение: *a, b, c = [1, 2, 3, 4, 5]
Итог: a-> [1, 2, 3] , b -> 4 , c -> 5
----------------------------------------
Выражение: a, *b, c = [1, 2, 3, 4, 5]
Итог: a-> 1 , b -> [2, 3, 4] , c -> 5
----------------------------------------
Выражение: a, b, *c = [1, 2, 3]
Итог: a-> 1 , b -> 2 , c -> [3]
----------------------------------------
Выражение: a, b, *c = [1, 2]
Итог: a-> 1 , b -> 2 , c -> []
----------------------------------------
Выражение: c, = [1, 2]
Итог: c == [1, 2]
```

Благодаря наличию имени со звездочкой все лишние значения автоматически помещаются в соответствующий список именно так, как и ожидается. Однако нужно не забывать, что можно использовать только одну переменную со звездочкой, а сама переменная должна принадлежать последовательности даже в том случае, если она будет одна (поставив запятую в инструкции *c, = [1, 2] мы автоматически получили кортеж и смогли избежать ошибки).

Следует добавить, что расширенная операция распаковывания последовательностей может использоваться не только в случаях явного присвоения значений, но также и в заголовках циклов, вызовах функций и других операциях неявного присвоения значений.

3.2.4. Групповое присваивание одного значения

В случае группового присваивания объект, расположенный справа, присваивается всем переменным группы. В результате несколько отдельных инструкций присваивания могут быть заменены одной эквивалентной и более компактной инструкцией.

Пример программы, в которой показано использования группового присваивания, приведен в листинге 3.10.

Листинг 3.10. Использование группового присваивания

```
# Листинг 3.10
# Обычное присваивание
a = "Иванов"
b = a
c = b
print('Выражение: a = "Иванов", b = a, c = b')
```

```
print('Итог: a ->', a, ', b ->', b, ', c ->', c)
print('-'*40)

# Групповое присваивание
c = b = a = "Иванов"
print('Выражение: c = b = a = "Ok"')
print('Итог: a ->', a, ', b ->', b, ', c ->', c)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.10. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.7).

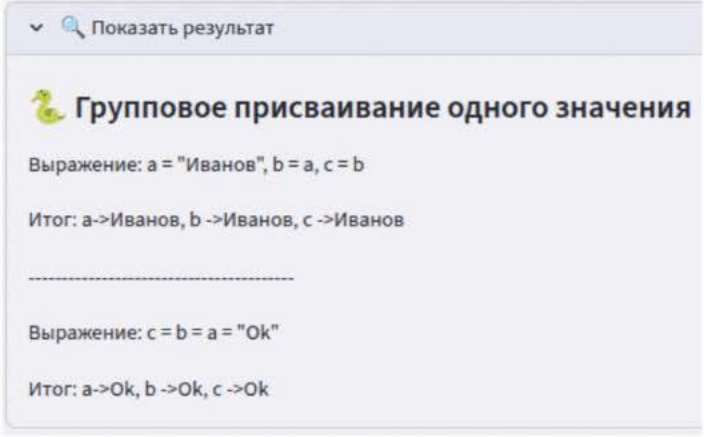


Рис. 3.7. Использование группового присваивания

Следует иметь в виду, что при групповом присваивании в памяти создается всего лишь один объект, разделяемый всеми переменными группы. Поэтому такая форма присваивания будет полезна для неизменяемых объектов, например, чисел или строк.

А вот при использовании изменяемых объектов типа списков или словарей нужно быть осторожными, так как изменение объекта через один из элементов группы будет оказывать влияние и на другие элементы этой группы. Это связано с тем, что переменные в Python хранят не сами объекты, а ссылки на них. А раз так, то даже после изменения объекта через одну из ссылок все остальные по-прежнему будут указывать на тот же, пусть и модифицированный, объект.

Пример программы, в которой показано отличие использования группового присваивания для неизменяемых и изменяемых объектов, приведен в листинге 3.11.

Листинг 3.11. Отличие использования группового присваивания для неизменяемых и изменяемых объектов

```
Листинг 3.11
# Используем число (неизменяемый объект)
print('Для неизменяемого объекта')
c = b = a = 111
print('a ->', a, ', b ->', b, ', c ->', c)

# Изменим b и c
b = 1
c = 2
print('a ->', a, ', b ->', b, ', c ->', c)
print('-'*40)

# Используем список (изменяемый объект)
print('Для изменяемого объекта')
c = b = a = []
print('a ->', a, ', b->', b, ', c ->', c)

# Пробуем изменить b и c
b += [1]
c += [2]
print('a ->', a, ', b ->', b, ', c ->', c)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.11. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.8).

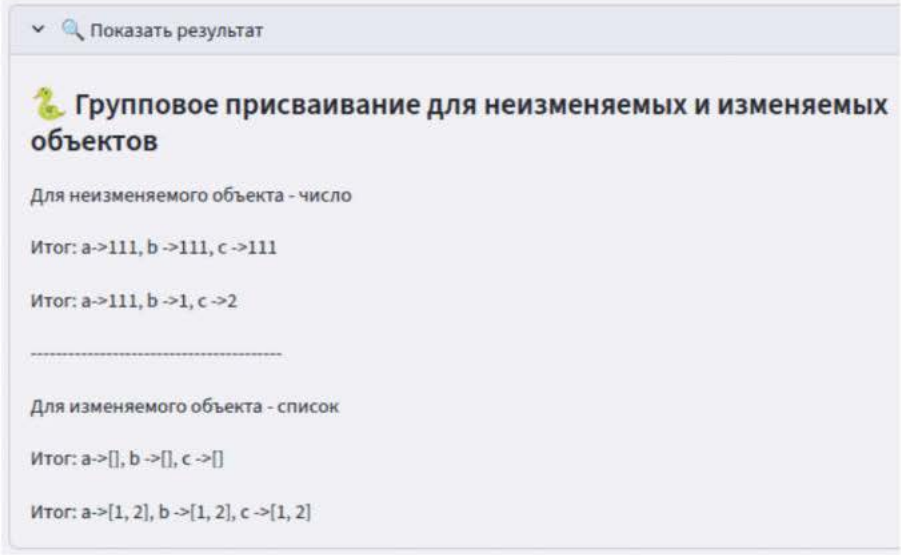


Рис. 3.8. Групповое присваивание для неизменяемых и изменяемых объектов

В данном примере при использовании числа мы на первом шаге всем трем переменным присвоили неизменяемый объект - число 111, в результате чего все три переменные получили ссылки на него. Затем на втором шаге двум переменным было присвоены новые значения (c=1, d=2). Поскольку числа относятся к неизменяемым объектам, то для чисел 1 и 2 были созданы новые объекты, и переменные c и b получили на них ссылки. После этого все три переменные стали иметь свои уникальные значения (рис. 3.9).

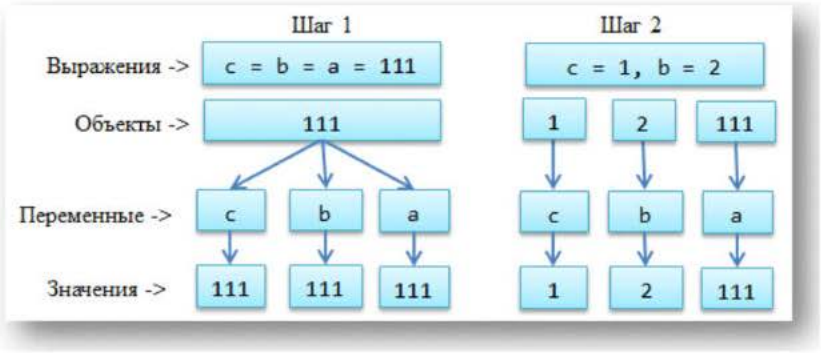


Рис. 3.9. Групповое присваивание неизменяемых значений

Затем при использовании списка мы на первом шаге всем трем переменным присвоили изменяемый объект - пустой список ([]), в результате чего все три переменные получили ссылки на него. Затем на втором шаге двум переменным было присвоены новые значения (c=1, d=2). Поскольку списки относятся к изменяемым объектам, то для чисел 1 и 2 новые объекты не создавались, а данные были внесены в существующий изменяемый объект. После этого все три переменные стали иметь новое (измененное) значение, полученное из существующего объекта (рис. 3.10).

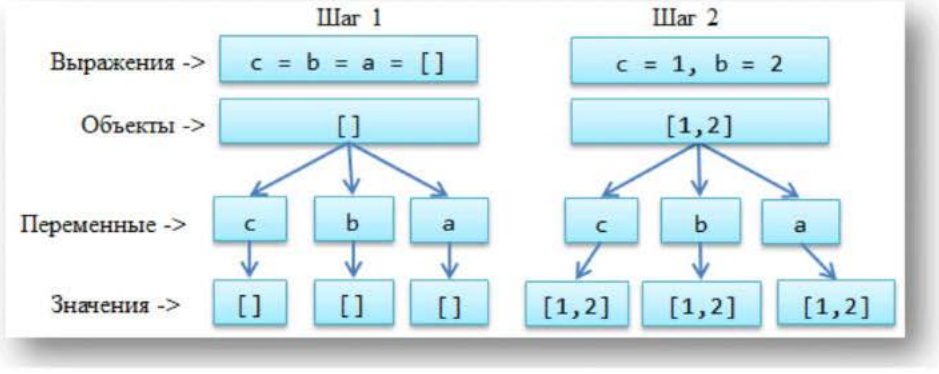


Рис. 3.10. Групповое присваивание изменяемых значений

Таким образом, во избежание проблем с изменяемыми объектами, их инициализацию лучше производить в отдельных инструкциях.

3.2.5. Выражения присваивания (оператор «морж»)

Все рассмотренные нами формы присваивания относятся к инструкциям. Но их запрещено использовать в выражениях, что иногда может быть весьма полезным. Поэтому, начиная с версии Python 3.8, было решено ввести новую конструкцию «идентификатор:= значение» с возможностью использования в выражениях. Конструкция получила название выражение присваивания или именованное выражение, а оператор «:=» стал неофициально называться оператор «морж». Такое название он получил из-за того, что его написание напоминает мордочку моржа с клыками.

Пример программы, в которой показано использование оператора «морж», приведен в листинге 3.12 (этот модуль работает только в Python версии 3.8 и выше).

Листинг 3.12. Использование оператора «морж»

```
# Листинг 3.12
print('Оператор "морж" в Python с версии 3.8')
# В выражении инициализируются сразу 3 переменные.
a = (b := 3) + (c := 5)
print('Выражение: a = (b := 3) + (c := 5)')
print('Итог:', 'a ->', a, ', b ->', b, ', c ->', c)
print('-'*40)

# Список функций (для однозначности используем скобки).
list_1 = [(x_1 := lambda y: y**2), (x_2 := lambda y: y**3)]
print('Выражение: list_1 = [(x_1 := lambda y: y**2), (x_2 := lambda y:
y**3)]')
print('-'*40)

z = 3
print('Результаты вычислений при z=3')
print('Список: ', 'list_1[0] ->', list_1[0](z), ', list_1[1] ->',
list_1[1](z))
print('Значения переменных: ', 'x_1 ->', x_1(z), ', x_2 ->', x_2(z))
print('-'*40)

z = 5
print('Результаты вычислений при z=5')
print('Список: ', 'list_1[0] ->', list_1[0](z), ', list_1[1] ->',
list_1[1](z))
print('Значения переменных: ', 'x_1 ->', x_1(z), ', x_2 ->', x_2(z))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.12. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.11).

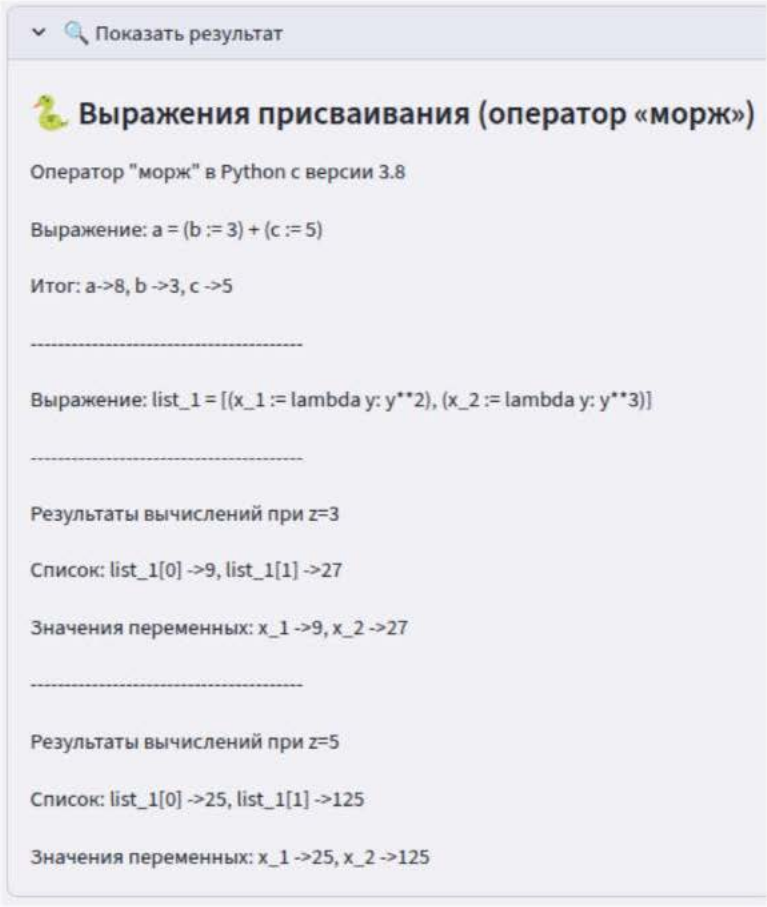


Рис. 3.11. Использование оператора «морж»

Из данного примера видно, что использование оператора «морж» позволяет сократить программный код. Так одна строка кода: a = (b := 3) + (c := 5) аналогична трем строкам кода, приведенных ниже:

```
b = 3
c = 5
a = b + c
```

А строка кода: list = [(x_1 := lambda y: y**2), (x_2 := lambda y: y**3)] выполняет целую группу операций:

- создает список – list;
- создает две lambda-функции (lambda y: y**2 и lambda y: y**3);
- переменной x_1 присваивает значение lambda–функции «lambda y: y**2» (возводит в квадрат значение аргумента функции – y);
- переменной x_2 присваивает значение lambda–функции «lambda y: y**3» (возводит в куб значение аргумента функции – y).

С одной стороны такая конструкция выглядит довольно сложно, но работает достаточно эффективно.

У оператора «морж» есть одна особенность – выражение, в котором он используется, обязательно должно быть заключено в скобки. Например, при выполнении инструкции «x= y := 5» интерпретатор выдаст ошибку, нужно обязательно идентификатор с присваиваемым ему значением заключать в скобки: «x = (y := 5)». Использование скобок с оператором «морж» следует взять на вооружение, так как это поможет избежать досадных ошибок.

3.2.6. Краткие итоги раздела 3.2

8. В Python инструкции присваивания всегда сохраняют ссылки на объекты, например, в переменных или в элементах структур данных, и никогда не создают копии присваиваемых объектов.
9. Классическая форма присваивания имеет вид «идентификатор=значение», что соответствует синтаксису большинства языков программирования. Однако в Python доступны и другие формы инструкций присваивания:
    - комбинированные инструкции присваивания (например, y **= 3),
    - позиционное присваивание (например, x, y = 3, 5),
    - групповое присваивание одного значения нескольким элементам (например, a=b=c=3).
10. При групповом присваивании одного значения в памяти создается всего лишь один объект, разделяемый всеми переменными группы. Поэтому такая форма присваивания будет полезна для неизменяемых объектов, например, чисел или строк. А вот при использовании изменяемых объектов типа списков или словарей нужно быть осторожными, так как изменение объекта через один из элементов группы будет оказывать влияние и на все другие элементы группы.
11. В Python, начиная с версии 3.8, была введена специальная конструкция «идентификатор:=значение», получившая название выражения присваивания и предназначенная для присваивания значений внутри выражений (например, a = (b := 3) + (c := 5)). Оператор, который используется в такой конструкции присваивания, получил экзотическое название – оператор «морж».

3.2.7. Вопросы для самоконтроля к разделу 3.2



9. Групповым присваиванием 3-м переменным было присвоено следующее значение: a=b=c=1. Затем двум переменным были присвоены другие значения: b=2, c=3. Какие в итоге значения будут иметь переменные a, b и c:
    - a=1, b=2, c=3;
    - a=1, b=1, c=1;
    - a=3, b=1, c=2;
    - a=3, b=2, c=1.
10. Групповым присваиванием 3-м переменным было присвоено следующее значение: a=b=c=[]. Затем этим переменным были присвоены другие значения: a=1, b=2, c=3. Какие в итоге значения будут иметь переменные a, b и c:

- a=[1,2,3], b=[1,2,3], c=[1,2,3];
- a=[1], b=[2], c=[3];
- a=[1], b=[1,2], c=[1,2,3];
- a=[1,1], b=[2,2], c=[3,3].

11. Какой из представленных фрагментов кода не содержит ошибку:

- x = (y := 5);
- x = y := 5;
- x = (y) := 5;
- x = y := (5).

3.3.Условная инструкция if в Python

В ходе работы программ часто приходится выполнить (или пропустить) тот или иной фрагмент программного кода в зависимости от того, какие результаты были получены в предыдущих выражениях. Другими словами, в программе может присутствовать ветвление, которое реализуется условными операторами или операторами управления.

Как и в большинстве других языков программирования в Python присутствует условная инструкция «if», которая в общем случае имеет следующий формат записи:

```
if <Условие №1>:
  <Блок инструкций №1>
elif <Условие №2>:
  <Блок инструкций №2>
...
else:
  <Запасной блок инструкций>
```

Если первое условие истинно (результатом вычислений будет True), то выполняется первый блок инструкций. Если первое условие ложно (результатом вычислений будет False), то по очереди проверяются все условия необязательных блоков «elif» и, если найдется истинное условие, то выполняется соответствующий этому условию блок инструкций. Если все дополнительные условия окажутся ложными, выполнится запасной блок инструкций «else». Обязательным является только блок инструкций «if», остальные блоки могут быть опущены. При этом разрешается использовать любое количество блоков «elif», но только один блок «else».

Схематично ветвление программы показано на рис. 3.12.

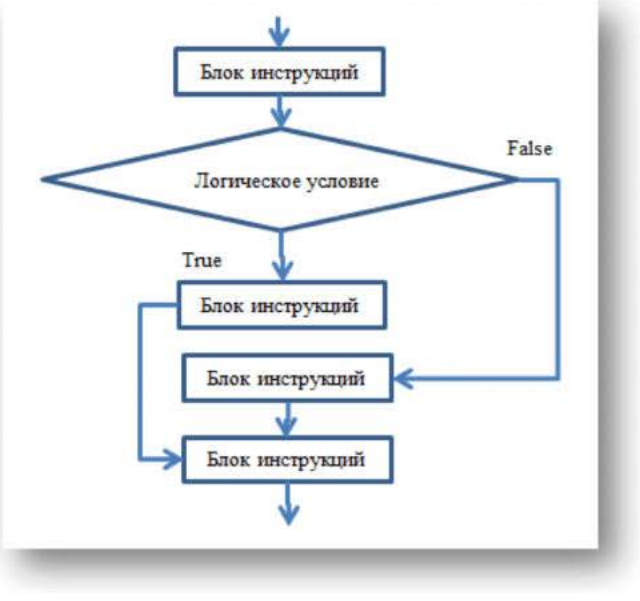


Рис. 3.12. Ветвление программы

3.3.1. Простейшая форма инструкции if

Для наглядности начнем с простейшей условной инструкции. Простейшая инструкция содержит только основную часть «if», в которой вычисляется значение единственного условного выражения и по результатам вычисления принимается решение:

- если конечный результат имеет значение True, то интерпретатор выполняет блок кода, который следует за «if»;
- если конечный результат имеет значение False, то интерпретатор пропускает данный фрагмент кода и начинает выполнять инструкцию, следующую за текущей условной инструкцией.

Пример программы, в которой показано использование простейшей условной инструкции «if», приведен в листинге 3.13.

Листинг 3.13. Использование простейшей условной инструкции «if»

```
# Листинг 3.13
num_1 = 10
num_2 = 20

if num_1 > num_2:
    s = num_1 + num_2

s = num_1 * num_2
print('Результат s=', s)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.13. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.13).

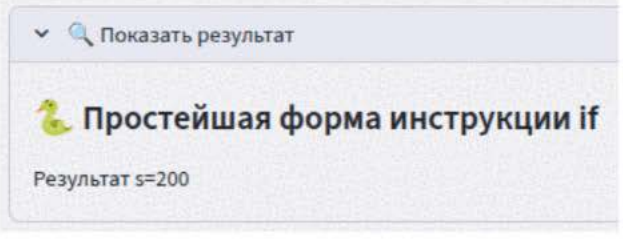


Рис. 3.13. Использование простейшей условной инструкции «if»

В этом примере двум переменным присвоены значения: num_1 = 10, num_2 = 20. Далее идет инструкция «if», которая проверяет условие num_1 > 10, num_2. Если это условие выполнится (результат True), то будет получена сумма значений этих переменных, а если не выполнится (результат False), то их произведение. Поскольку 10 меньше, чем 20, то

результатом этого сравнение будет значение – False, а значит, будет подсчитано произведение двух переменных, то есть число – 200.

Отметим, что в блоках кода инструкции if разрешается использовать, как другие вложенные инструкции «if», так и вложенные управляющие конструкции других видов (например, циклов или обработчиков ошибок). Это значительно повышает возможности организации условного выполнения различных фрагментов программы.

3.3.2.Управляющая конструкция if/else

Если в зависимости от результата проверки условного выражения нужно сделать выбор и выполнить только один из двух допустимых наборов инструкций, инструкция «if» может быть расширена при помощи ключевого слова «else», которое в случае, когда значение условия равно False, позволяет выполнить альтернативный набор инструкций.

Пример программы, в которой показано использование простейшей условной инструкции if … else, приведен в листинге 3.14.

Листинг 3.14. Использование условной инструкции if … else

```
# Листинг 3.14
num_1 = 10
num_2 = 20

if num_1 > num_2:
    s = num_1 + num_2
else:
    s = num_1 * num_2

print('Результат s=', s)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.14. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.14).

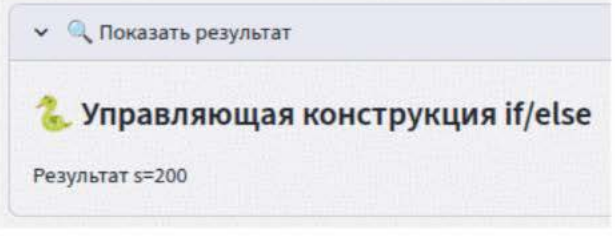


Рис. 3.14. Использование условной инструкции if … else

В этом примере было явно задано выполнение действий, как при значении логического выражения True, так и при значении логического выражения False. Такой код более понятен и лучше читается.

3.3.3. Ключевое слово elif

Когда требуется выполнить выбор не из двух вариантов, а из нескольких, то инструкция «if» может быть расширена при помощи требуемого количества ключевых слов «elif», которые, по сути, служат для объединения двух соседних инструкций «if».

Пример программы, в которой показано использование условной инструкции «if» и ключевых слов «elif», приведен в листинге 3.15.

Листинг 3.15. Использование условной инструкции «if» и ключевых слов «elif»

```
# Листинг 3.15
# Задаем цены на телевизоры
JVS = 20000
Sony = 30000
Samsung = 40000

TV = 1
# TV = 2
# TV = 3
# TV = 4

# Осуществляем проверку условных выражений.
if TV == 1:
    print('Введен код ->', TV, ', Цена JVS->', JVS)
elif TV == 2:
    print('Введен код ->', TV, ', Цена Sony->', Sony)
elif TV == 3:
    print('Введен код ->', TV, ', Цена Samsung->', Samsung)
else:
    print('Введен код ->', TV, ', Нет в продаже!')

print('-'*55)
```

В этом примере трем переменным заданы цены на различные марки телевизоров: JVS, Sony и Samsung. Далее предлагается задать код телевизора, для которого нужно узнать цену (1, 2, 3, 4). Часть строк закомментировано символом «#», снимая комментарий с одной из строк можно поменять значение этого кода.

Затем следует блок инструкций с использованием конструкции «if elif». В этом блоке будет проверено значение переменной TV и выдана цена на тот телевизор, код которого содержится в переменной TV. Если будет получен неизвестный этой инструкции код (число 4), то будет выдано сообщение – «Нет в продаже».

Итак, несколько раз запустим этот программный модуль, каждый раз снимая комментарий с одной из строк со значением кода (1, 2, 3, 4).

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.15. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.15).

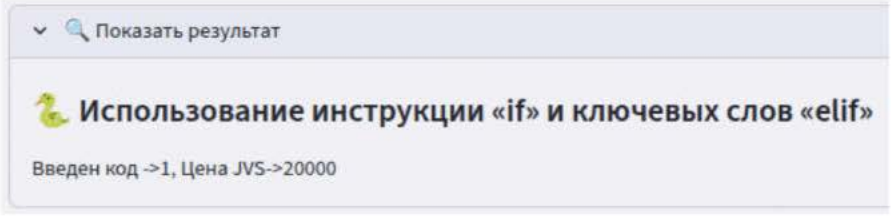


Рис. 3.15. Использование условной инструкции «if» и ключевых слов «elif»

Как видно из данного примера, при использовании ключевых слов «elif» выполнение альтернативного блока инструкций происходит лишь тогда, когда соответствующее блоку условие имеет значение True, а условные выражения всех предыдущих блоков в результате вычислений дают False. В случае отсутствия совпадений, выполняется набор инструкций дополнительного блока «else».

3.3.4. Трехместное выражение if/else

Как и в случае с инструкцией присваивания, использовать условные инструкции в выражениях запрещается. Но иногда ситуации бывают настолько простые, что использование многострочных условных инструкций становится нецелесообразным. Именно по этим причинам в Python была введена специальная конструкция, называемая трехместным выражением if/else или тернарным оператором if/else, которая может использоваться непосредственно в выражениях.

Пример программы, в которой показано использование трехместного выражения if/else, приведен в листинге 3.16.

Листинг 3.16. Использование трехместного выражения if/else

```
# Листинг 3.16
# Инициализируем переменные
x = 15
y = 10
# Обычная инструкция
if x > y:
    s = x + y
else:
    s = x - y
print('Обычная инструкция: s=', s)

# Трехместное выражение
```

```
s = x + y if x > y else x - y
print('Трехместное выражение: s=', s)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.16. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.16).

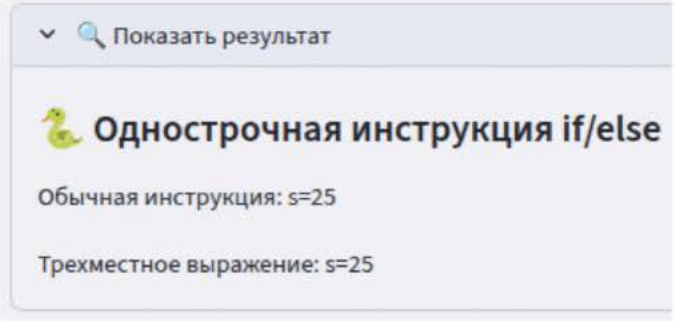


Рис. 3.16. Использование трехместного выражения if/else

Как видно из данного примера, трехместное выражение «if/else» дает тот же результат, что и традиционная инструкция «if». Однако в этом случае программный код гораздо короче и имеет всего одну строку, в то время как традиционный способ организации вычисления в разных ветках программы имеет 4 строки.

В этом примере операции в трехместном выражении «s = x + y if x > y else x – y» выполняются в следующей логической последовательности:

- проверяется условие x > y;
- если результат этой проверки будет иметь значение True, то будет найдена сумма переменных (s = x + y);
- если результат этой проверки будет иметь значение False, то будет найдена разность переменных (s = x - y).

При помощи такой короткой записи можно и косвенным путем обрабатывать исключения, то есть предотвращать прекращение работы программ из-за ошибок.

Пример программы, в которой показано использование трехместного выражения if/else для игнорирования потенциальных ошибок, приведен в листинге 3.17.

Листинг 3.17. Использование трехместного выражения if/else для игнорирования потенциальных ошибок

```
# Листинг 3.17
# Деление на 0 допустимо
x, y = 0, 3
res = x / y if y != 0 else 'Операция не выпонима (деление на 0)!'
```

```
print('res->', res)

# Деление на 0 не допустимо
x, y = 3, 0
res = x / y if y != 0 else 'Операция не выполнима (деление на 0)!'
print('res->', res)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.17. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.17).

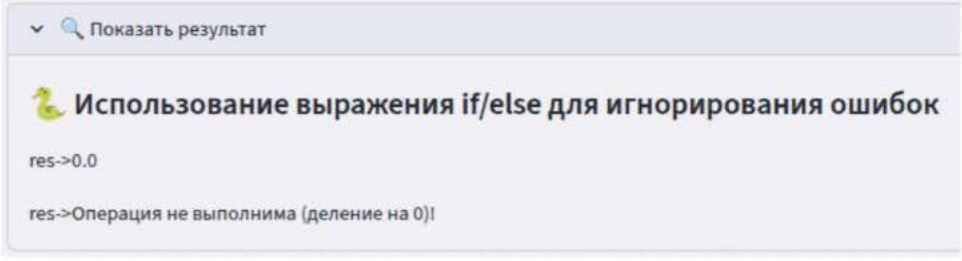


Рис. 3.17. Трехместного выражения if/else для игнорирования потенциальных ошибок

Трехместные выражения легко помещаются в одну строку и их можно использовать в простых выражениях. В более сложных случаях, когда выражения получаются громоздкими и читабельность кода падает, лучше использовать традиционную условную инструкцию «if/else».

3.3.5. Краткие итоги раздела 3.3

9 Условная инструкция «if/elif/else» дает возможность изменить ход программы за счет выполнения только определенных фрагментов (веток) программного кода. Обязательным является только ключевое слово «if», остальные ключевые слова конструкции могут быть опущены. При этом разрешается использовать любое количество ключевых слов «elif», но только одно ключевое слово «else».

10 В Python есть специальное трехместное выражение «if/else» (его еще называют тернарным оператором «if/else»). Эту конструкцию можно представить в следующем виде: «REZ = A if X else B». Вычисления здесь выполняются по сокращенной схеме: интерпретатор обрабатывает выражение «A», только когда условие «X» истинно, а выражение «Z» обрабатывается, только когда условие «X» ложно.

3.3.6.Вопросы для самоконтроля к разделу 3.3



13. 1. Какая из частей условной инструкции if/elif/else является обязательной:

- if;
- elif;
- else;
- if/else.

14. 2. Какая из частей условной инструкции if/elif/else позволяет разделить ход выполнение программы на 3 и более веток.
    - ❑ elif;
    - ❑ if;
    - ❑ else;
    - ❑ if/else.

3.4.Циклы for и while в Python

Выполнение программ, написанных на любом языке программирования, по умолчанию является последовательным (строка за строкой программного кода). Но достаточно часто требуется изменить ход выполнение программы, например, повторить несколько раз выполнение определенного фрагмента кода. Для этого в языках программирования предусмотрены различные типы циклов, которые способны повторять определенный код несколько раз. Чтобы понять принцип работы оператора цикла, рассмотрим следующие схемы (рис. 3.18).

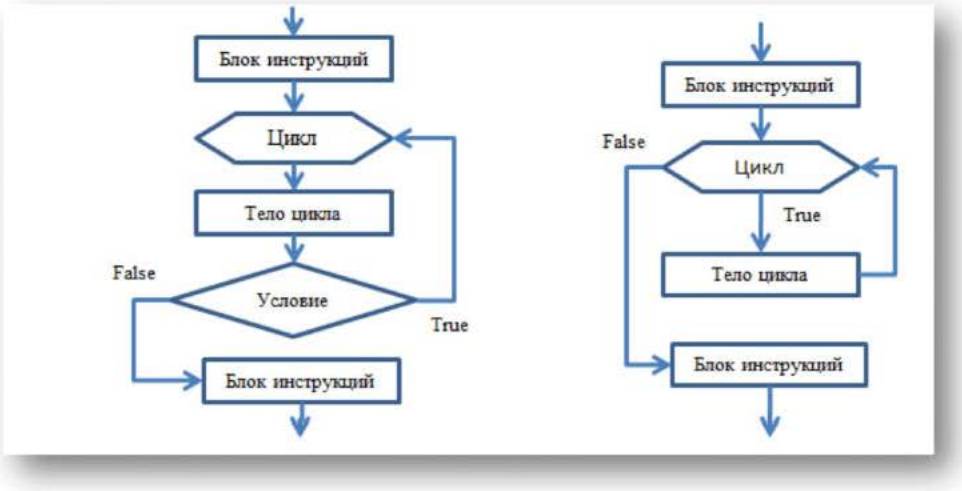


Рис. 3.18. Варианты организация циклов в программах

Согласно схемам, приведенным выше, возможны два варианта организации циклов в программах:

- схема «пока» - программный код в блоке цикла повторяется до тех пор, «пока» логическое выражение, присутствующее в инструкции организации выхода из цикла, имеет значение True;
- схема «для» - программный код в блоке цикла повторяется до тех пор, пока «для» переменной, присутствующей в инструкции организации цикла, не будут перебраны все значения в интервале 1 … n.

Циклы позволяют изменить ход программы таким образом, что вместо многократного написания и выполнения блоков похожего кода, можно повторить один блок кода заданное (или конечное) число раз.

3.4.1. Цикл for/in

Важной инструкцией языка Python является инструкция организация циклов «for/in». Это универсальный итератор (повторитель) последовательностей, поскольку эта инструкция позволяет выполнять обход (перебор) элементов в любых объектах, поддерживающих возможность совершения итераций. Сюда относятся как встроенные объекты (например, строки, списки или словари), так и объекты, создаваемые с помощью классов (о классах

будет рассказано чуть позже). Подробности про итераторы вы также узнаете немного позже (итераторам посвящен целый раздел).

В общем виде для цикла «for» используется следующий синтаксис:

```
for <Переменная> in <Итерируемый объект>:
  <Тело цикла>
else:
  <Дополнительный блок инструкций>
```

Пример программы, в которой показано использование инструкции организации цикла «for/in», приведен в листинге 3.18.

Листинг 3.18. Использование инструкции организации цикла «for/in»

```
# Листинг 3.18
# Формируем список
list_1 = [2, 5, 9, 4]

# Цикл перебора элементов списка
for elem in list_1:
    print('Элемент списка ->', elem)
print('-'*30)

# Формируем кортеж
kor = ['Иванов', 'Петров', 'Сидоров', 'Смирнов']

# Цикл перебора элементов кортежа
for elem in kor:
    print('Элемент кортежа ->', elem)
print('-'*30)

sum = 0  # Задаем начальную сумму

# Запускаем цикл получения суммы элементов списка
for elem in list_1:
    sum += elem

print('Сумма элементов списка ->', sum)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.18. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.19).

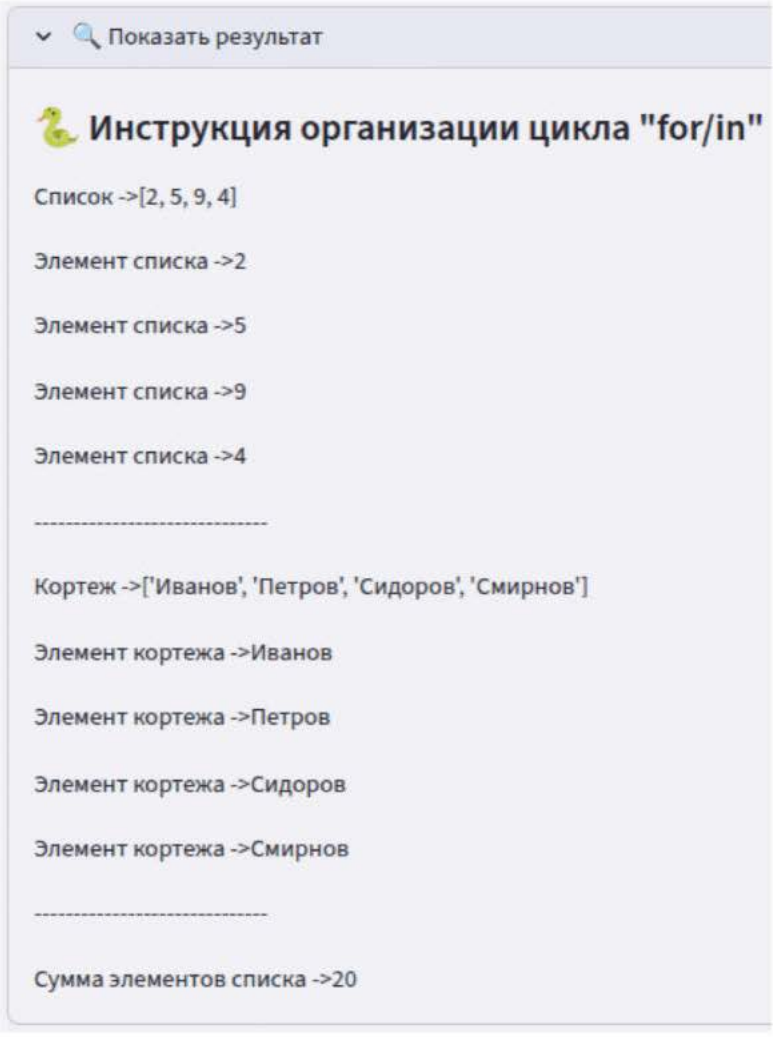


Рис. 3.19. Использование инструкции организации цикла «for/in»

В ходе выполнения цикла «for» интерпретатор один за другим присваивает переменной заголовка элементы итерируемого объекта и выполняет для них инструкции тела цикла.

Если цикл нужно прервать, используется ключевое слово break, которое передает управление инструкции, следующей сразу за циклом. Кроме того, в цикле «for» разрешается использовать необязательный блок «else», набор инструкций которого выполняется только после удачного завершения цикла (т.е. при отсутствии ошибки или прерывания за счет использования инструкции «break»).

Пример программы, в которой показано использование ключевых слов «break» и «else», приведен в листинге 3.19.

Листинг 3.19. Использование ключевых слов «break» и «else»

```
# Листинг 3.19
# Формируем список
list_1 = [1, 2, 3, 4, 5]

sum = 0          # Задаем начальную сумму
max_sum = 8      # Задаем предельную сумму
# max_sum = 40   # Задаем предельную сумму

# Запускаем цикл по элементам списка
for elem in list_1:
    if sum < max_sum:  # Если сумма чисел меньше max_sum
        sum += elem    # Увеличиваем сумму
    else:              # Если сумма чисел больше max_sum
        print("Сумма больше числа-", max_sum, "(прерываем цикл)!")
        break  # прерываем цикл

else:  # Если цикл не был прерван
    print("Сумма элементов списка sum =", sum)

# Выводим сообщение о завершении проверки
print("Конец работы программы!")
```

В этой программе был создан список из пяти элементов. Затем были созданы две переменные:

3. sum = 0 - начальная сумма элементов списка;
4. max_sum = 8 - предельная сумма элементов списка;
5. # max_sum = 40 - предельная сумма элементов списка (строка закомментирована).

Затем организуется цикл с помощью инструкции «for» (выборка из списка очередного элемента). В теле цикла делается проверка: не превысила ли сумма «sum» максимально допустимого значения «max_sum» (для этого используется условная инструкция «if/else»). Если в ходе очередной итерации цикла сумма не достигает указанного предела, то значение суммы «sum» увеличивается на значение переменной заголовка цикла «elem» (sum += elem). После этого цикл переходит к следующей итерации и присваивает переменной «elem» значение следующего элемента списка. Это продолжится до тех пор, пока не будет завершен обход (перебор) всех элементов списка, или не будет достигнуто предельное значение суммы.

Если сумма (sum) превысит заданный предел (max_sum), то будет выдано сообщение о прерывании цикла, после чего выполнится принудительный выход из цикла инструкцией «break».

Если цикл не был прерван, то программа выдаст результирующую сумму всех элементов списка.

И, наконец, вне зависимости от хода выполнения предыдущих блоков кода, в заключение будет напечатана строка с информацией о завершении работы программы.

Эту программу нужно запустить дважды: в том виде, как она есть, затем после снятия комментария со строки # max_sum = 40. Это обеспечит проход программы по всем веткам.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.19. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.20).

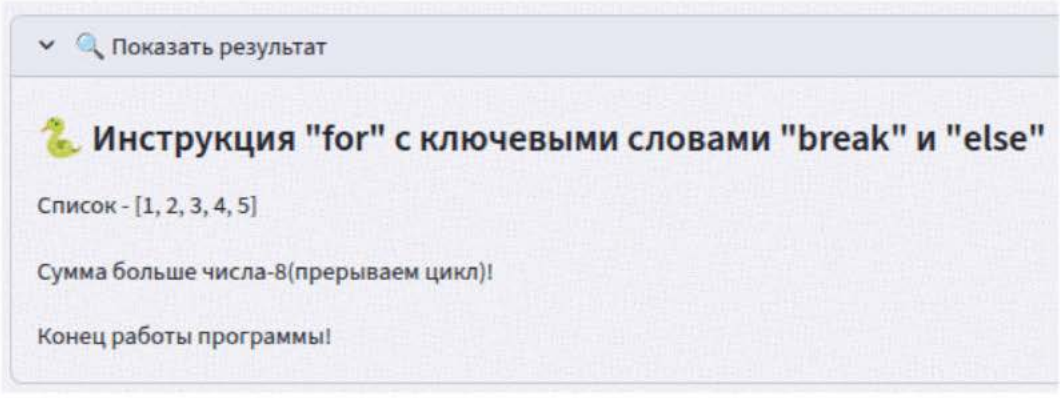


Рис. 3.20. Использование ключевых слов «break» и «else»

Из данного примера видно, что в первом случае цикл был действительно прерван инструкцией «break».

Обратите внимание, что в нашем примере были использованы два блока «else»: один принадлежит условной инструкции «if», а другой – инструкции цикла «for/in». Отличить их легко по уровню отступов.

Также стоит помнить, что значение переменной заголовка цикла «elem» при каждой новой итерации обновляется, поэтому после завершения цикла в ней будет храниться значение, присвоенное в ходе последней итерации.

С инструкции организации цикла «for» очень часто используется функция range(), которая позволяет генерировать ряд чисел в рамках заданного диапазона. Эта функция возвращает последовательность целых чисел на основе переданных ей аргументов. Эта функция может иметь три аргумента - range(start, stop, [step]):

- start — это первое значение в диапазоне чисел (если функция range() вызывается только с одним аргументом, то start = 0);
- stop — это верхняя граница диапазона (важно помнить, что само граничное значение не включается в последовательность).
- step – шаг изменения чисел в заданном диапазоне (не обязательный аргумент).

Пример программы, в которой показано организация цикла с инструкцией for и функцией range(), приведен в листинге 3.20.

Листинг 3.20. Организация цикла с инструкцией for и функцией range()

```
# Листинг 3.20
print('Все числа в диапазоне от 0 до 5 - range(5)')
for i in range(5):
    print('i=', i)
print('-'*5)

print('Все числа в диапазоне от 2 до 5 - range(2, 5)')
for i in range(2, 5):
    print('i=', i)
print('-'*5)

print('Все числа в диапазоне от 3 до 16 с шагом 3 - range(3, 16, 3)')
for i in range(3, 16, 3):
    print('i=', i)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.20. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Все числа в диапазоне от 0 до 5 - range(5)
i= 0
i= 1
i= 2
i= 3
i= 4
-----
Все числа в диапазоне от 2 до 5 - range(2, 5)
i= 2
i= 3
i= 4
-----
Все числа в диапазоне от 3 до 16 с шагом 3 - range(3, 16, 3)
i= 3
i= 6
i= 9
i= 12
i= 15
```

При использовании в циклах функции range() нужно помнить об одной особенности: число, в котором указана верхняя граница диапазона (stop), не войдет в список генерируемых чисел. Из приведенного выше примера видно, что когда был задан диапазон получения чисел от 0 до 5 - range(5), фактически был получен набор чисел от 0 до 4 (0, 1, 2, 3, 4).

Достаточно часто функции range() в циклах используется для генерации индексов, например при последовательном переборе элементов списков. Пример такого

программного кода, где в цикле значение каждого элемента списка умножается на 2, приведен в листинге 3.21.

Листинг 3.21. Организация цикла с инструкцией for и функцией range()

```
# Листинг 3.21
list_1 = [10, 20, 30, 40, 50]

for i in range(len(list_1)):
    s = list_1[i] * 2
    print('list_1[i]=', list_1[i], 's=', s)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.21. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.21).

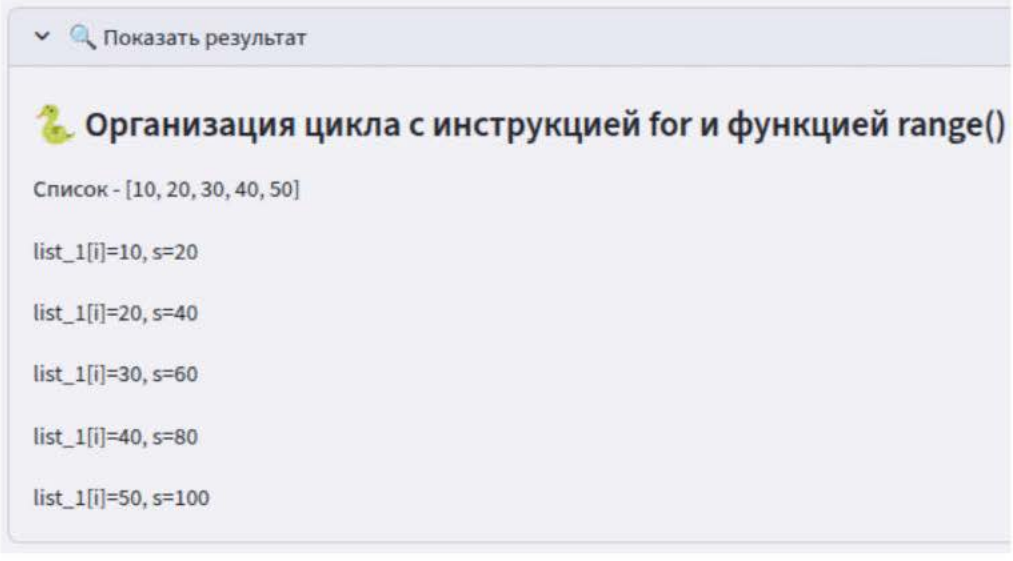


Рис. 3.21. Организация цикла с инструкцией for и функцией range()

Заголовок цикла может иметь и более сложную структуру, то есть сочетать такие операции, как выборка элементов из последовательностей, операции сравнения, арифметические действия. Ниже приведен программный код, в котором представлено несколько вариантов организации таких циклических структур (листинг 3.22).

Листинг 3.22. Варианты организации циклических структур

```
# Листинг 3.22
# Выводим символы строки с разделителем в виде пробела.
name = 'Привет'
for s in name: print(s)
print('-'*60)

name = 'ПРруитвоент'

# Используем функцию range(a, b, step).
for n in range(0, len(name), 2):
    print(name[n], end='_')  # Выводим буквы через одну, начиная с
первой

for n in range(1, len(name), 2):
    print(name[n], end='_')  # Выводим буквы через одну, начиная со
второй
print('', end='\n\n')


# Выводим ключи и значения словаря
d = {'Один': 1, 'Два': 2}
for key in d: print(key, '->', d[key])
print('-'*60)

# Перебираем список и находим суммы элементов кортежей
d = [(1, 2), (3, 4), (5, 6)]
for x, y in d: print(x + y, end=' ')
print('', end='\n\n')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.22. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.22).

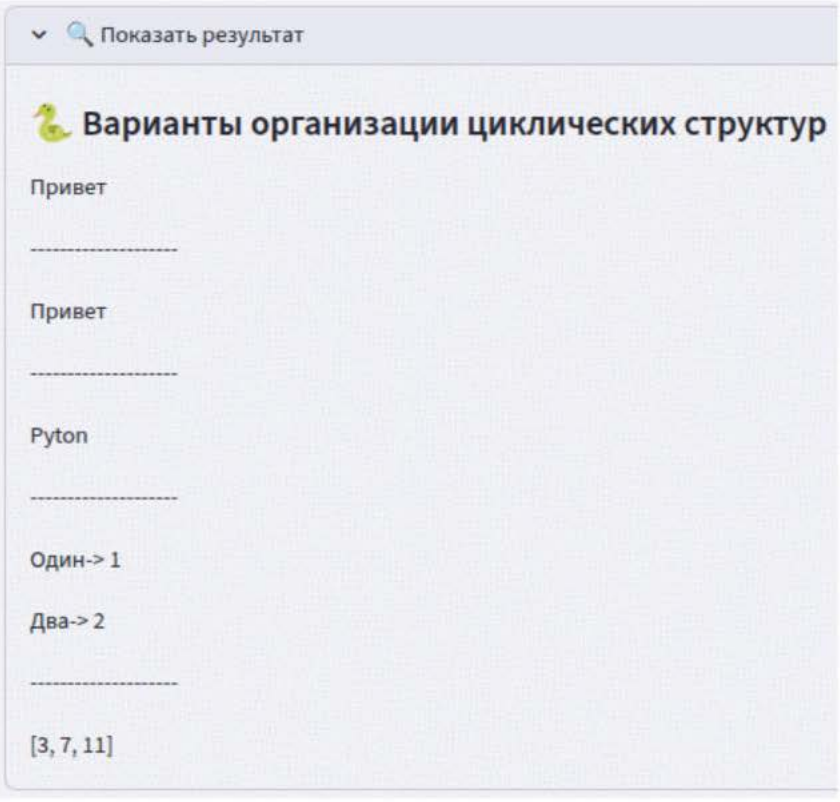


Рис 3.22. Варианты организации циклических структур

Как показали вышеприведенные примеры, инструкция цикла «for» в Python позволяет совершать обход (перебор) элементов в любых итерируемых объектах, особенно внутри упорядоченных последовательностей вроде строк или списков, в которых обход осуществляется поочередно в порядке возрастания индексов их элементов. При этом в заголовке цикла можно составлять как простые, так и достаточно сложные инструкции.

3.4.2. Цикл while

Инструкция цикла «while» представляет собой универсальный цикл общего назначения. Он служит для повторного выполнения кода, расположенного в его теле, до тех пор, пока условие цикла остается истинным, то есть имеет значение True. В общем случае синтаксис цикла «while» в Python можно представить в следующем виде:

```
while <Условное выражение>:
  <Тело цикла>
else:
  <Дополнительный блок инструкций>
```

Работает цикл «while» следующим образом. Каждый раз перед выполнением новой итерации, то есть перед очередным выполнением блока инструкций в теле цикла, происходит проверка его условия и, если результат будет иметь значение True, код в теле цикла выполняется. Далее интерпретатор возвращается в начало цикла и снова проверяет

его условие, повторяя все процедуры заново до тех пор, пока условие цикла не вернет значение False. При этом значении интерпретатор прерывает выполнение цикла и, пропустив блок инструкций в его теле, передает управление либо необязательному ключевому слову «else», выполняя дополнительный блок инструкций, либо инструкции, следующей за циклом. При этом следует иметь в виду, что набор инструкций блока «else» выполняется только тогда, когда выход из цикла производится не инструкцией «break».

Самый простой пример создания и использования цикла «while» приведен в листинге 3.23.

Листинг 3.23. Использование цикла «while»

```
# Листинг 3.23
s = 0

while s < 100:
    s = s + 1
print('Цикл окончен, s=', s)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.23. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.23).

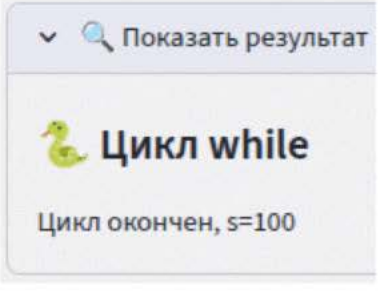


Рис. 3.23. Использование цикла «while»

Тело цикла может иметь и более сложную структуру, включая инструкции «else» и «break». Пример такой программы приведен в листинге 3.24.

Листинг 3.24. Использование цикла «while» с инструкциями «else» и «break»

```
# Листинг 3.24
# Формируем список
list_1 = [1, 2, 3, 4, 5]

sum = 0          # Задаем начальную сумму
max_sum = 8      # Задаем предельную сумму
# max_sum = 40   # Задаем предельную сумму
```

```
i = 0  # Устанавливаем счетчик итераций

# Запускаем цикл по элементам списка.
while i <= len(list_1) - 1:
    if sum < max_sum:   # Если сумма чисел меньше max_sum
        sum += list_1[i]  # Увеличиваем сумму
        i += 1          # Счетчик увеличиваем на 1
    else:               # Если сумма чисел больше max_sum
        print("Сумма больше числа-", max_sum, "(прерываем цикл)!")
        break  # прерываем цикл.

else:  # Если цикл не был прерван
    print("Сумма элементов списка sum =", sum)

# Выводим сообщение о завершении проверки.
print("Конец работы программы!")
```

В этой программе был создан список из пяти элементов. Затем были созданы три переменные:

- sum = 0 - начальная сумма элементов списка;
- max_sum = 8 - предельная сумма элементов списка;
- # max_sum = 40 - предельная сумма элементов списка (строка закомментирована);
- i = 0 - нулевое значение для счетчика итераций (циклов).

Затем организуется цикл с помощью инструкции «while». В этой инструкции проверяется условие: не превысил ли счетчик итераций (циклов) – «i» количества элементов списка. До тех пор, пока «i» будет меньше 5, логическое выражение будет иметь значение True, и цикл будет продолжаться.

В теле цикла делается проверка: не превысила ли сумма (sum) максимально допустимого значения (max_sum), для чего используется условная инструкция «if/else». Если в ходе очередной итерации цикла сумма не достигает указанного предела, то она будет увеличиваться оператором sum+=list[i], а счетчика суммы «i» будет увеличиваться на единицу i+=1. А если сумма (sum) превысит предельное значение (max_sum), то команда «break» прервет выполнение цикла и будет выдано соответствующее сообщение. Цикл будет продолжаться до тех пор, пока не будет завершен обход (перебор) всех элементов списка, или не будет достигнуто предельное значение суммы.

Еесли сумма (sum) превысит заданный предел (max_sum), то будет выдано сообщение о прерывании цикла, и будет выполнен принудительный выход из цикла инструкцией «break».

Если цикл не был прерван, то программа выдаст результирующую сумму всех элементов списка.

И, наконец, вне зависимости от хода выполнения предыдущих блоков кода, в заключение будет напечатана строка с информацией о завершении работы программы.

Эту программу нужно запустить дважды: в том виде, как она есть, затем после снятия комментария со строки # max_sum = 40. Это обеспечит проход программы по всем веткам.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.24. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.24).

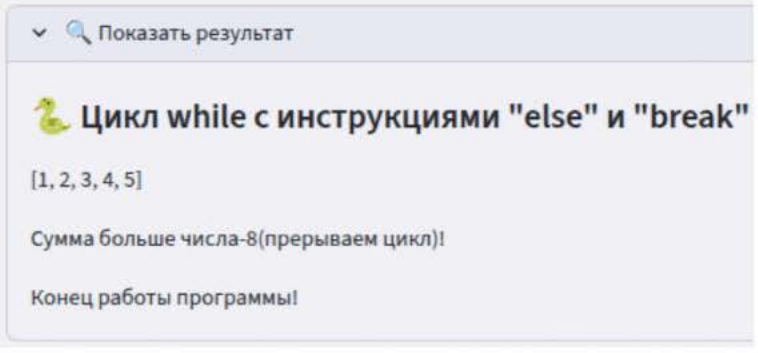


Рис. 3.24. Использование цикла «while» с инструкциями «else» и «break»

Обратите внимание, что в нашем примере были использованы два блока «else»: один принадлежит условной инструкции «if», а другой – инструкции цикла «while». Отличить их легко по уровню отступов.

Стоит заметить, что в языке Python отсутствует цикл «do/while», имеющийся в других языках программирования. Однако его можно имитировать, добавив в конец тела цикла условную инструкцию одновременно с инструкцией «break».

Пример такой программы приведен в листинге 3.25.

Листинг 3.25. Использование цикла «while» с инструкциями «break»

```
Листинг 3.25
i = 1  # Устанавливаем значение счетчика циклов

while True:  # Запускаем бесконечный цикл
    print(i, end=' ')  # Выводим номер итерации
    i += 1  # Увеличиваем счетчик циклов на 1
    if i > 5: break  # Выходим из цикла если i > 5
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.25. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.25).

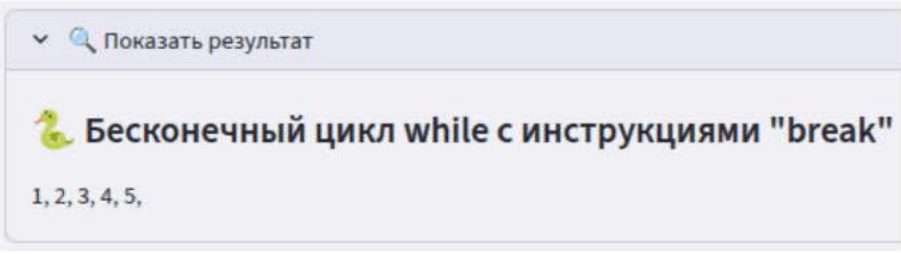


Рис. 3.25. Использование цикла «while» с инструкциями «break»

В данной конструкции оператор входа из цикла стоит последним. Отсюда следует, что строки программы, расположенные в теле цикла, будут выполнены как минимум один раз при любом значении счетчика. Это обстоятельство нужно учитывать при составлении своих программ.

Обратите внимание, что при использовании счетчиков в приведенных примерах начальные значения им присваивались вне цикла. Это нужно для того, чтобы при каждой новой итерации их значения не сбрасывались циклом в первоначальное состояние.

3.4.3. Инструкции break и continue

Кроме инструкции «break» (прервать) с циклами можно использовать и инструкцию «continue» (продолжить). Эти инструкции используются только внутри циклов и выполняют примерно те же функции, что и в других языках программирования:

- break - прерывает цикл и передает управление следующей за циклом инструкции;
- continue - прерывает только текущую итерацию цикла и производит переход в начало цикла, то есть в строку заголовка.

Принципиальное отличие инструкций «break» и «continue» представлено на рис. 3.26.

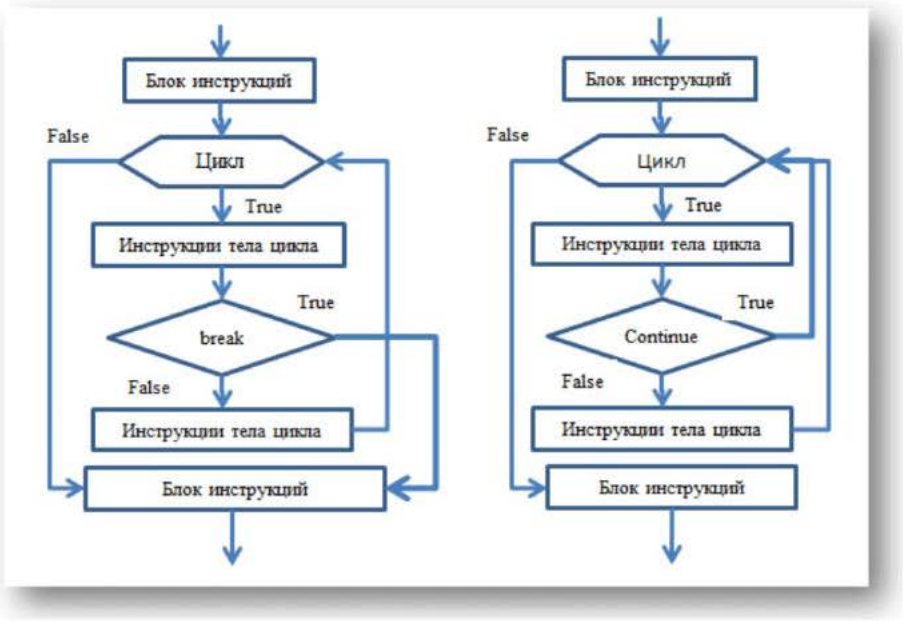


Рис. 3.26. Принципиальное отличие инструкций break и continue

Из данного рисунка видно, что инструкция «break» полностью завершает цикл и передает управление блоку инструкций, которые находятся сразу за пределами цикла, а инструкция «continue» прерывает выполнение только следующую за ней инструкцию и передает управление в блок организации цикла.

Обе инструкции (break и continue) могут находиться в любом месте внутри тела цикла, но, как правило, их используют в условных инструкциях «if», вложенных в цикл, чтобы выполнить необходимое действие в ответ на некоторое условие.

Ниже приведен программный код, в котором показано использование инструкций «break» и «continue» в тела цикла (листинг 3.26).

Листинг 3.26. Использование цикла «while» с инструкциями «break» и «continue»

```
# Листинг 3.26
k = 1  # Устанавливаем счетчик циклов

print('Бесконечный цикл инструкцией - while True:')
while True:  # Запускаем бесконечный цикл
    if k > 10:  # Задаем условие прерывания цикла
        break  # Прерываем цикл и идем дальше

    if k % 2 == 0:  # Отсеиваем четные числа
        k += 1      # Увеличиваем счетчик циклов на 1
        continue    # Возвращаемся в заголовок цикла

    print(k, end=' ')  # Выводим нечетные числа на экран
```

```
    k += 1              # Увеличиваем счетчик циклов на 1

# Печать результатов после завершения цикла
print('\nВывод нечетных чисел окончен!\n')

list_1 = list(range(1, 21))  # Формируем список чисел от 1 до 20

print('Цикл по элементам списка инструкцией - for k in list:')

for k in list_1:  # Запускаем цикл по элементам списка
    if k > 10:  # Задаем условие прерывания цикла
        break   # Прерываем цикл и идем дальше

    if k % 2 == 0:  # Отсеиваем четные числа
        k += 1      # Увеличиваем счетчик циклов на 1
        continue    # Возвращаемся в заголовок цикла

    print(k, end=' ')  # Выводим нечетные числа на экран
    k += 1             # Увеличиваем счетчик циклов на 1

# Печать результатов после завершения цикла
print('\nВывод нечетных чисел окончен!')
```

В программном коде этого примера производится выборка нечетных чисел из интервала чисел от 1 до 10. При этом используются две различные инструкции для организации циклов:

- инструкция «while True» для организации бесконечного цикла;
- инструкция «for k in list» для организации бесконечного цикла выборки элементов из списка.

В обоих случаях используется счетчик циклов «k». В условных инструкциях «if» по значению этого счетчика принимается решение, как по отсеиванию четных чисел, так и по выходу из циклов.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.26. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.27).

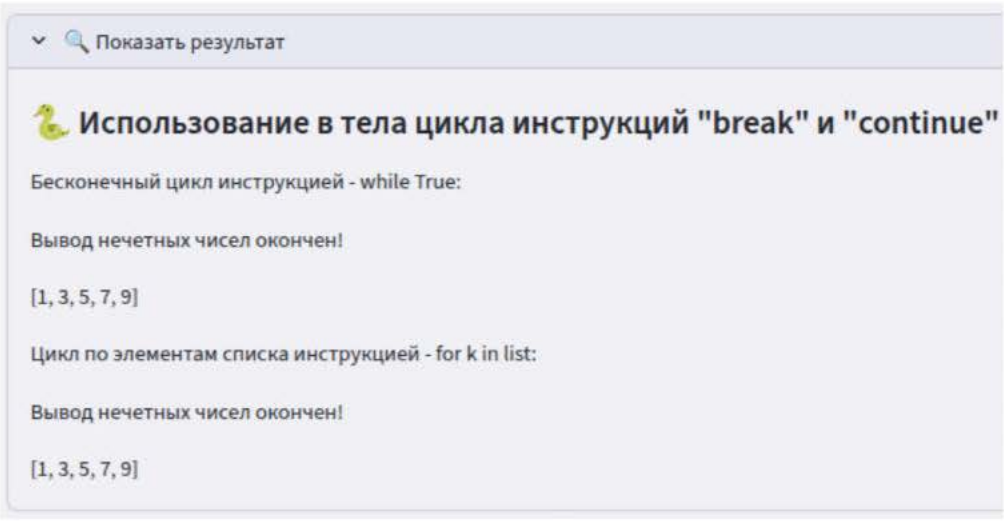


Рис. 3.27. Использование цикла «while» с инструкциями «break» и «continue»

Все виды циклов в Python могут иметь другие вложенные циклы и условные инструкции, позволяя нужным образом изменять ход выполнения программы. При этом каждая инструкция «break» или «continue» относится только к одному циклу, в теле которого она находится. Поэтому завершить все циклы из внутреннего (вложенного) цикла не получится, нужно использовать инструкцию «break» во всех циклах текущей иерархии.

Пример программы, в которой показана организация вложенных циклов, приведен в листинге 3.27.

Листинг 3.27. Организация вложенных циклов

```
# Листинг 3.27
print('Таблица умножения чисел от 2 до 5')
y = 1  # Задаем стартовое значение счетчика внутреннего цикла

for x in range(1, 11):  # Внешний цикл
    if x == 1:    # Если x==1, то умножение на 1 пропускаем
        continue  # Переходим в начало внешнего цикла for

    while True:       # Внутренний цикл
        if y == 1:    # Умножение на 1 пропускаем
            y += 1    # Увеличиваем счетчик внутреннего цикла на единицу
            continue  # Переходим в начало внутреннего цикла while

        print(x, 'x', y, '=', x * y)  # Выводим произведение чисел

        if y == 5:   # Если y==5
            print()  # Переходим на новую строку
            y = 2    # Задаем новое стартовое значение счетчика
внутреннего цикла
```

```
            break    # Прерываем внутренний цикл (внешний нет)

        y += 1  # Увеличиваем счетчик внутреннего цикла на единицуу

    if x == 5:  # Если x==5
        break   # Прерываем внешний цикл

# Выводим сообщение о завершении циклов и все программы
print('Программа завершила работу!')
```

В этой программе с использованием вложенных циклов, реализован алгоритм формирования таблицы умножения чисел в интервале от 1 до 5. Внешний цикл организован с использованием инструкции «for x in range(1, 11):», в котором переменная «x» меняет свое значение от 1 до 10. Внутренний (вложенный) цикл организован с использованием инструкции «while True» - бесконечный цикл, в котором используется счетчик циклов (переменная «y»). Внутренний цикл прерывается инструкцией «break» тогда, когда счетчик циклов достигает значения 5, а внешний цикл прерывается инструкцией «break» тогда, когда значение переменной «x» принимает значение 5. В обоих циклах используется инструкция «if», в которых проверяются условия:

- имеют ли переменные «x» или «y» значения, равные 1 (для таких значений элемент таблицы умножения не формируется, и управление передается в заголовки циклов инструкцией «continue»);
- имеет ли переменная «y» значение 5 (если да, то прерывается внутренний цикл инструкцией «break»);
- имеет ли переменная «x» значение 5 (если да, то прерывается внешний цикл инструкцией «break»).

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.27. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Таблица умножения чисел от 2 до 5
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10

3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15

4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
```

```
4 x 5 = 20

5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25

Программа завершила работу!
```

3.4.4. Краткие итоги раздела 3.4

6. Для обхода (перебора) элементов в любых объектах, поддерживающих возможность совершения итераций, в Python предназначен цикл «for/in». В ходе выполнения этого цикла интерпретатор один за другим присваивает переменной, которая находится в заголовке цикла, значения элементов итерируемого объекта, а в теле цикла выполняет инструкции с использованием значения этой переменной.
7. Для многократного повторного выполнения кода (не только обхода элементов) служит универсальный цикл общего назначения «while», который выполняет блок инструкций, расположенный в его теле, до тех пор, пока условие цикла остается истинным, то есть имеет значение True.
8. При необходимости в циклах «for» и «while» можно использовать необязательный блок «else», набор инструкций которого выполняется только после удачного завершения цикла (то есть при отсутствии ошибки или прерывания цикла инструкцией «break»).
9. При использовании в циклах счетчиков их следует инициализировать вне циклов, чтобы при каждой новой итерации значения счетчиков не сбрасывались циклом в первоначальное состояние.
10. Для прерывания циклов в Python используется инструкция «break», которая принудительно завершает цикл и передает управление следующей за циклом инструкции. Если нужно прервать только текущую итерацию, необходимо использовать инструкцию «continue», которая после прерывания итерации производит переход в начало цикла, то есть к заголовку. Обе инструкции могут появляться в любом месте внутри тела цикла, но, как правило, их используют во вложенных в цикл условных инструкциях «if», чтобы выполнить необходимое действие в ответ на некоторое условие.
11. В Python можно использовать многоуровневые вложенные циклы.

3.4.5. Вопросы для самоконтроля к разделу 3.4



- Для чего в программах используется такая инструкция организации цикла:
  - «for i in list:»
  - для последовательного перебора всех элементов списка «list»;
  - для присвоения переменной «i» индексов элементов списка «list»;
  - для организации цикла, где «i»-значение счетчика начала цикла (start), «list»-значение счетчика конца цикла (stop);
  - для поиска значения переменной «i» среди элементов списка «list».
- Для чего используется значение «A» при такой инструкции организации цикла:
  1. «for i in range(A, B, C):»

2. для задания начального значения счетчика цикла «i»;
3. для задания конечного значения счетчика цикла «i»;
4. для задания шага изменения счетчика цикла «i»;
5. для задания значения первого элемента, который будет обработан в цикле.

- Для чего используется значение «B» при такой инструкции организации цикла:
  - «for i in range(A, B, C):»
  - для задания конечного значения счетчика цикла «i»;
  - для задания начального значения счетчика цикла «i»;
  - для задания шага изменения счетчика цикла «i»;
  - для задания значения второго элемента, который будет обработан в цикле.
- Для чего используется значение «C» при такой инструкции организации цикла:
  - «for i in range(A, B, C):»
  - для задания шага изменения счетчика цикла «i»;
  - для задания конечного значения счетчика цикла «i»;
  - для задания начального значения счетчика цикла «i»;
  - для задания значения второго элемента, который будет обработан в цикле.
- Для чего в программах используется такая инструкция организации цикла:
  - «while i <= len(list) - 1:»
  - для последовательного перебора всех элементов списка «list»;
  - для присвоения переменной «i» индексов элементов списка «list»;
  - для организации цикла, где «-1» указывает на перебор с конца списка;
  - для поиска значения переменной «i» среди элементов списка «list».
- Для чего в программах используется такая инструкция организации цикла «while True»:
  - для организации бесконечных циклов;
  - для присвоения переменным, расположенным в теле цикла, значения «True»;
  - для выхода из цикла, если значение одного из параметров в теле цикла равно «True»;
  - для поиска переменных в цикле, значения которых равно «True».

3.5.Итераторы и генераторы в Python

Во многих языках программирования используют такие термины как итератор и итерация, генератор и генерация.

В обобщенном виде итерация - это некий процесс, который позволяет поочередно выбирать элементы из некоторой последовательности. Такой последовательностью может быть что угодно: набор чисел, набор символов, списки, сведения из базы данных или удаленного сервиса. В итераторах замечательно то, что программный код, использующий итератор, даже не должен знать, какой источник он использует, а сосредоточиться только на одном - «Что нужно сделать с каждым элементом?».

В обобщенном виде генерация – это формирование некой последовательности данных «на ходу», то есть в процессе выполнения программного кода.

Предположим, что требуется оценить время работы программы при обработке больших массивов данных, измеряемых десятками гигабайт. Для этого такие большие массивы информации нужно как то разместить на компьютере, вполне вероятно, что они просто не поместятся на диск. Но можно поступить проще, не хранить большие массивы информации, а создавать их в ходе выполнения программы. В Python для этих целей служат генераторы.

Генератор — это объект, который при создании сразу не вычисляет значения всех своих элементов. Он всегда хранит в памяти только последний вычисленный элемент. Вычисление следующего значения происходит лишь при выполнении метода next(), при этом предыдущее значение теряется. Если создать объект-список из 10 млн. элементов, то все эти элементы будут храниться в памяти. Если создать объект-генератор, который будет формировать тот же набор данных из 10 млн. элементов, то только один элементы будет храниться в памяти. То есть генераторы позволяют эффективно использовать память при обработке больших последовательностей данных.

Преимущество итераторов и генераторов заключается в том, что они используют отложенные вычисления, чего нет в традиционных функциях. Результаты вычисляются только в моменты запросов, при этом весь объект не обязательно должен находиться в памяти одновременно (что предотвращает проблемы, связанные с переполнением памяти).

3.5.1. Итераторы и итераций в Python

Итераторы – мощные и очень полезные составляющие Python.

Итерация (от англ. iteration) – это процесс обработки данных, при котором некоторые действия повторяются многократно, а одно повторение такого процесса (или один виток) мы также будем называть итерацией.

Основное назначение итераторов – это упрощение навигации по элементам объектов, которые, как правило, представляет собой некоторую коллекцию (список, словарь и т.п.), или итерируемый объект.

Итерируемый объект (от англ. iterable object) – это любой объект, элементы которого можно обрабатывать поочередно (один за другим) и за один проход.

Итератор (от англ. iterator) – это специальный объект, который имеет метод __next__(), позволяющий при каждом новом вызове получать следующий элемент итерируемого объекта.

Итерируемый объект и итератор – тесно связанные понятия. Итерируемый объект – это объект, который можно передать в функцию iter() и получить к нему итератор. Обычно итерируемые объекты – это какие-то коллекции, или последовательности, у которых есть элементы, и их можно перебрать как бусины. Однако объект не может перебирать сам себя, но он может предоставить такую возможность объекту другого вида «объекту – итератору. Итератор – это объект, который непосредственно осуществляет перебор итерируемого объекта. Схематично взаимодействие между итерируемым объектом и итератором представлено на рис. 3.28.

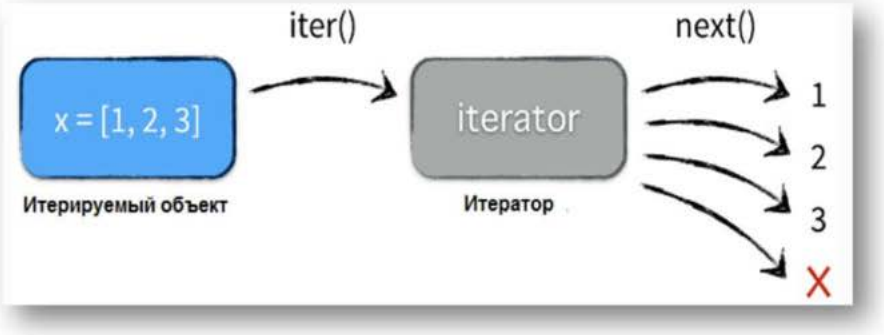


Рис. 3.28. Взаимодействие между итерируемым объектом и итератором

Язык Python, как и многие современные языки, поддерживает итераторы. Итератор представляет собой объект, который поочередно получает элементы итерируемого объекта. Например, с использованием следующей инструкции организован цикл: «for elem in list:», где list[1,2,3,4,5] это список из пяти элементов. В теле такого цикла будет выполнено пять итераций (повторений), а объект – итератор «elem» поочередно получит следующие значения: elem=1, elem=2, elem=3, elem=4, elem=5. Как это выглядит в программе на Python представлено в листинге 3.28.

Листинг 3.28. Итераторы в Python

```
# Листинг 3.28
# Формируем список
list_1 = [1, 2, 3, 4, 5]

# Запускаем цикл по элементам списка
for elem in list_1:
    print("Значение объекта-итератора ->", elem)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.28. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.29).

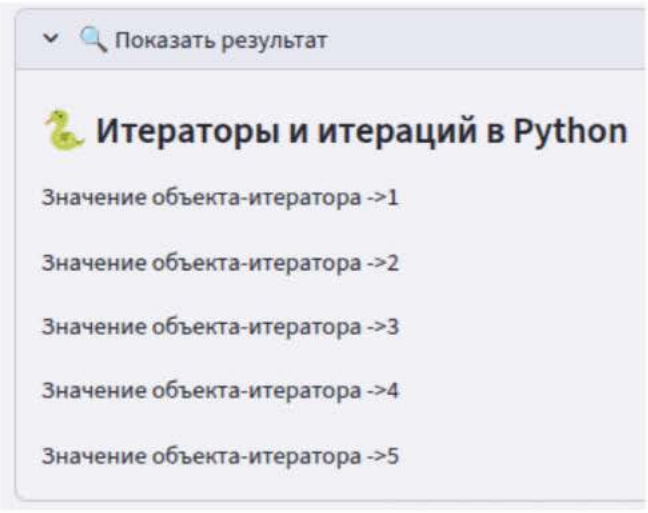


Рис. 3.29. Итераторы в Python

3.5.2. Протокол итераций в Python

В Python при обработке последовательностей данных используется так называемый протокол итераций, который позволяет с легкостью обрабатывать требуемое количество элементов любого итерируемого объекта. Именно этот протокол лежит в основе работы цикла «for», различных генераторов и некоторых встроенных функций, например map() или filter(). Перебрать все элементы объекта с помощью итераций можно только для итерируемых объектов, а в Python большинство объектов из последовательностей данных являются итерируемыми.

В Python все данные, в том числе и итерируемые, являются объектами, для работы с которыми имеется множество встроенных функций и методов. Среди этих методов стоит выделить так называемые «магические методы». Признаком «магических методов» является окружение их имен с обеих сторон двойным подчеркиванием (например, __next__). Ниже приведены некоторые из таких функций и методов:

- iter() - функция, которая создает объект-итератор из последовательности;
- next() - функция, которая «вычисляет» и возвращает следующий элемент объекта итератора;
- __next__() - метод, который «вычисляет» и возвращает следующий элемент итератора;
- hasattr(object, name) - функция, которая проверяет существование атрибута у объекта (здесь: object – объект, в котором нужно проверить существование атрибута, name - имя проверяемого атрибута);
- __iter__() - метод, который позволяет определить, поддерживает ли объект итерации;
- __getitem__() - метод, который позволяет определить, поддерживает ли объект выборку элементов по индексу.

Пример использования функций iter(), next() и метода __next__() приведен в листинге 3.29.

Листинг 3.29. Итераторы в Python

```
# Листинг 3.29
print('----- Создаем список "x" из строк -----')
x = ["apple", "banana", "cherry"]
print('Список "x":  ', x)
print()

print('----- Создаем объект-итератор "x" из списка строк -----')
x = iter(["apple", "banana", "cherry"])
print('Объект-итератор "x":  ', x)
print()

print('----- Вывод элементов итератора функцией next(x) ----- ')
print(next(x))
print(next(x))
print(next(x))
print('-'*30)
print()

print('----- Создаем объект-итератор - "x" из списка чисел -----')
x = iter([1, 2, 3])
print('Объект-итератор "x":  ', x)
print()
print('----- Вывод элементов итератора методом __next__() -----')
print(x.__next__())
print(x.__next__())
print(x.__next__())
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.29. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
----- Создаем список "x" из строк -----
Список "x":  ['apple', 'banana', 'cherry']

----- Создаем объект-итератор "x" из списка строк -----
Объект-итератор "x":  <list_iterator object at 0x0000000001F3B8C8>

----- Вывод элементов итератора функцией next(x) -----
apple
banana
cherry
```

```
------------------------------

----- Создаем объект-итератор - "x" из списка чисел -----
Объект-итератор "x":   <list_iterator object at 0x0000000001F3B948>

----- Вывод элементов итератора методом __next__() -----
1
2
3
```

Теперь рассмотрим пример, в котором показано использование функции hasattr() и методов «__iter__()» и «__getitem__()», который представлен в листинге 3.30.

Листинг 3.30. Итераторы в Python

```
# Листинг 3.30
# Проверяем строки на наличие методов __iter__ и __getitem__
s = 'Программа на Python'
print('Исходная строка ->  ', s)
print('Поддерживают ли строки итерации ->', hasattr(s, '__iter__'))
print('Поддерживают ли строки порядок следования элементов ->:',
hasattr(s, '__getitem__'))
print('----- Цикл обхода символов строки -----')
for simb in s: print(simb, end=' ')
print(end='\n\n')

# Проверяем списки на наличие методов __iter__ и __getitem__
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print('Исходный список ->  ', list)
print('Поддерживают ли списки итерации ->', hasattr(list, '__iter__'))
print('Поддерживают ли списки порядок следования элементов ->',
hasattr(list, '__getitem__'))
print('----- Цикл обхода элементов списка -----')
for elem in list_1: print(elem, end=' ')
print(end='\n\n')

# Проверяем словари на наличие методов __iter__ и __getitem__
d = {1: 'Иванов', 2: 'Петров', 3: 'Сидоров'}
print('Исходный словарь ->  ', d)
print('Поддерживают ли словари итерации ->', hasattr(d, '__iter__'))
print('Поддерживают ли словари порядок следования элементов ->',
hasattr(d, '__getitem__'))
# Запускаем цикл для обхода ключей словаря.
print('----- Цикл обхода элементов словаря -----')
for key in d: print(key, end=' ')
print(end='\n\n')

# Проверяем множество на наличие методов __iter__ и __getitem__
```

```
__
st = {'Один', 'Два', 'Три', 'Четыре', 'Пять'}
print('Исходное множество ->  ', st)
print('Поддерживают ли множества итерации ->', hasattr(st, '__iter__'))
print('Поддерживают ли множества порядок следования элементов ->',
hasattr(st, '__getitem__'))
print('----- Цикл обхода элементов множества -----')
for elem in st: print(elem, end=' ')
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.30. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Исходная строка ->  Программа на Python
Поддерживают ли строки итерации -> True
Поддерживают ли строки порядок следования элементов ->: True
----- Цикл обхода символов строки -----
П р о г р а м м а  н а  P y t h o n

Исходный список ->  [1, 2, 3, 4, 5, 6, 7, 8, 9]
Поддерживают ли списки итерации -> True
Поддерживают ли списки порядок следования элементов -> True
----- Цикл обхода элементов списка -----
1 2 3 4 5 6 7 8 9

Исходный словарь ->  {1: 'Иванов', 2: 'Петров', 3: 'Сидоров'}
Поддерживают ли словари итерации -> True
Поддерживают ли словари порядок следования элементов -> True
----- Цикл обхода элементов словаря -----
1 2 3

Исходное множество ->  {'Один', 'Четыре', 'Три', 'Пять', 'Два'}
Поддерживают ли множества итерации -> True
Поддерживают ли множества порядок следования элементов -> False
----- Цикл обхода элементов множества -----
Один Четыре Три Пять Два
```

Как видно из данного примера, все рассмотренные объекты (строки, списки словари, множества) представляют собой итераторы. Кроме того, элементы строк, списков и словарей всегда упорядочены, то есть элементы в этих объектах всегда следуют друг за другом в строгом порядке. А вот множества таким свойствам не обладают. При инициализации объекта они следовали в следующем порядке: Один, Два, Три, Четыре, Пять, а на экран вышли в последовательности: Один, Четыре, Три, Пять, Два. Мало того, при многократном запуске программы они каждый раз будут выдаваться в разной последовательности.

Из всего вышесказанного следует, что итерируемый объект предоставляет нам некоторый набор элементов, а также инструмент для поэлементной обработки этого набора. У строк таким набором являются символы, у списков и множеств – их элементы, у словарей – ключи и значения, у объектов – атрибуты, у файлов – строки и т.д. При этом, каким бы доступным итерационным инструментом мы не воспользовались, интерпретатор совершает итерации не хаотично, а в определенном порядке, согласно принятому протоколу итераций. Этот порядок заключается в следующем.

- У итерируемого объекта вызывается метод __iter__, который возвращает итератор объекта. Итератор хранит в себе информацию обо всех элементах итерируемого объекта и способен выдавать их по одному за раз.
- Затем для получения каждого элемента итерируемого объекта у полученного итератора начинает вызываться метод __next__().
- Каждый последующий вызов метода __next__() возвращает следующий элемент итерируемого объекта. Это продолжается до тех пор, пока очередной вызов метода не возбудит исключение StopIteration.
- Возбуждение исключения StopIteration приводит к остановке итераций и выходу из процесса.

В Python имеется удобный аналог метода __next__ в виде встроенной функции next(iterator [, default]), которая имеет следующие аргументы:

- iterator – объект итератор;
- default – значение, которое будет выдано в случае исключения (необязательный аргумент).

Эта функция при отсутствии аргумента «default» вызывает метод __next__ итератора, который возвращает значение очередного элемента. Если элементы закончились, то возбуждение исключение «StopIteration» и происходит остановка итераций. Это исключение нужно обработать, иначе программа выдаст ошибку и остановит свою работу.

Если же аргумент «default» присутствует, то по окончании итераций, функция перехватывает исключение StopIteration и вместо него возвращает значение аргумента «default». В этом случае не нужно предпринимать дополнительных мер по обработке исключений.

Пример использования функции next() представлен в листинге 3.31.

Листинг 3.31. Итераторы, использование функции next()

```
# Листинг 3.31
list_1 = [1, 2, 3]  # Создаем список
print('Исходный список')
for e in list_1: print(e, end=' ')  # Выводим элементы списка циклом
print()
print('-'*60)

# Получаем объект-итератор из списка посредством функции iter()
obj_iter = iter(list_1)  # создаем объект-итератор
print('Исходный список в виде объекта obj_iter')
print('obj_iter: ', obj_iter)
```

```
print('Элементы obj_iter с помощью функции next(obj_iter)')
print('Элемент списка ->', next(obj_iter))
print('Элемент списка ->', next(obj_iter))
print('Элемент списка ->', next(obj_iter))
print()

obj_iter = iter(list_1)  # создаем объект-итератор заново
print('Элементы obj_iter с помощью функции next(obj_iter, default)')
print('Элемент списка ->', next(obj_iter, '<- Список закончился ->'))
print('Элемент списка ->', next(obj_iter, '<- Список закончился ->'))
print('Элемент списка ->', next(obj_iter, '<- Список закончился ->'))
print('Элемент списка ->', next(obj_iter, '<- Список закончился ->'))
print('Элемент списка ->', next(obj_iter, '<- Список закончился ->'))
print('Элемент списка ->', next(obj_iter, '<- Список закончился ->'))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.31. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Исходный список
1 2 3
------------------------------------------------------------
Исходный список в виде объекта obj_iter
obj_iter:  <list_iterator object at 0x000000000224B888>
Элементы obj_iter с помощью функции next(obj_iter)
Элемент списка -> 1
Элемент списка -> 2
Элемент списка -> 3

Элементы obj_iter с помощью функции next(obj_iter, default)
Элемент списка -> 1
Элемент списка -> 2
Элемент списка -> 3
Элемент списка -> <- Список закончился ->
Элемент списка -> <- Список закончился ->
Элемент списка -> <- Список закончился ->
```

Как видно из данного примера, в случае «ручной» организации процесса перебора элементов списка, при попытке выдать несуществующий элемент списка ошибка программы автоматически перехватывается.

При необходимости никто не запрещает совершать итерации в «ручном» режиме, используя для этого протокол итераций, однако это не совсем удобно. Когда мы запускаем процесс итерации в обычном режиме (например, циклом «for»), то работа протокола итераций совершается в автоматическом (фоновом) режиме, предоставляя на выходе готовые

результаты. Это делается специально, чтобы использование итерационных инструментов для пользователей было максимально простым и удобным.

Насколько проще использовать традиционный поход работы с итерациями по сравнению с ручным режимом (на основе соблюдения правил протокола итерации) представлено в программе листинга 3.32.

Листинг 3.32. Итераторы, традиционный поход работы с итерациями

```
# Листинг 3.32
li = [1, 2, 3]  # Создаем список
for e in li: print(e, end=' ') # Получаем элементы списка в авторежиме
print()

# Используем протокол итераций и
# получаем элементы списка в ручном режиме
li_iter = iter(li)
print(next(li_iter), end=' ')
print(next(li_iter), end=' ')
print(next(li_iter), end=' ')
print(next(li_iter, 'Конец списка'))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.32. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.30).

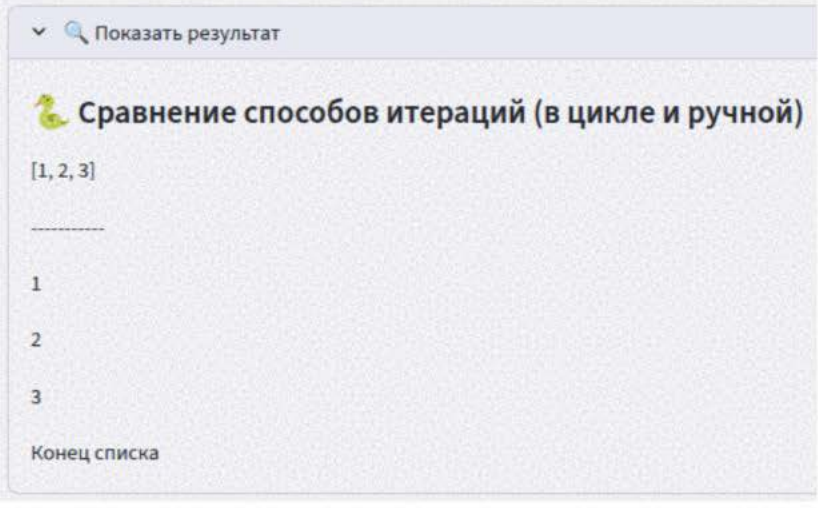


Рис. 3.30. Итераторы, традиционный поход работы с итерациями

Из данного примера видно, что использовать протокол итераций вручную без острой необходимости не рационально. Именно поэтому вся черновая и рутинная работа выполняется интерпретатором, позволяя программисту экономить время в процессе написания кода. Кроме того, использование в итерациях вместо самих итерируемых объектов их итераторов позволяет экономить и память, чем больше элементов содержит итерируемый объект, тем большей будет такая экономия. Насколько сильно отличается объем памяти для хранения итерируемого объекта и итератора наглядно демонстрирует пример программного кода, представленного в листинге 3.33.

Листинг 3.33. Итераторы, оценка объема памяти

```
# Листинг 3.33
import sys  # Импортируем модуль sys

list_1 = list(range(0, 100))  # Создадим список 100 элементов
iter_list = iter(list_1)         # Получаем итератор списка
# Выводим размеры списка и его итератора
print('Размер списка 100 элементов:', sys.getsizeof(list_1))
print('Размер итератора этого списка:', sys.getsizeof(iter_list))

print()

list_1 = list(range(0, 100000))  # Создадим список 1 000 000 элементов
iter_list = iter(list_1)         # Получаем итератор списка
# Выводим размеры списка и его итератора
print('Размер списка 1 000 000 элементов:', sys.getsizeof(list_1))
print('Размер итератора этого списка:', sys.getsizeof(iter_list))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.33. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.31).

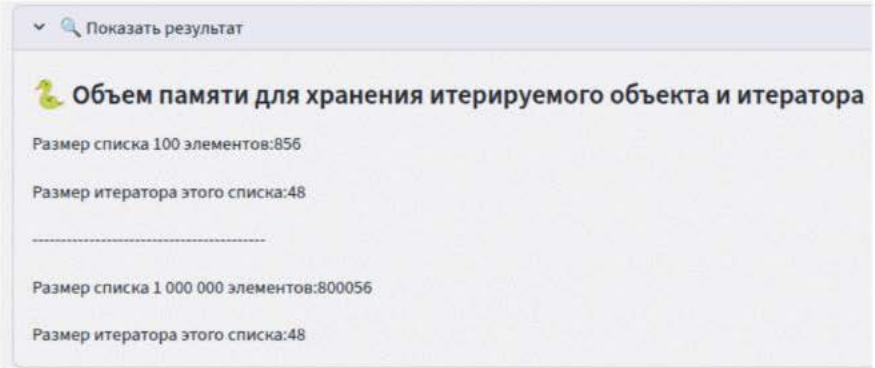


Рис. 3.31. Итераторы, оценка объема памяти

Как видно из данного примера, даже небольшой список по размеру превышает размер итератора. Но, что важно, при увеличении количества элементов любого итерируемого объекта его размер будет стремительно расти, а вот размер итератора останется прежним. Поэтому, например, при построчном чтении файла при помощи цикла «for» лучше использовать его итератор, а не имеющийся у файлов метод readlines(), который загружает весь файл целиком.

Также стоит отметить, что файловые объекты и некоторые другие типы данных имеют собственные итераторы, а значит и собственный метод __next__. Поэтому для работы с файлами не требуется получать другой объект, то есть итератор. А вот списки и многие другие встроенные объекты не имеют собственных итераторов, так как они поддерживают возможность участия сразу в нескольких итерациях. Поэтому для начала итерации по таким объектам нужно предварительно вызывать функцию iter().

Пример, демонстрирующий возможности итерации для файловых объектов и списков, представлен в листинге 3.34.

Листинг 3.34. Итераторы, возможности итерации для файловых объектов и списков

```
# Листинг 3.34
import os
# Удаляем текстовый файл (если он ранее был создан)
os.remove('text_file.txt')

# Создаем пустой текстовый файл
my_file = open("text_file.txt", "a+", encoding='utf-8')

# Записываем в текстовый файл две строки
my_file.write("Это первая строка текстового файла" + '\n')
my_file.write("А это вторая строка")
my_file.close()  # закрываем текстовый файл

#  Создаем объект и помещаем в него текстовый файл
obj_file = open('text_file.txt', encoding='utf-8')

print('К элементам файлового объекта можно обращаться напрямую (итератор
встроен)')
print(obj_file.__next__())  # Выведет первую строку
print(obj_file.__next__())  # Выведет вторую строку
print('-'*30)

list_1 = ['Иванов', 'Петров']  # создаем список

# У списков нет встроенных итераторов
# поэтому следующая строка сгенерирует ошибку
# print(list_1.__next__())

obj_iter = list_1.__iter__()  # создаем итератор для list_1

print('У списков нет встроенного итератора')
```

```
# Листинг 3.34

print('Но для них можно создать объект - итератор')
print('Через который можно получить доступ к элементам списка')
print('Первый элемент списка ->', obj_iter.__next__())
print('Второй элемент списка ->', obj_iter.__next__())
```

Здесь мы создали пустой текстовый файл с именем text_file.txt (перед этим на всякий случай удаляем файл с таким именем, если он ранее был создан). Затем в этот файл поместили две строки. После этого был создан объект с именем obj_file, в который загрузили только что созданный текстовый файл. Далее с помощью метода __next__() напрямую получили доступ к строкам текстового файла. Это подтверждает, что файловые объекты имеют встроенный итератор.

Затем был создан список list_1 из двух элементов. Попытка вывести элементы списка напрямую, через метод __next__(), приведет к ошибке, так как списки не имеют встроенного итератора. В связи с этим в программном модуле строка «print(list_1.__next__())» закомментирована. Но в случае необходимости для списков можно создать внешний объект итератор с помощью встроенного метода __iter__(). В вышеприведенном примере для списка был создан такой объект-итератор с именем obj_iter. Созданный объект-итератор obj_iter имеет встроенный метод __next__(), через который можно перебирать элементы списка. В следующих строках программы через итератор obj_iter элементы списка были выведены на печать.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.34. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.32).

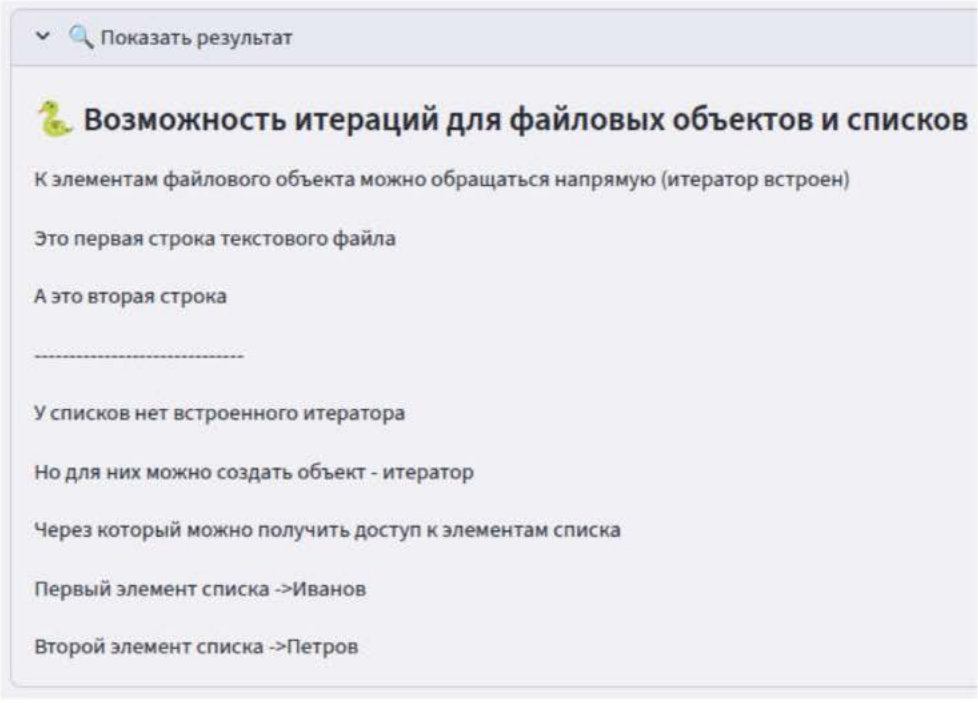


Рис. 3.32. Итераторы, возможности итерации для файловых объектов и списков

При прямом переборе элементов последовательностей с использованием итераторов нужно помнить, что после того, как итерации по итераторам были завершены, повторное их использование невозможно. Это связано с тем, что итераторы в языке Python представляют собой одноразовые инструменты. При каждом новом вызове метода __next__() из итератора извлекается очередной элемент, в результате чего итератор постепенно опустошается и в конечном итоге метод __next__ возбуждает исключение StopIteration (чтобы избежать ошибок в программе это исключение нужно обрабатывать). Пустые итераторы становятся непригодными для дальнейшего использования. Однако повторный запуск итераций по тому же итерируемому объекту может быть возобновлен, для чего нужно создать новый итератора (можно под тем же именем).

3.5.3. Выражения-генераторы и генерация последовательностей

Генератор — это альтернативный и более простой способ возвращать итераторы, и чтобы при этом оперативная память не перегружалась большими объемами информации. Генератор может эффективно производить бесконечный ряд значений. По своей сути генератор — это функция, которая выдает последовательность результатов, а не одного значение. Работу такой функции можно приостанавливать и возобновлять в ходе работы программы, при этом генераторы возвращают объекты, которые можно итерировать.

В предыдущих разделах уже использовались различные генераторы для создания списков, словарей и множеств. Но дело в том, что все эти генераторы, по сути, могут быть получены из более общих первичных конструкций, называемых выражениями-генераторами.

Выражение-генератор (от англ. generator expression) – это конструкция на базе цикла «for», которая применяет заданное выражение к каждому элементу переданного итерируемого объекта в соответствии с заданным условием и затем возвращает объект-генератор со встроенным итератором, содержащий информацию обо всех сформированных выражением элементах.

В общем случае синтаксис генератора-выражения в Python можно представить в следующем виде:

```
gen_obj = (expression for variable in iterable_obj [if condition])
```

В этом выражении термины имеют следующий смысл:

- gen_obj – создаваемый объект-генератор со встроенным итератором (получает значения, формируемые генератором-выражением);
- expression – выражение, формирующее новые элементы для итератора;
- variable – переменная-счетчик цикла «for», которой поочередно передаются элементы итерируемого объекта iterable_obj;
- iterable_obj – итерируемый объект, который служит источником элементов для создаваемого генератором итератора;
- condition – необязательное условное выражение, в соответствии с которым происходит отбор элементов итерируемого объекта iterable_obj.

В общей форме синтаксис генератора-выражения может показаться в некоторой мере затруднительным для восприятия, однако на практике все выглядит значительно проще.

Пример использования генератора-выражения приведен в листинге 3.35.

Листинг 3.35. Пример использования генератора-выражения

```
# Листинг 3.35
# Выражение-генератор без условия
gen_1 = (x ** 2 for x in range(2, 6))
print('Это объект-генератор (gen_1)')
print(gen_1)

print('Это значения, выдаваемые генератором gen_1')
print(gen_1.__next__(), end=' ')
print(gen_1.__next__(), end=' ')
print(gen_1.__next__(), end=' ')
print(gen_1.__next__(), end=' ')
print(next(gen_1, 'StopIteration'))
print('-' * 30)

# Выражение-генератор с условием if
gen_2 = (x * 10 for x in range(2, 10) if x > 4)
print('Это объект-генератор (gen_2)')
print(gen_2)
print('Это значения, выдаваемые генератором gen_2')
for elem in gen_2:
    print(elem, end=' ')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.35. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.33).

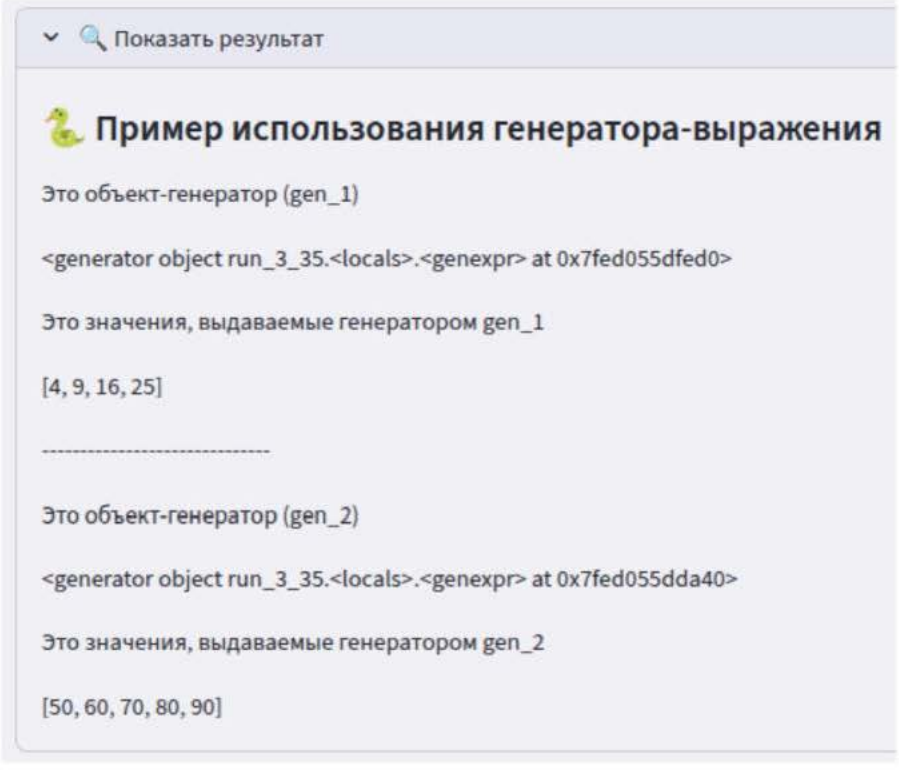


Рис. 3.33. Пример использования генератора-выражения

В этой программе на первом шаге был создан генератор gen_1 без условия. С использованием инструкции «for» организован цикл, в которой переменная «x» получает значения от 2 до 5. Такой генератор будет создавать последовательность чисел, определяемых выражением «x**2». Причем в тот момент, когда генератор gen_1 создан, последовательность самих чисел не сформирована, и они не занимают память. Однако при каждом вызове метода __next__() этого генератора «gen_1.__next__()» в соответствии с выражением «x**2» будет создаваться последовательность таких чисел. Причем при создании нового числа последовательности предыдущее число будет удаляться из памяти. В приведенной выше программе было создано 4 числа последовательности (4, 9, 16, 24), после чего с использованием функции next() была остановлена работа этого генератора.

На втором шаге был создан генератор gen_2 с условием. С использованием инструкции «for» организован цикл, в которой переменная «x» получает значения от 2 до 10. Кроме того задано условие «if x > 4». Такой генератор будет создавать последовательность чисел, определяемых выражением «x*10». В этом случае в цикле, организованном инструкцией

«for», переменная «x» будет получать значения от 2 до 9, а выражение «x*10» будет подсчитываться только для чисел, которые больше, чем 4. В тот момент, когда генератор gen_2 создан, последовательность самих чисел не сформирована, и они не занимают память.

Сами числа будут генерироваться в другом цикле «for elem in gen_2». В этом цикле, переменная «elem» будет получать значения от чисел, которые будет выдавать генератор «gen_2». Обратите внимание, что здесь в инструкции «for» отсутствуют граничные значения начала и окончания цикла, они будут взяты из параметров цикла «for», которые были заданы при инициализации генератора «gen_2». В теле цикла присутствует инструкция print(), которая в цикле выдаст последовательность чисел, формируемых генератором. В приведенной выше программе было создано 5 чисел последовательности (50, 60, 70, 80, 90). Причем при создании нового числа последовательности предыдущее число будет удаляться из памяти.

Благодаря тому, что выражения-генераторы создают итераторы вместо полного списка или множества, они позволяют рационально использовать память в ходе работы программ. Проверим это на примере: сформируем список и множество с числами от 0 до 1 млн., и создадим объект генератор, который будет выдавать ту же последовательность. А потом посмотрим на размер памяти, которое займет каждый из этих объектов. Пример такой программы приведен в листинге 3.36.

Листинг 3.36. Генераторы (оценка объема памяти)

```
# Листинг 3.36
# Импортируем модуль sys
import sys

# Генератор создает список
li_1 = [x for x in range(0, 1000000)]
print('Размер памяти, занятой списком li_1:', sys.getsizeof(li_1))

# Генератор создает множество
s_1 = {x for x in range(0, 1000000)}
print('Размер памяти, занятой множеством s_1:', sys.getsizeof(s_1))

# Генератор-выражение создает итератор
gen_1 = (x for x in range(0, 1000000))
print('Размер памяти, занятой итератором gen_1:', sys.getsizeof(gen_1))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.36. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.34).



Рис. 3.34. Генераторы (оценка объема памяти)

Если подсчитать разницу, то в этом случае генератор займет памяти в 167 773 раза меньше, чем множество.

Как было отмечено выше генераторы в момент вызова создают только одно число генерируемой последовательности. Однако с использованием генераторов можно выдавать и полные последовательности итерируемых значений. Для этого нужно использовать в отношении сгенерированного итератора функции-конструкторы, например, «list()» или «set()». В этом случае будет получена вся последовательность генерируемых значений.

Пример использования функций конструкторов для списков, кортежей и множеств приведен в листинге 3.37.

Листинг 3.37. Использование функций конструкторов для списков, кортежей и множеств

```
# Листинг 3.37
# Создаем список при помощи функции list()
list_1 = list(x for x in range(10))
print('Список list_1:', list_1)

# Создаем кортеж при помощи функции tuple()
tup_1 = tuple(x for x in range(10))
print('Кортеж tup_1:', tup_1)

# Создаем множество при помощи функции set()
set_1 = set((x for x in range(10)))
print('Множество s_1:', set_1)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.37. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.35).

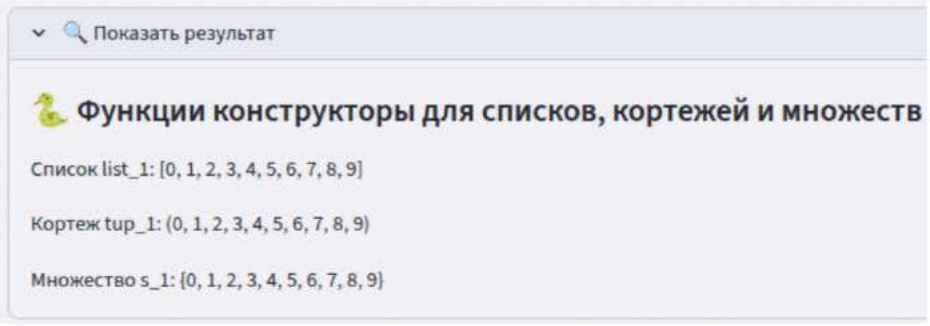


Рис. 3.35. Использование функций конструкторов для списков, кортежей и множеств

Как видно из данного примера, такой подход по функциональности является эквивалентом использования квадратных, круглых и фигурных скобок при создании этих последовательностей традиционным способом.

С использованием генераторов можно формировать и словари, хотя они имеют более сложную структуру «ключ:значение_ключа». При этом конструкция самого генератора словаря выглядит чуть сложнее. Для генератора словаря сначала необходимо сформировать кортеж из пары «ключ:значение_ключа», и уже его использовать в генераторе словаря.

Пример программы, в которой показаны способы генерации словарей из кортежей, приведен в листинге 3.38.

Листинг 3.38. Генерация словарей из кортежей

```
# Листинг 3.38
# Источник ключей и значений для словаря в виде списка кортежей
list_1 = [('Один', 1), ('Два', 2), ('Три', 3)]
print('Источник для словаря в виде списка кортежей')
print('list_1 ->', list_1)

print('Простой генератор словаря')
d_1 = {k: v for k, v in list_1}
print('d_1 ->', d_1)

print('Генератор словаря с условием')
d_2 = {k: v for k, v in list_1 if v != 3}
print('d_2 ->', d_2)

print('Словарь из выражения-генератора и списка кортежей')
d_3 = dict(((k, v) for k, v in list_1))
print('d_3 ->', d_3)

print('Словарь со значением по умолчанию')
d_4 = {k: None for k in 'ABC'}
print('d_4 ->', d_4)
```

```
print('-'*50)

# Источник ключей и значений для словаря из вложенного кортежа
tup_1 = (('Один', 1), ('Два', 2), ('Три', 3))
print('Источник для словаря вложенный кортеж')
print('tup_1 ->', tup_1)

print('Словарь из выражения-генератора и вложенного кортежа')
d_5 = dict(((k, v) for k, v in tup_1))
print('d_5 ->', d_5)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.38. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Источник для словаря в виде списка кортежей
list_1 -> [('Один', 1), ('Два', 2), ('Три', 3)]
Простой генератор словаря
d_1 -> {'Один': 1, 'Два': 2, 'Три': 3}
Генератор словаря с условием
d_2 -> {'Один': 1, 'Два': 2}
Словарь из выражения-генератора и списка кортежей
d_3 -> {'Один': 1, 'Два': 2, 'Три': 3}
Словарь со значением по умолчанию
d_4 -> {'A': None, 'B': None, 'C': None}
--------------------------------------------------
Источник для словаря вложенный кортеж
tup_1 -> (('Один', 1), ('Два', 2), ('Три', 3))
Словарь из выражения-генератора и вложенного кортежа
d_5 -> {'Один': 1, 'Два': 2, 'Три': 3}
```

Из данного примера видно, что в генераторе-выражении в итератор заносились не пары значений по отдельности, а кортежи с парами этих значений. Именно из них потом конструктор типа и сформировал нам готовый словарь. Сами кортежи с парами «ключ:значение_ключа» могут быть оформлены в виде списка, или вложенного кортежа. Лучше такие кортежи оформлять в виде списка, поскольку пары скобок для списков - «[( )]» читаются лучше и понятней, чем пары скобок для вложенных кортежей - «(( ))».

3.5.4. Функции-генераторы и ключевое слово yield

На данный момент мы познакомились с целым рядом генераторов: генераторами списков, словарей, множеств, а также с генераторами-выражениями. Первые три вида генераторов выдают нам готовые наборы значений целиком, в то время как генераторы-выражения возвращают итераторы, позволяя экономить память и производить дополнительные вычисления между операциями получения результатов. Однако в качестве генераторов

последовательностей могут выступать и пользовательские функции. В Python такие функции называют «функции-генераторы».

Функция-генератор (от англ. generator function) – это специальная функция, которая использует инструкцию yield, способную приостанавливать выполнение функции с сохранением текущего состояния до следующего обращения к ней.

По внешним признакам и синтаксису функция-генератор это обычная функция, которая имеет всего одно отличие: вместо завершающей инструкции return, которая используется в обычных пользовательских функциях, функция-генератор завершается инструкцией yield.

Инструкция return в пользовательских функциях всегда является последней инструкцией и выполняет два действия: возвращает результат работы функции, полностью завершает ее работу. Инструкция yield то же возвращает результат работы функции, но лишь временно приостанавливает ее работу и сохраняет текущее состояние. Позже, при следующем обращении к такой функции, она продолжит свою работу из текущего состояния. На основе такой функции создает новый объект-генератор со встроенным итератором, а итераторы, как мы знаем, имеют в своем арсенале метод __next__, или могут быть обработаны функцией next(). Вызов этого метода приводит к снятию паузы в функции-генераторе и продолжению ее выполнения с места приостановки. Схематично взаимодействие функции-генератора и объекта-генератора с итератором представлено на рис. 3.36.

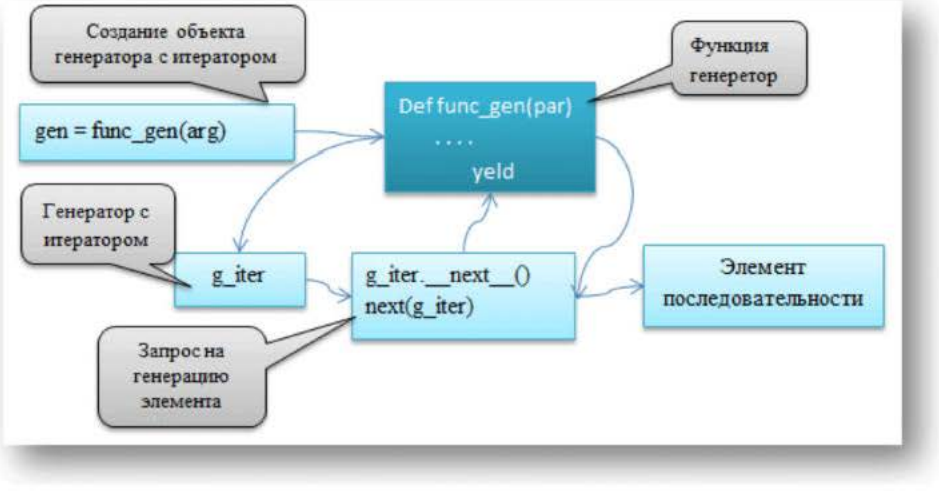


Рис. 3.36. Схема взаимодействия функции-генератора и объекта-итератора

Пример программы, в которой показана работа функции-генератора, приведен в листинге 3.39.

Листинг 3.39. Пример работы функции-генератора

```
# Листинг 3.39
# Обычная функция Python
def list_func(n):
    list_1 = []  # Пустой список
    for i in range(1, n + 1):  # Запускаем цикл
```

```
        list_1.append(i)  # Добавляем элемент в список
    return list_1  # Возвращаем готовый список


my_list = list_func(3)
print('Объект list_func ->', list_func)
print('Список из обычной функции list_func(3) ->', my_list, end='\n\n')


# Функция-генератор возвращает объект-генератор со встроенным
итератором.
def gen_func(n):
    for i in range(1, n + 1):  # Запускаем цикл
        yield i  # Возвращаем значение и приостанавливаем работу


print('Объект gen_func ->', gen_func)
g_iter_1 = gen_func(2)  # Генератор для формирования 2-х значений с
итератором
g_iter_2 = gen_func(3)  # Генератор для формирования 3-х значений с
итератором


print('Многократное обращение к функции-генератору через g_iter_1')
print('1-е обращение к g_iter_1:', next(g_iter_1, 'Пусто'))
print('2-е обращение к g_iter_1:', next(g_iter_1, 'Пусто'))
print('3-е обращение к g_iter_1:', next(g_iter_1, 'Пусто'))
print()

print('Многократное обращение к функции-генератору через g_iter_2')
print('1-е обращение к g_iter_2:', next(g_iter_2, 'Пусто'))
print('2-е обращение к g_iter_2:', next(g_iter_2, 'Пусто'))
print('3-е обращение к g_iter_2:', next(g_iter_2, 'Пусто'))
print('4-е обращение к g_iter_2:', next(g_iter_2, 'Пусто'))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.39. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Объект list_func -> <function list_func at 0x000000000282FAF8>
Список из обычной функции list_func(3) -> [1, 2, 3]

Объект gen_func -> <function gen_func at 0x000000000282FCA8>
Многократное обращение к функции-генератору через g_iter_1
```

```
1-е обращение к g_iter_1: 1
2-е обращение к g_iter_1: 2
3-е обращение к g_iter_1: Пусто

Многократное обращение к функции-генератору через g_iter_2
1-е обращение к g_iter_2: 1
2-е обращение к g_iter_2: 2
3-е обращение к g_iter_2: 3
4-е обращение к g_iter_2: Пусто
```

В данной программе на первом шаге создана традиционная пользовательская функция list_func(), которая создает и возвращает список. Она имеет один параметр «n», который определит – сколько элементов списка нужно сформировать. При однократном обращении к этой функции была дана команда - сформировать 3 элемента списка «my_list = list_func(3)». Как видно из результатов работы программа, эта функция при одном обращении вернула все три элемента.

На втором шаге была создана функция генератор – с именем gen_func(n). Она так же имеет один параметр «n», который определит – сколько элементов списка нужно сформировать. Далее дважды было выполнено обращение к этой функции:

- g_iter_1 = gen_func(2);
- g_iter_2 = gen_func(3).

Однако, не смотря на то, что мы два раза обратились к функции-генератору и передали ей аргументы 2 и 3, она не вернула ни одного значения. Все, что было сделано, это создано два новых объекта: g_iter_1 и g_iter_2. Теперь они являются объектами-генераторами со встроенным итератором, при этом объект g_iter_1 будет способен выдавать два значения, а объект g_iter _2 три значения.

Поскольку g_iter_1 и g_iter_2 это итерируемые объекты, то они могут быть обработаны функцией nex(), которая позволяет извлекать элементы последовательности один за другим.

На последнем шаге с использованием объектов g_iter_1 и g_iter_2 все элементы последовательностей, которые формировались в функции-генераторе, были выведены на печать (на экран). При этом мы обращались к функции-генератору не напрямую, а через созданные с ее помощью объекты-генераторы.

В отличие от обычных функций, которые возвращают значение и сразу же завершают работу, функции-генераторы лишь приостанавливают процесс, сохраняя информацию о своем состоянии. Они возобновляют свою работу по требованию в момент вызова метода __next__, или функции next() объектов-генераторов, которые созданы на основе этой функции. Функции-генераторы могут поочередно выдавать элементы последовательностей в различных фрагментах программы. Они не создают всю последовательность значений за один раз, что позволяет рационально использовать память компьютера при обработке очень больших массивов данных

Также следует отметить, что при каждом отдельном вызове функции-генератора она создает совершенно новый объект-генератор со своим встроенным итератором. Это дает возможность одновременного использования одной и той же функции-генератора из разных точек программы.

3.5.5. Краткие итоги раздела 3.5

7. Итерируемый объект – это любой объект, набор элементов которого можно обрабатывать многократно и по одному за раз. В Python все такие объекты имеют встроенный метод __iter__() , который возвращает итератор объекта.
8. Итератор – это специальный объект, в котором реализован метод __next__(), позволяющий при каждом новом вызове получать следующий элемент итерируемого объекта. Итераторы возвращаются, например, методом __iter__(), встроенной функцией iter(object), выражениями-генераторами и другими инструментами для их получения. Основным предназначением итераторов следует считать экономию памяти за счет генерации значений по одному за раз, а не использования всего набора значений сразу.
9. Некоторые объекты (например, файловые объекты) являются итерируемыми, они имеют встроенный итератор и метод __next__().
10. Некоторые объекты (например, списки), являются итерируемыми объектами, но не имеют встроенного итератора, поэтому метод __next__() у них отсутствует. Для получения итератора такие объекты требуют вызова метода __iter__() или использования функции iter(). Это связано с тем, что списки предназначены для многократного использования с возможностью участия сразу в нескольких итерациях. Однако итераторы такой возможности не предоставляют, так как они относятся к одноразовым инструментам, которые с каждым вызовом метода __next__() опустошаются и становятся непригодными для дальнейших итераций.
11. Все доступные в Python итерационные инструменты используют в своей работе специальный протокол итераций. Большинство инструкций используют этот протокол по умолчанию, но при необходимости имеется возможность использовать правила этого протокола в ручном режиме.
12. Выражение-генератор – это конструкция на базе цикла «for», которая применяет заданное выражение к каждому элементу итерируемого объекта, В отличие от генераторов списков, множеств и словарей, которые возвращают их целиком за раз, выражения-генераторы возвращают только итераторы итерируемых объектов, позволяя экономить память.
13. Функция-генератор – это специальная функция, которая использует инструкцию «yield», способную приостанавливать выполнение функции с сохранением текущего состояния до следующего обращения к ней. Вызов этой функции возвращает специальный объект-генератор со встроенным итератором. Этот объект имеет метода __next__(), который позволяет получать по одному элементу генерируемой последовательности за раз.

3.5.6. Вопросы для самоконтроля к разделу 3.5



- Какое действие выполняет функция iter() в итераторах:
  - создает объект-итератор из последовательности;
  - «вычисляет» и возвращает следующий элемент итератора;
  - позволяет определить, поддерживает ли объект итерации.
- Какое действие выполняет функция next() в итераторах:
  - «вычисляет» и возвращает следующий элемент итератора;
  - создает объект-итератор из последовательности;
  - позволяет определить, поддерживает ли объект итерации.

- Какое действие выполняет метод __iter__() в итераторах:
  - позволяет определить, поддерживает ли объект итерации;
  - создает объект-итератор из последовательности;
  - «вычисляет» и возвращает следующий элемент итератора.
- Какой объект имеет встроенный итератор:
  - файловый объект;
  - объект список;
  - объект картеж;
  - объект множества.
- Какое назначение имеют генераторы:
  - За один проход генерируют один элемент последовательности;
  - За один проход генерируют все элементы последовательности;
  - Создают функцию для генерации последовательностей;
  - Определяют, являются ли объекты итераторами.
- Какой объект будет занимать минимальное место в памяти компьютера:
  - Объект - Итератор списков;
  - Объект – Списки;
  - Объект – множества;
  - Объект – картеж.
- Что получим при обращении к функции-генератору:
  - объект-генератор со встроенным итератором;
  - тип данных, которые будет генерировать функция-генератор;
  - один элемент из генерируемой последовательности;
  - все элементы генерируемой последовательности.
- Что выдает объект-генератор, созданный на основе функции-генератора при однократном обращении к нему:
  - один элемент из генерируемой последовательности;
  - тип генерируемых объектов;
  - количество генерируемых элементов;
  - все элементы генерируемой последовательности.
- Какое действие выполняет функция-генератор:
  5. Возвращает только один элемент последовательности и ждет следующего обращения;
  6. Возвращает только один элемент последовательности и завершает работу;
  7. Возвращает все элементы последовательности и ждет следующего обращения;
  8. Возвращает все элементы последовательности и завершает работу.
- Для чего нужна инструкция yield у функции-генератора:
  - Возвращает только один элемент последовательности и переводит функцию в режим ожидания;
  - Возвращает только один элемент последовательности и завершает работу функции;
  - Возвращает все элементы последовательности и ждет следующего обращения;
  - Возвращает все элементы последовательности и завершает работу функции.

3.6.Обработка исключений в Python с использованием инструкций try, except и finally

В языке Python существует множество встроенных исключений, которые вызываются, когда программа сталкивается с ошибкой (что-то в программе идет не так). Когда такое исключение возникает, интерпретатор Python останавливает текущий процесс и передает управление новому процессу для обработки данного исключения. Если его не удается обработать, то программа получит аварийное завершение.

В Python исключения обрабатываются при помощи инструкции «try». Эта инструкция имеет следующий синтаксис:

```
try:
   <Базовый код>
except:
   <Неожиданная ошибка в базовом коде>
else:
   <Базовый код выполнился без ошибок>
finally:
   <Код, который выполнится в любом случае>
```

Критическая инструкция, которая может вызвать исключение (ошибку), помещается внутрь блока «try». Инструкции (программный код), при помощи которых это исключение (ошибка) будет обработано, помещается внутрь блока «except». В блоке «else» (не обязательный блок) можно поместить инструкции или сообщения, подтверждающие, что в базовом коде ошибок не произошло. В заключительную инструкцию «finally» (не обязательная) можно поместить код, который выполнится в любом случае (не зависимо от наличия или отсутствия ошибок). Пример программы, в которой показано использование инструкции «try», приведен в листинге 3.40.

Листинг 3.40. Использование инструкции «try»

```
# Листинг 3.40
x = 0
try:
    y = 100 / x
    print('y =', y)
except ValueError:
    print("Это не число!")
except ZeroDivisionError:
    print("На ноль делить нельзя!")
except:
    print("Неожиданная ошибка")
else:
    print("Код выполнился без ошибок")
finally:
    print("Я выполняюсь в любом случае!", 'x =', x)
    print('-'*30)
```

```
x = 'Это строка символов'
try:
    y = 100 / x
    print('y =', y)
except ValueError:
    print("Это не число!")
except ZeroDivisionError:
    print("На ноль делить нельзя!")
except:
    print("Неожиданная ошибка")
else:
    print("Код выполнился без ошибок")
finally:
    print("Я выполняюсь в любом случае!", 'x =', x)
    print('-' * 30)

x = 10
try:
    y = 100 / x
    print('y =', y)
except ValueError:
    print("Это не число!")
except ZeroDivisionError:
    print("На ноль делить нельзя!")
except:
    print("Неожиданная ошибка.")
else:
    print("Код выполнился без ошибок")
finally:
   print("Я выполняюсь в любом случае!", 'x =', x)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.40. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.37).

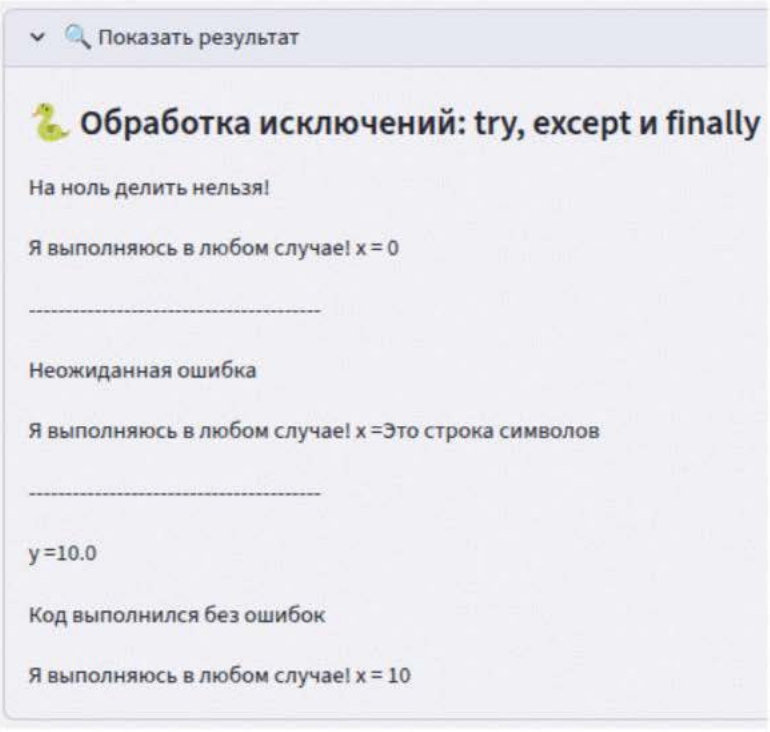


Рис. 3.37. Использование инструкции «try»

Как видно из данного примера, не смотря на то, что в программе были ошибки, она доработала до конца, поскольку все возможные ошибки были перехвачены с использованием инструкции «try». Обратите внимание, что после инструкции «except» можно указать и тип ошибки, которая возникла при выполнении базовой инструкции. В данном примере были использованы расшифровки следующих типов ошибок:

- ValueError (недопустимое значение числа),
- ZeroDivisionError (деление на ноль).

В Python используется достаточно много указаний на типы ошибок, при необходимости все их можно найти в оригинальной документации на официальном сайте.

Важно! Если присутствуют инструкции «except» с указанием типа ошибки, то инструкция «except» без указания типа ошибки должна быть последней.



3.6.1. Вопросы для самоконтроля к разделу 3.6

- Для чего в Python используется инструкция – try:
  - для обработки исключений в случае возникновения ошибок;
  - для сохранения значений в объектах при появлении ошибок;
  - для вывода сведений об ошибке и аварийной остановки программы;
  - для буферизации возможных ошибок.

3.7.Менеджеры контекста (инструкция with/as)

Менеджеры контекста необходимы для выделения определенных ресурсов и их освобождение, когда необходимость в них отпадает. Конструкции «with/as» обычно используется при работе с файлами. Например, открытый файл будет гарантированно закрыт после того, как с ним будут выполнены заданные операции. Фактически конструкция «with/as» используется для «оборачивания» блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем «try...except...finally». В упрощенном виде синтаксис такой конструкции выглядит следующим образом

```
with <менеджер контекста> as <переменная>:
    <список инструкций языка Python>
```

Чаще всего такая конструкция используется при открытии файлов пред чтением информации из файла, или для записи в файл некоторых данных, например:

```
with open(path, 'w') as file:
  file.write(data)
```

Преимущество использования ключевого слова «with» перед вызовом функции open() в том, что функция file.close(), которая закроет файл, вызовется автоматически и освободит занятые файлом ресурсы после того, как отработает код. На первый взгляд здесь экономится лишь одна строка программного кода, но это не совсем так, главной особенностью конструкции «with» является то, что финальный код (в данном случае «file.close()») вызывается гарантированно, даже в том случае, если при обработке интерпретатором строк внутри конструкции произойдет ошибка.

Важно заметить, что Python позволяет разработчикам писать собственные контекстные менеджеры. Этой возможностью необходимо пользоваться в тех случаях, когда в программе есть фрагменты кода, выполнение которых критично для ее стабильной работы. В этом пособии мы не будем создавать собственный контекстный менеджер, однако будем использовать такую конструкцию в примерах следующего раздела при работе с файлами.

3.7.1. Вопросы для самоконтроля к разделу 3.7



9. 1. Для чего в Python используется инструкция – «with/as»:

- Для выделения и освобождения ресурсов при выполнении программ;
- Для организации цикла при записи информации в файл;
- Для организации цикла при чтении информации из файла;
- Для организации разветвления в программе.

3.8.Ввод – вывод данных в Python

Ввод данных в программу и их вывод важны в программировании. Без ввода данных программы не понимали бы, что нужно обрабатывать (исключая случаи, когда в них самих генерируются какие-либо значения). Вывод позволяет увидеть, использовать, передать дальше результат работы программы. Мы уже встречались с функцией print(). Она отвечает за вывод данных, по умолчанию они выводятся на экран, но можно перенаправить вывод в файл или на принтер. Достаточно часто нужно сохранить информацию в файле, прочитать из файла, или пользователю нужно ввести в программу какие то данные с клавиатуры. В этих случаях без инструкций ввода-вывода не обойтись. В Python предусмотрены следующие возможности ввода-вывода данных:

- функция open() – для создания объектов, взаимодействующих с файлами;
- функция inpun() – для ввода данных с клавиатуры;
- функция print() – для вывода информации на экран (возможно в файл или на принтер).

Рассмотрим все эти возможности.

3.8.1.Функция open() для работы с файлами

В компьютерах вся информация хранится в файлах: документы, изображения, данные, видео ролики, аудио записи и т.п. Средства Python позволяют работать с такими файлами: открыть (закрыть) файл, прочитать данные из файла, записать в файл результаты работы программы. Итак, начнем. Прежде, чем работать с файлом, его надо открыть. С этим замечательно справляется встроенная функция open(), которая имеет следующий синтаксис:

```
obj_file = open('<имя_файла>', '<режим>', encoding='<кодировка>' )
```

Эта функция создает объект obj_file, который позволит выполнять манипуляции с файлом, а передать в нее нужно следующие аргументы:

- <имя_файла> – это имя файла (с расширением), который будет открыт для чтения, или для записи в него информации;
- <режим> - это аргумент, в котором задается режим работы с файлом;
- encoding=<кодировка> - тип кодировки символов (используется для текстовых файлов).

Функция open() имеет много параметров, но нам пока важны три аргумента, которые можно передать в данную функцию. Первый аргумент - это имя файла, которое может включать и путь к файлу (путь к файлу может быть абсолютным или относительным). Второй аргумент, это режим, в котором файл будет открыт. Возможны следующие режимы работы с файлом:

- r открытие на чтение (является значением по умолчанию);
- w - открытие на запись, содержимое файла удаляется (если файла не существует, создается новый);
- x - открытие на запись, если файла не существует (иначе исключение);
- a - открытие на до запись, информация добавляется в конец файла;

- b - открытие файла в двоичном режиме;
- t - открытие файла в текстовом режиме (является значением по умолчанию);
- (+) - открытие на чтение и запись.

Режимы могут быть объединены, то есть, к примеру, ‘rb’ - чтение в двоичном режиме. По умолчанию будет использоваться режим ‘rt’. И последний аргумент - кодировка (encoding) нужен только в текстовом режиме работы с файлом. Этот аргумент задает кодировку символов (например, utf-8).

Объект, который создан на основе функции open(), имеет три важных встроенных метода:

- obj_file.write – позволяет сохранить информацию в файле (записать);
- obj_file.read – позволяет вывести информацию из файла (прочитать).
- obj_file.close – позволяет закрыть файл.

Сначала познакомимся с первым методом, который обеспечивает запись информации в файл. Пример программы, в которой показано использование метода «write», приведен в листинге 3.41.

Листинг 3.41. Использование функций «open» и «write»

```
# Листинг 3.41
import os
text = 'Это первая строка текста\nЭто вторая строка текста\nЭто третья строка
текста\n'
file_txt = open('my_file.txt', 'w')
file_txt.write(text)
file_txt.close()

# Получить информацию о файле
my_file = open("my_file.txt")
print(my_file)

if os.path.isfile('my_file.txt'):
    os.remove('my_file.txt')
```

В этой программе была создана текстовая строка text, где символы «\n» является признаком перехода к новой строке. Затем с использование функции open() создан объект file_txt. В функции open() задано имя файла, в который будет записываться информация, и режим записи – «w». Затем, с использованием метода write «file_txt.write(text)», текстовая строка была записана в файл. Следующая инструкция выполнила закрытие файла. И в заключение файл был снова открыт и на экран выведена информация о созданном файле.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.41. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.38).

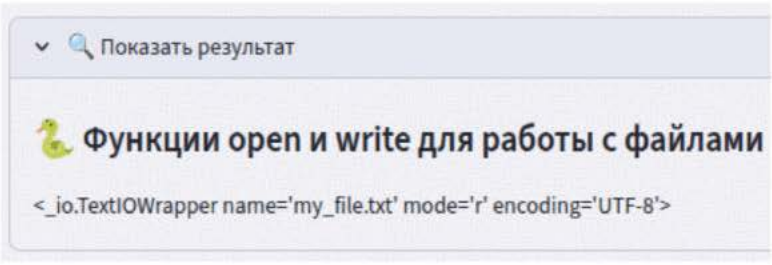


Рис. 3.38. Использование функций «open» и «write»

Из полученной строки видно, что файл имеет имя - my_file.txt, открыт в режиме - «r» (только для чтения). Информация в этом файле имеет кодировку символов - «UTF-8» (при выполнении данного модуля в ОС Linux). Если выполнить данный модуль на компьютере с ОС Windows, то в этой операционной системе по умолчанию будет использована кодировка символов - «CP1251».

После запуска данной программы мы не увидели содержание файла с именем my_file.txt, который был создан в текущем каталоге. Однако мы можем заглянуть в содержимое этого файла с использованием любого текстового редактора. В Windows по умолчанию средством просмотра и редактирования текстовых файлов является приложение – Блокнот, воспользуемся этим приложением. Содержимое файла my_file.txt, открытое в Блокноте, приведено на рис. 3.39.

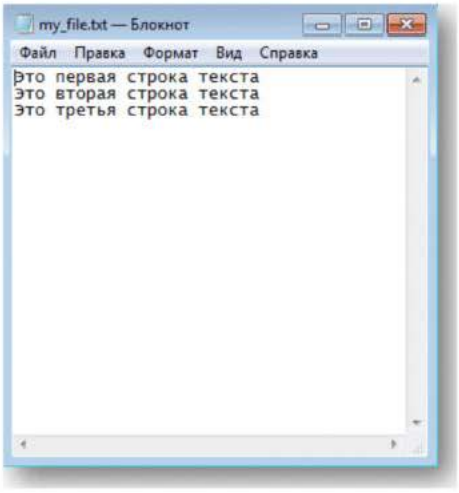


Рис. 3.39. Содержимое файла my_file.txt в окне блокнота Windows

Как видно из данного рисунка, в файле действительно записано три сроки, которые содержались в переменной text. Они записаны с кодировкой символов «CP1251» и приложение Windows успешно отобразила эти строки.

При работе с файлами функцию open() предпочтительнее использовать с оператором контекстного менеджера «with … as», при использовании которого файл принудительно закрывать не нужно. Здесь не будем останавливаться на тонкостях работы контекстного менеджера. Отметим лишь то, что применение инструкции «with ... as» это гарантия того,

что программный код, который следует за этой инструкцией, выполнится в любом случае (даже если в этом коде возможно появление ошибок). Эта конструкции чаще всего используется как раз для открытия файлов, она имеет следующий синтаксис:

```
with open('<имя_файла>', '<режим> ', encoding='<кодировка>') as obj_file:
```

Эта инструкция создает объект obj_file, который позволит выполнять манипуляции с файлом, а передать в нее нужно следующие аргументы:

- <имя_файла> – это имя файла (с расширением), который будет открыт для чтения, или для записи в него информации;
- <режим> - это аргумент, в котором задается режим работы с файлом;
- encoding=<кодировка> - тип кодировки символов (используется для текстовых файлов).

Пример программы, в которой показано использование инструкции «with ... as» для открытия файла и записи в него информации, приведен в листинге 3.42.

Листинг 3.42. Использование инструкции «with ... as»

```
# Листинг 3.42
import os
text = 'Это первая строка текста\nЭто вторая строка текста\nЭто третья строка
текста\n'
with open('my_file.txt', 'w', encoding="utf-8") as file_txt:
    file_txt.write(text)

# Получить информацию о файле
my_file = open("my_file.txt")
print(my_file)
if os.path.isfile('my_file.txt'):
    os.remove('my_file.txt')
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.42. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.30).

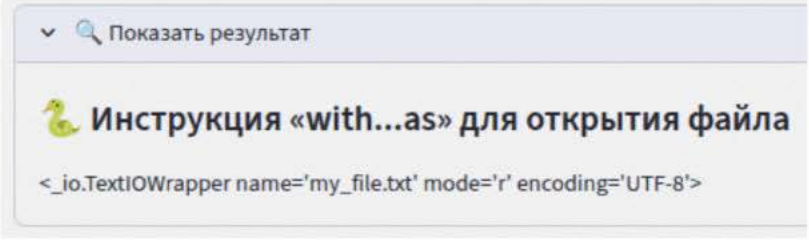


Рис. 3.40. Использование инструкции «with ... as»

Как видно из этих результатов, они аналогичны результатам предыдущего примера. Способ открытия файла с использованием инструкции «with ... as» является более предпочтительным, так как и программный код короче, и работает он надежнее. Например, при сбоях записи файла на диск, программа не выдаст ошибки и продолжит свою работу.

Теперь рассмотрим метод «read», который обеспечивает чтение информации из файла. Пример программы, в которой показано использование метода «read» вместе с инструкции «with ... as», приведен в листинге 3.43.

Листинг 3.43. Использование метода «read» с инструкции «with ... as»

```
# Листинг 3.43
import os

text = 'Это первая строка текста\nЭто вторая строка текста\nЭто третья строка
текста\n'
with open('my_file.txt', 'w', encoding="utf-8") as file_txt:
    file_txt.write(text)

with open('my_file.txt', 'r') as file_txt:
    data_text = file_txt.read()

# Получить информацию о файле
print(file_txt)
print('-'*65)

# Вывести содержимое файл построчно
for line in data_text.splitlines():
    print(line)

if os.path.isfile('my_file.txt'):
    os.remove('my_file.txt')
```

В этой программе с использование функции open() и инструкции «with ... as» создан объект «file_txt». В функции open() задано имя файла, из которого будет прочитана информация, и режим чтения – «r» (открыт только для чтения). Затем, с использованием метода «read» «file_txt.read()», в объект «data_txt» было прочитано (загружено) все содержимой файла.

На следующем шаге мы просто вывели информацию об открытом файле (объект file_txt).

В заключение модуля находятся инструкции построчного вывода информации из открытого файла. В этот файл было записано три строки, и выводить этот текст так же будем построчно. У объекта «data_txt» есть метод разбивки информации на строки (splitlines).

Здесь используется уже знакомая нам инструкция «for», которая организует цикл. В этом цикле построчно извлекается информация из объекта «data_text», куда был загружен файл, и эти строки выводятся на печать.

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.43. Результаты работы модуля можно увидеть, открыв контейнер «Показать

результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.31).

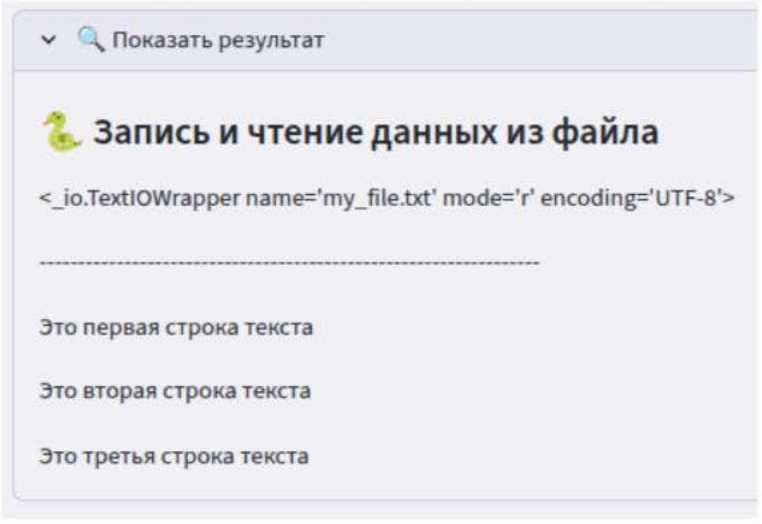


Рис. 3.41. Использование метода «read» с инструкции «with ... as»

На экран были выведены те же три строки, которые были записаны в файл. Обратите внимание, что файл записан с кодировкой символов CP1251, а Python работает с кодировкой UTF-8. Не смотря на это, информация была корректно выведена на экран. В Python встроен механизм, который по возможности сам преобразует символы в нужную кодировку. Однако тут есть «подводные камни», достаточно часто можно столкнуться с ситуацией, когда вместо текста вдруг начнут выплывать совершенно незнакомые символы, или знаки вопросов вместо текстовой информации. Если мы этот же файл попытаемся напрямую прочитать средствами IDE PyCharm, то получим следующую картину (рис. 3.42).

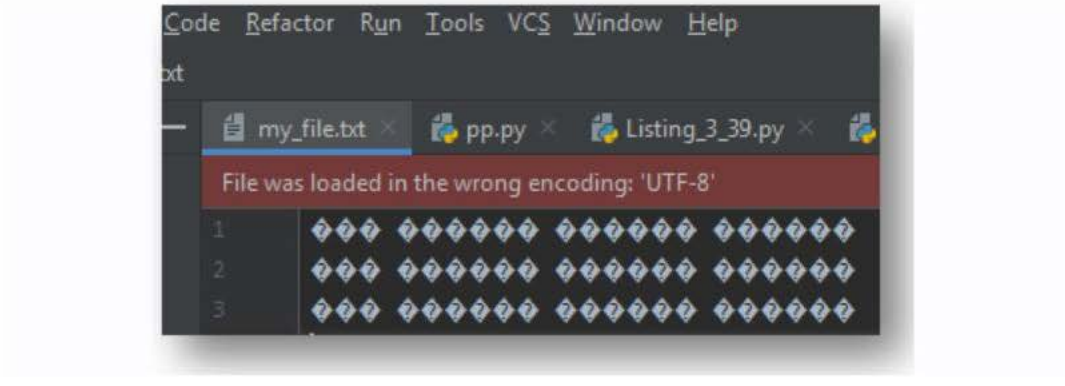


Рис. 3.42. Содержимое файла my_file.txt в окне IDE PyCharm

Как видно из данного рисунка, вместо трех строк текста, которые были записаны в файл, мы получили три строки совершенно непонятных символов, и сообщение, что файл загружен с кодировкой, отлично от UTF-8.

Разберемся в чем же тут дело. Мы уже знаем, что Python работает с кодировкой символов UTF-8, а Windows записал информацию в этот файл с кодировкой CP1251, что видно из предыдущей распечатки. Инструментальная среда IDE PyCharm настроена на работу с Python, она ожидала получение данных в кодировке UTF-8 и, не получив ее, не смогла корректно вывести текстовое содержимое файла. Чтобы избежать подобных ситуаций у функции open() есть параметр, который принимает значение аргумента - encoding, то есть тип кодировки символов, в которой нужно записывать или читать информацию из файлов. Скорректируем приведенные выше примеры и зададим кодировку символов в явном виде. Пример такого программного модуля приведен в листинге 3.44.

Листинг 3.44. Использование кодировки символов в явном виде

```
# Листинг 3.44
import os
# Запись информации в файл
text = 'Это первая строка текста\nВ этом программном модуле\nиспользуется
кодировка UTF-8\n'
with open('my_file.txt', 'w', encoding="utf-8") as file_txt:
    file_txt.write(text)

# Чтение информации из файла
with open('my_file.txt', 'r', encoding="utf-8") as file_txt:
    data_text = file_txt.read()

# Получить информацию о файле
print(file_txt)
print('-'*30)

print(data_text)  # Вывести содержимое файл
print('-'*30)

# Вывести содержимое файл
for line in data_text.splitlines():
    print(line)

if os.path.isfile('my_file.txt'):
    os.remove('my_file.txt')
```

В этом программном модуле сначала была записана информация в файл, потом мы ее прочитали и вывели на экран. Вывод содержимо файла был выполнен двумя способами: целиком и построчно. Также вывели сведения о созданном файле. Обратите внимание, что и при записи данных в файл, и при чтении была принудительно задана кодировка символов UTF-8.

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел

«Листинг 3.44. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.43).

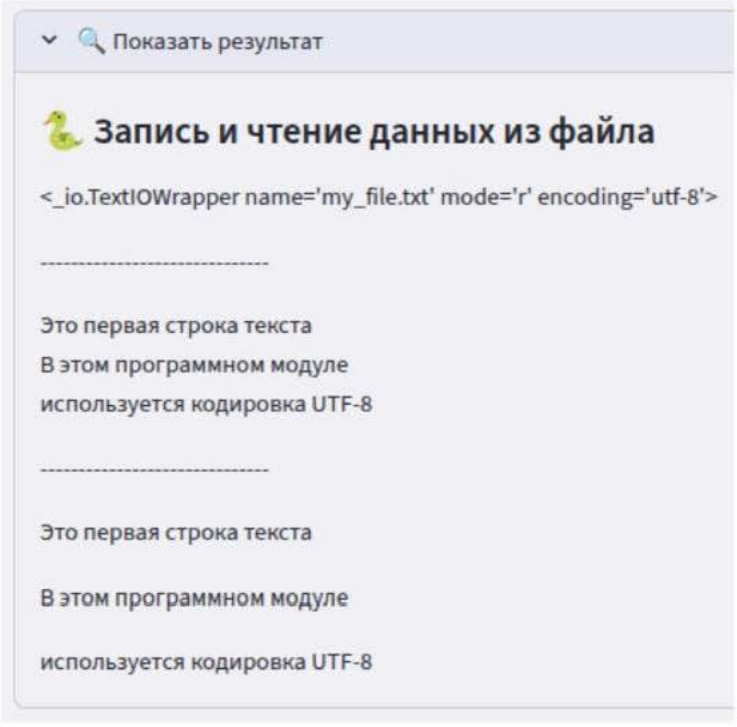


Рис. 3.43. Использование кодировки символов в явном виде

Из первой строки распечатки видно, что в данном файле символы имеют кодировку UTF-8, и все строки корректно были выведены на экран. Посмотрим теперь, как отобразит данный файл инструментальная среда IDE PyCharm (рис. 3.44).

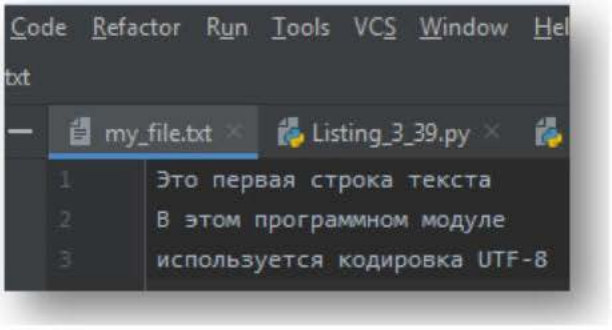


Рис. 3.44. Содержимое файла my_file.txt с кодировкой UTF-8 в окне IDE PyCharm

Как видно из данного рисунка, в этом случае информация из файла была выведена совершенно корректно, так как символы имеют ту кодировку, которую и ожидает PyCharm. А что же теперь будет в Windows, как там будут выглядеть символы, ведь там используется совершенно другая кодировка? Откроем этот файл в Windows приложением Блокнот (рис. 3.45).

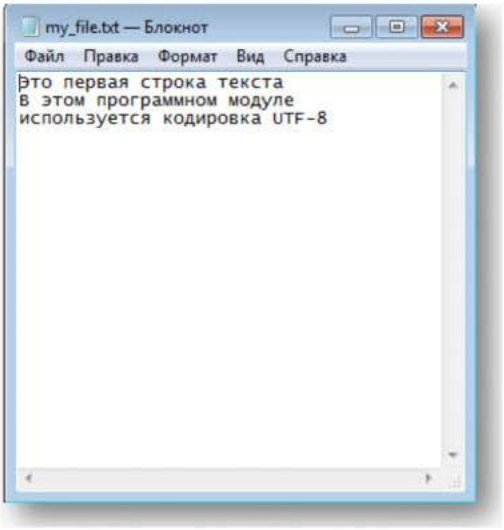


Рис. 3.45. Содержимое файла my_file.txt с кодировкой UTF-8 в окне блокнота Windows

Как видно из данного рисунка и Windows совершенно корректно прочитал и вывел содержание данного файла, не смотря на то, что он по умолчанию использует другую кодировку.

Из всего вышесказанного следует вывод, что для того, чтобы не возникало проблем с искажением символов, необходимо в функции open() явно указывать тип кодировки символов.

3.8.2.Функция input() для ввода данных

Получение и обработка информации — один из важнейших элементов любого языка программирования, особенно если речь идет о получении информации от пользователей. За ввод в программу данных с клавиатуры в Python отвечает функция input(). Когда вызывается эта функция, программа останавливает свое выполнение и ждет, когда пользователь введет текст. После этого, когда он нажмет Enter, функция input() заберет введенный текст и передаст его программе, которая уже будет обрабатывать его согласно своим алгоритмам.

Если в интерактивном режиме ввести команду input(), то ничего интересного вы не увидите. Компьютер будет ждать, когда вы что-нибудь введете и нажмете Enter или просто нажмете Enter. Если вы что-то ввели, это сразу же отобразиться на экране.

По умолчанию функция input() конвертирует всю получаемую информацию в строку. Даже если вы набираете последовательность цифр, пытаясь ввести число, они тоже изначально являются строками. Если введенное число будет участвовать в арифметических операциях, то такую строку нужно принудительно преобразовывать в число.

Пример программы, которая демонстрирует использование функции input(), приведен в листинге 3.45 (этот модуль будет работать только в инструментальной среде Python).

Листинг 3.45. Использование функции input() для ввода данных

```
# Листинг 3.45
# Строковое значение
text_1 = input('Ведите любой текст:')
print(text_1)
print('Тип объекта text_1:', type(text_1))

# Целое число
num_1 = input('Ведите любое число:')
print(num_1)
print('Тип объекта num_1 после ввода:', type(num_1))
num_1 = int(num_1)
print('Тип объекта num_1 после преобразования в целое :', type(num_1))


# Вещественное число
num_2 = input('Ведите любое число с десятичной точкой:')
print(num_2)
print('Тип объекта num_2 после ввода:', type(num_2))
num_2 = float(num_2)
print('Тип объекта num_2 после преобразования :', type(num_2))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.45. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.36).

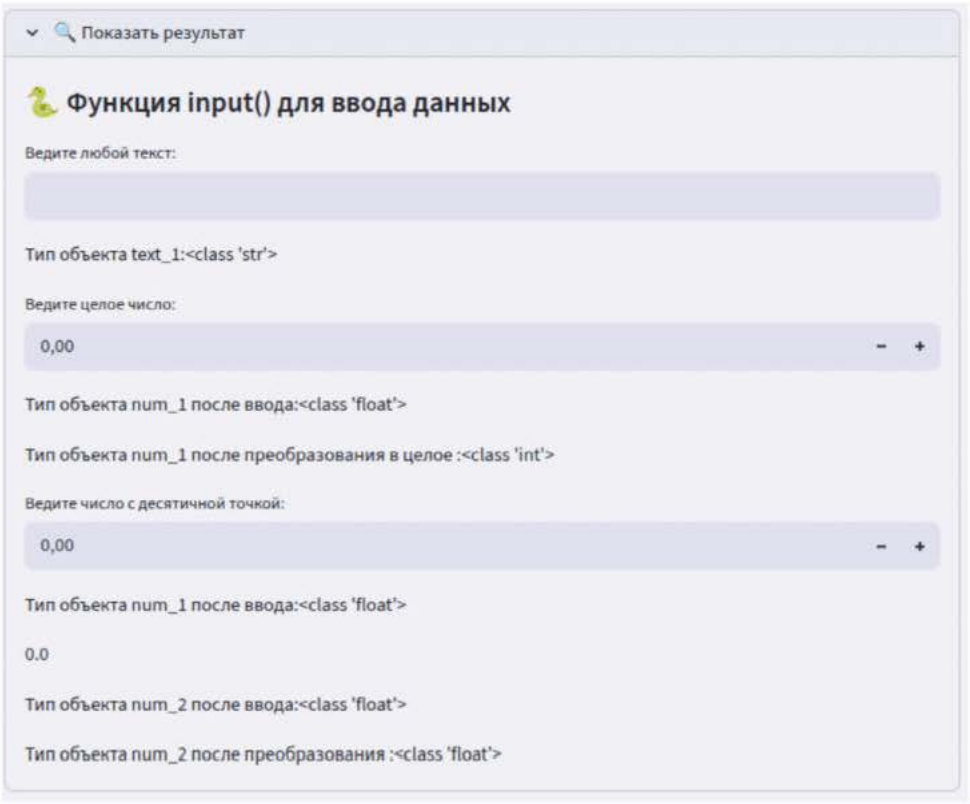


Рис. 3.46. Использование функции input() для ввода данных

Как видно из данного примера, после преобразования введенных данных соответствующими функциями, строковый тип данных преобразуются к данным числового типа.

Если пользователь ввел некорректные данные, например, при вводе числа ввел недопустимый символ, то в момент преобразование такой строки в число программа выдаст ошибку. Что бы этого не происходило, нужно функцию input() оборачивать инструкцией «try: … except». Пример программы, которая демонстрирует использование функции input() совместно с инструкцией «try: … except», приведен в листинге 3.46 (этот модуль будет работать только в инструментальной среде Python).

Листинг 3.46. Использование функции input() совместно с инструкцией «try: … except»

```
# Листинг 3.46
# Вещественное число
num_2 = input('Ведите любое число с десятичной точкой:')
print(num_2)
try:
    num_2 = float(num_2)
    print('Тип объекта num_2 после ввода:', type(num_2))
    print('Тип объекта num_2 после преобразования :', type(num_2))
```

```
except ValueError:
    print("Это не число! Введен недопустимый символ!")
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.46. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.47).

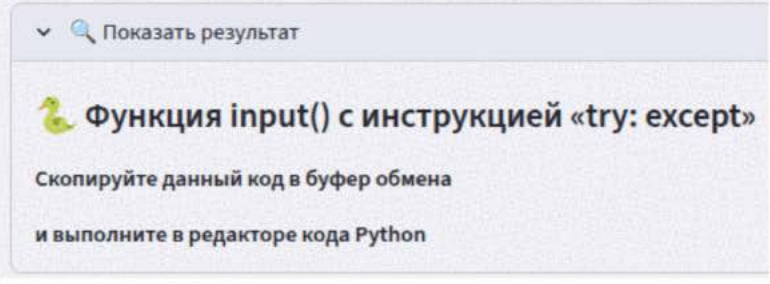


Рис. 3.47. Использование функции input() совместно с инструкцией «try: … except»

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Ведите любое число с десятичной точкой:124ю543
124ю543
Это не число! Введен недопустимый символ!
```

3.8.3.Функция print() для вывода данных

Обычно требуется, чтобы программа обрабатывала набор входных данных, которые поступают из внешних источников. В свою очередь вывод результатов работы программы возможен в файл, в базу данных или на принтер. Однако нередко информацию просто выводят на экран монитора, в Python это делается с использованием функции ptint(). Мы уже многократно сталкивались с данной функцией, теперь более подробно рассмотрим ее возможности.

Эта функция выводит все, что ей передано в аргументах при обращении: строки, числа, списки, кортежи, значения из других функций и методов классов. В качестве аргументов могут быть и выражения. Кроме информации, которую нужно вывести на печать, данная функция имеет и ряд встроенных параметров, например:

- sep – указывает на отличный от пробела разделитель строк;
- end - указывает на действия, которые нужно выполнить после вывода строки (по умолчанию происходит переход на новую строку).

Эти параметры, как правило, указываются после объектов, которые нужно отобразить (рис. 3.48).



Рис. 3.48. Значения параметров функции, которые можно задать при ее вызове

Рассмотрим несколько простых примеров использования функции print() для различных типов данных, и с использованием вышеуказанных параметров (листинг 3.47).

Листинг 3.47. Использование функции функции print() для различных типов данных

```
# Листинг 3.47
print('Строка')
print('Число -', 135.8)
print('Список -', [111, 'Иванов А.В.', 123.45])
print('Картеж -', [222, 'Петров П.П', 333.22])
print('Функци -', int(555.33), str('999'))
print(1, 2, 3, sep=" !! ")
print(111, 111, 111, end="\n\n")  # Задает дополнительный пропуск одной
строки
print(222, 222, 222, end="")      # Отменить перевод на новую строку
print(' Один ', ' Два ', ' Три ')
print()  # Печать пустой строки
print('Эта строка после пустой')
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.47. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.49).

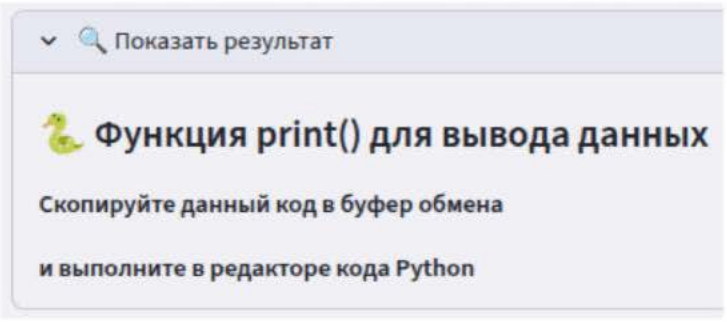


Рис. 3.49. Использование функции print() для различных типов данных

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Строка
Число - 135.8
Список - [111, 'Иванов А.В.', 123.45]
Картеж - [222, 'Петров П.П', 333.22]
Функци - 555 999
1 !! 2 !! 3
111 111 111

222 222 222 Один   Два   Три

Эта строка после пустой
```

Примечание.
Переход на новую строку обозначается символом '\n'. Однако если присвоить это значение параметру end (например, end='\n'), то никаких изменений в работе функции print() не произойдет, так как соответствующий параметр функции уже имеет это значение по умолчанию. Для того чтобы после печати текущей строки была вставлена пустая строка, данный параметр должен иметь значение end='\n\n'. Можно задать печать пустой строки вызовом данной функции без параметров – print().

В функцию print() можно передать так называемые форматированные строки, хотя по смыслу их правильнее называть строки-шаблоны. Никакого отношения к самой функции print() они не имеют. Когда такая строка находится в скобках функции print(), интерпретатор сначала преобразует ее к обычной строке (согласно заданному в ней формату), после чего передает результат в print(). О форматировании строк было достаточно подробно рассказано в разделе «Строковые данные в Python», подраздел «Использование оператора «%» и f-строки для форматирования строк».

3.8.4. Вопросы для самоконтроля к разделу 3.8



11. 1. Что означает режим ‘r’ при открытии файла функцией open():

- ·4 Открыть только на чтение;
- ·5 Открыть только на запись;

·6 Открыть на чтение и на запись;
·7 Открыть в текстовом режиме.

12. 2. Что означает режим ‘w’ при открытии файла функцией open():

- Открыть только на запись;
- Открыть только на чтение;
- Открыть на чтение и на запись;
- Открыть в текстовом режиме.

13. 3. С какой кодировкой символов работает Python по умолчанию:

5. UTF-8;
6. UTF-16;
7. cp1251;
8. ASCII.

14. 4. Какую роль играет параметр end в функции print():

- указывает на действия, которые нужно выполнить после вывода строки;
- указывает на прекращение вывода части строки;
- отменяет печать текущей строки;
- отменяет дальнейшую печать всех строк.

3.9.Задание путей к файлам в Python

Для решения задач, связанных с редактированием или чтением файлов, интерпретатору необходимо сообщить имя нужного файла, а также адрес (путь), по которому этот файл располагается. Существуют разные способы указания пути к файлу: от самого простого, до самого правильного. Давайте выясним, чем эти варианты отличаются и почему простой вариант не годится на роль лучшего!

Самый простой вариант задания пути выглядит как последовательность директорий, в которых находится файл, с именем самого файла, разделенные знаками слеша, например:

```
path = "Files/my_file.txt"  - в Linux;
path = "Files\my_file.txt"  - в Windows.
```

Здесь вместо «Files» нужно написать всю цепочку названий ваших директорий, а «my_file.txt» и имя файла.

Это достаточно простой вариант и вполне рабочий, однако он имеет один существенный недостаток. Проблема заключается в том, что заданные таким способом пути адаптированы только к одной из операционных систем: либо к Linux, либо к Windows, так как в Windows используются обратные слеши «\», а в Linux — обычные «/». Из-за этого скрипт, показывавший отличные результаты в Windows, начнет жаловаться на отсутствие файлов по прописанному пути в Linux, и наоборот. А с абсолютным путем еще сложней: никакого диска «C:» в Linux нет. Скрипт будет выдавать ошибку, и отказываться работать! Как же быть, как правильно указать путь к файлу вне зависимости от особенностей операционных систем?

Для Python было найдено универсальное решение - в его арсенале, начиная с версии 3.4, появился модуль pathlib, который позволяет автоматизировать построение путей к файлам, стоит только импортировать из него класс Path с использованием следующей инструкции:

```
from pathlib import Path
```

После того, как класс импортирован, программист получает полный контроль над слешами! Теперь решение об использовании прямых или обратных слешей в разных операционных системах ложится на плечи Path. Используя Path, можно задать относительный путь, который будет корректно работать в разных операционных системах.

Класс Path имеет методы:

- **cwd()** — возвращает путь к рабочей директории;
- **home()** — возвращает путь к домашней директории;
- **parent** — возвращает путь к родительской директории.

Рассмотрим пример расположения файлов, как на рис. 3.50.

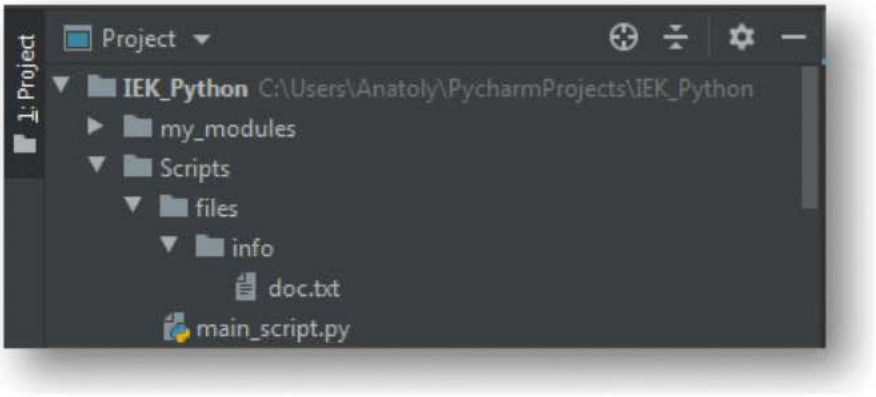


Рис. 3.50. Пример расположения файлов в директориях

Пример программного модуля, в котором продемонстрировано использование класса Path для манипуляции путями к файлу doc.txt, приведен ниже в листинге 3.48.

Листинг 3.48. Использование класса Path для манипуляции путями к файлу

```
# Листинг 3.48
# Пример использования класса Path модуля pathlib
from pathlib import Path

print('Получаем путь к домашней директории')
home_dir = Path.home()
print(home_dir)
print('-'*30)

print('Получаем путь к рабочей директории')
script_dir = Path.cwd()
print(script_dir)
print('-'*30)

print('Получаем родительскую папку рабочей директории')
parent_dir = script_dir.parent
print(parent_dir)
print('-'*30)

print('Получаем родительскую папку родителя рабочей директории')
parent_dir = script_dir.parent.parent
print(parent_dir)
print('-'*30)

print('Получаем путь к файлу из рабочей директории')
file_path_1 = Path("files", "info", "docs.txt")
print(file_path_1)
print('-'*30)
```

```
print('Получаем полный путь к файлу')
file_path_2 = Path(script_dir, 'files', 'info', 'docs.txt')
print(file_path_2)
print('-'*30)

print('Получаем тип объекта path')
print('Исходный объект path->', type(file_path_2))
file_path_2 = str(file_path_2)
print('Строковый объект path->', type(file_path_2))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 3» выбрать опцию «Листинги главы 3», и в выпадающем списке боковой панели выбрать раздел «Листинг 3.48. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 3.51).



Рис. 3.51. Использование класса Path для манипуляции путями к файлу

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Получаем путь к домашней директории
C:\Users\Anatoly
------------------------------
Получаем путь к рабочей директории
C:\Users\Anatoly\PycharmProjects\IEK_Python\Scripts
------------------------------
Получаем родительскую папку рабочей директории
C:\Users\Anatoly\PycharmProjects\IEK_Python
------------------------------
Получаем родительскую папку родителя рабочей директории
C:\Users\Anatoly\PycharmProjects
------------------------------
Получаем путь к файлу из рабочей директории
files\info\docs.txt
------------------------------
Получаем полный путь к файлу
C:\Users\Anatoly\PycharmProjects\IEK_Python\Scripts\files\info\docs.txt
------------------------------
```

```
Получаем тип объекта path
Исходный объект path-> <class 'pathlib.WindowsPath'>
Строковый объект path-> <class 'str'>
```

Здесь с начала с использованием метода Path.home() был получен путь к домашней директории, затем с использованием метода Path.cwd() – путь к рабочей директории (из которой был запущен скрипт). Затем с использованием метода parent показа возможность движения от рабочей директории к родительским. Наконец, с использованием следующих двух инструкций был получен путь к файлу doc.txt из рабочей директории и полный путь к данному файлу. В последних строках был распечатан тип объекта, который возвращают методы класса Path. Как видно из распечатки результатов – был получен объект, ориентированный на использование путей под операционную систему Windows. В таком виде не всегда удобно использовать полученный путь и его целесообразно преобразовывать к объекту строкового типа.

Примечание.

Модуль main_script.py будет корректно работать только при структуре каталогов данного примера. Вы можете импортировать его и использовать в качестве шаблона для манипулирования путями в рамках структуры каталогов вашего проекта.

Глава 4. Функции в Python



В Python, как и в других языках программирования, можно разделить большую программу на небольшие фрагменты, называемые функциями. Функции позволяют создавать эффективный программный код, избегая его повторения. В Python есть два типа функций: встроенные, и определяемые пользователем. К встроенным относятся те, которые предварительно «встроены» в Python, например print(), int(), str(). К пользовательским функциям относятся те функции, которые программист определяет самостоятельно для выполнения конкретных задач. Из материалов данной главы вы узнаете:

10. общие сведения о функциях;
11. что такое параметры функций и аргументы;
12. некоторые особенности, которые нужно учитывать при создании функций и обращениях к ним;
13. как создать функции, которые могут принимать переменное количество аргументов;
14. как в Python ограничивается область видимости переменных;
15. как создать анонимные функции (lambda функции);
16. что такое рекурсивные функции и как их использовать;
17. как получить доступ к атрибутам функций и выполнить их документирование;
18. что такое аннотированные функции;
19. что такое функции декораторы и для чего они нужны.

В конце главы будут даны рекомендации, которым нужно следовать при проектировании и реализации пользовательских функций.

4.1.Общие сведения о функциях

Функция (от англ. function) – это блок программного кода, который определяется один раз и далее может быть использован многократно. В функциях, как правило, программируются действия, которые необходимо будет выполнять в различных частях программы. Функция кодируется один раз, но сам по себе этот код не выполняется, он как бы спит и ждет, когда его вызовут для исполнения.

Функция в Python является основой при написании программ. С чем можно сравнить функцию? Напрашивается аналогия с «черным ящиком», когда мы знаем, что поступает на вход и что ожидаем получить на выходе, а содержание «черного ящика» от нас скрыто, да и не интересно. Примером черного ящика может являться стиральная машина. На вход поступает грязное белье, на выходе получаем чистое белье. Нас в этом случае совершенно не интересует устройство стиральной машины и принцип ее работы, важен результат (рис. 4.1).

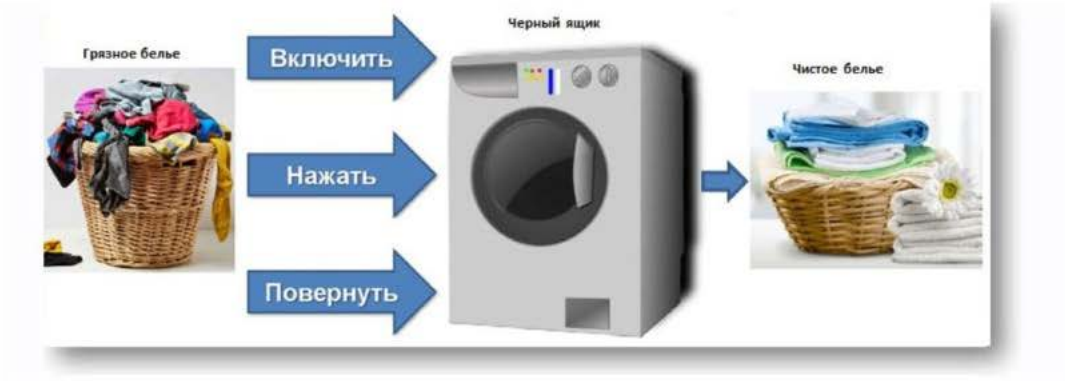


Рис. 4.1. «Черный ящик на примере стиральной машины

Точно так же и функция, она содержит некий сложный алгоритм, который от нас скрыт, но мы точно знаем, что ей нужно передать, что функция выдаст на выходе.

Различают системные функции и функции пользователя. К системным функциям относятся те, которые уже имеются в структуре языка программирования. В вышеприведенных примерах мы достаточно часто использовали следующие системные функции:

- print() - вывода данных на экран (на печать);
- len() - определение количества элементов в объекте;
- min(s) - минимальное значение элемента из итерируемого объекта;
- max(s) - наибольшее значение элемента итерируемого объекта;
- sum(s) - суммирование элементов последовательности и т.п.

Однако системные функции не могут обеспечить выполнение всех действий, которые могут потребоваться в конкретной программе. Поэтому программисты имеют возможность написать собственные пользовательские функции и затем многократно обращаться к ним из любой точки программного кода.

Каждая пользовательская функция решает какую-то одну конкретную задачу, что дает возможность программистам разбивать сложную программу на достаточно простые и легко читаемые части. Пользовательские функции заменяют объемные копии блоков одного и того же программного кода. Как следствие, при необходимости внесения изменений в код, делать это придется уже не во многих местах программы, а только в одном месте – в теле самой функции.

Таким образом, функции можно смело назвать основными программными структурами языка Python, которые позволяют существенно уменьшить время и трудозатраты на разработку приложений, а также их дальнейшее сопровождение. Это достигается за счет следующих преимуществ функций:

- ➲ дают возможность разбить сложную программу на небольшие и легко управляемые фрагменты, каждый из которых может разрабатываться отдельно;
- ➲ обеспечивают многократное использование программного кода;
- ➲ уменьшают его избыточность;
- ➲ повышают читабельность исходного кода и упрощают его редактирование.

Ниже приведена структура программного кода, который создает пользовательскую функцию.

```
def my_fun(a, b):
    c = a + b
    return c
```

Задается функция с помощью ключевого слова «def». Далее следует название функции (в вышеприведенном примере – «my_fyn»), затем следуют скобки и двоеточие. В скобках через запятую можно перечислить параметры, которые будет принимать функция (в нашем случае это «a» и «b»). Функция может иметь, а может и не иметь параметры. Если принимаемые параметры отсутствуют, то скобки все равно необходимо сохранить. В следующих строках с **обязательным отступом** следуют строки тела функции - программного кода с действиями, которые будут выполнены при вызове функции. Если функция возвращает результат вычислений, то последней ее строкой будет инструкция «return» с указанием возвращаемого результата (в нашем случае это значение переменной «c»). Если функция ничего не возвращает, то инструкция «return» может отсутствовать и интерпретатор не выдаст ошибки. Но такой стиль считается «плохим тоном». В простых случаях интерпретатор по наличию отступов сможет отделить тело функции от тела программы, но в сложных программных структурах такая программа может выдать непредсказуемые результаты. В связи с этим тело каждой функции рекомендуется завершать инструкцией return, что делает программный код более понятным и лучше читаемым.

Программный код, который находится в теле функции, не будет выполняться до тех пор, пока одна из инструкций не вызовет эту функцию. Для вызова функции нужно указать ее имя и в скобках перечислить передаваемые в нее аргументы. Для приведенного выше примера вызвать функцию можно следующей инструкцией: d = my_fun(10, 20). В этом случае числа 10 и 20 будут переданы в функцию, эти значения будут присвоены параметрам a=10, b=20, сумма этих чисел – 30 будет присвоена переменной «c». Последняя строка «return c» вернет это значение в точку вызова функции, то есть результатом работы функции будет d=30.

Пример программного кода, в котором реализована работа описанной выше функции, приведен в листинге 4.1.

Листинг 4.1. Пример простейшей функции

```
# Листинг 4.1
# определяем и объявляем функцию
def my_fun(a, b):
    c = a + b
    return c

d = my_fun(10, 20)  # вызов функции
print('d =', d)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.1. Результаты работы модуля можно увидеть, открыв контейнер «Показать

результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.2).

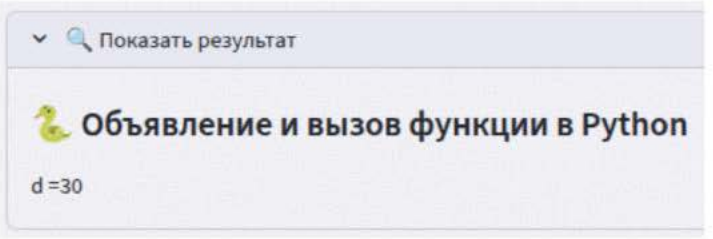


Рис. 4.2. Пример простейшей функции

Модифицируем данный программный код - сделаем несколько обращений к созданной функции, каждый раз задавая новые аргументы при вызове (листинг 4.2).

Листинг 4.2. Пример многократного обращения к функции

```
# Листинг 4.2
# определяем и объявляем функцию
def my_fun(a, b):
    c = a + b
    return c


a = my_fun(1, 2)  # вызов функции
print('a =', a)

a = 100
b = 200
k = my_fun(a, b)  # вызов функции
print('k =', k)

m = 50
n = 60
d = my_fun(m, n)  # вызов функции
print('d =', d)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.2. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.3).

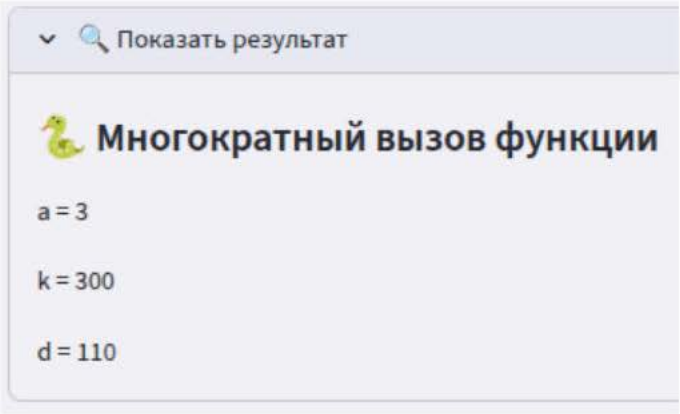


Рис. 4.3. Пример многократного обращения к функции

Как видно из данного примера, в теле функции присутствуют переменные «a», «b» и в самой программе есть переменные «a», «b», при этом значения этих переменных в программе никак не влияет на значения тех же переменных в функции. Дело в том, что интерпретатор Python воспринимает их как разные переменные, благодаря чему все инструкции, которые находятся в теле функции, никак не влияет на работу основной программы.

4.2. Аргументы и параметры функций

При работе с функциями нужно четко различать такие понятия, как аргументы и параметры функции (рис. 4.4).

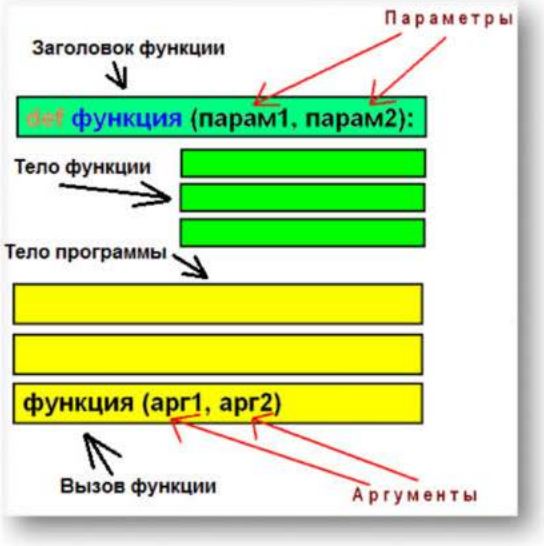


Рис. 4.4. Параметры функции и аргументы

Параметры - это локальные переменные, которым присваиваются значения в момент вызова функции. Параметры объявляются в заголовке функции и отделяются друг от друга запятыми.

Аргументы – это конкретные значения, которые передаются в функцию при ее вызове. Аргументы находятся в инструкции, которая вызывает функцию, и отделяются друг от друга запятыми.

В заголовке функции параметры перечисляются через запятую. При этом разрешается присваивать параметрам значения, которые в теле функции будут использоваться, как значения по умолчанию. Это значение будет использовано в тех случаях, когда при вызове функции ей будет передано меньшее количество аргументов.

Тело функции представляет собой обычный программный код, который будет выполняться каждый раз при вызове функции. Вызов функции осуществляется при помощи имени функции и круглых скобок с передаваемыми функции значениями аргументов, которые также должны перечисляться через запятую. Если аргументы передаются без имен, их называют позиционными, т.к. интерпретатор сопоставляет их с порядком следования параметров в заголовке функции.

Кроме того, имеется возможность передавать функциям аргументы, явно указывая их имена и значения. Такие аргументы называются именованными и могут передаваться функции в любом порядке, но только после позиционных аргументов, если такие присутствуют в вызове функции. Необязательная инструкция «return» возвращает результат вызывающей программе в место вызова функции.

Пример программного кода, в котором используются разные варианты передачи аргументов в функцию, приведен в листинге 4.3.

Листинг 4.3. Разные варианты передачи аргументов в функцию

```
# Листинг 4.3
# Объявление функции
def my_func(x=20, y=10):
    rez = x - y
    return rez


res_1 = my_func()  # Аргументы не заданы
print('Аргументы не заданы: res_1 =', res_1)

res_2 = my_func(40)  # В функцию передан один аргумент
print('Передан 1 аргумент (20): res_2 =', res_2)

res_3 = my_func(200, 100)  # В функцию передано два аргумента
print('Передано 2 аргумента (200, 100): res_3 =', res_3)

res_4 = my_func(x=100, y=500)
print('Переданы именованные аргументы (x=100, y=500): res_4 =', res_4)

# Явно указываем имена аргументов, поэтому порядок не важен
```

```
res_5 = my_func(y=500, x=100)
print('Переданы именованные аргументы (y=500, x=100): res_5 =', res_5)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.3. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.5).



Рис. 4.5. Разные варианты передачи аргументов в функцию

В этом примере функция my_fun имеет 2 параметра, которым присвоено значение по умолчанию (x=20, y=10). К этой функции было 5 обращений и получены следующие результаты:

9. rez_1 = 10 – при обращении к функции аргументы не заданы - my_func(), в выражении тела функции использовались значения параметров по умолчанию x=20, y=10, получен результат rez = 20 – 10 = 10;
10. rez_2 = 30 – передан 1 аргумент - my_func(40), в выражении параметр «x» получил значение x=40, а значение параметра «y» не изменилось y=10, в итоге получился результат z=40-10=30;
11. rez_3 = 100 – передано 2 аргумента - my_func(200, 100), в выражении параметр «x» получил значение x=200, и параметра «y» тоже изменил значение y=100, в итоге получился результат z=200-100=100;
12. rez_4 = -400 – передано 2 именованных аргумента my_func(x=100, y=500), в выражении параметр «x» получил значение x=100, параметра «y» значение y=500, в итоге получился результат z=100-500=-100;

13. rez_5 = -400 – передано 2 именованных аргумента my_func(y=500, x=100), в выражении параметр «x» получил значение x=100, параметра «y» значение y=500, в итоге получился результат z=100-500=-100.

Обратите внимание на два последних варианта вызова функции. В них использовались именованные параметры. Не смотря на то, что в процедуре вызова они поменялись местами, результат не изменился. Принцип передачи в функцию аргументов и возврата результатов схематично представлен на рис. 4.6.

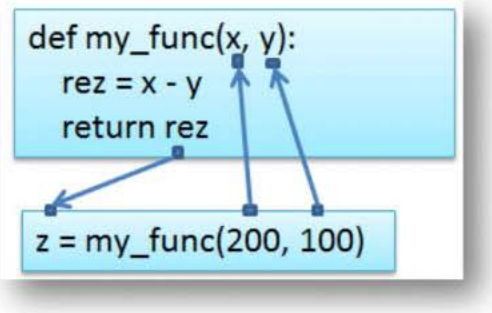


Рис. 4.6. Принцип передачи в функцию аргументов и возврата результатов

Предоставленной здесь информации нам вполне хватит, чтобы писать простейшие функции. Однако далее более подробно рассмотрим некоторые детали инструкций, которые применяются при использовании функций.

4.3. Некоторые особенности при создании и вызове функций

Заголовок инструкции, которая создает функцию, всегда начинается с ключевого слова «def». Функции имеют параметры (не путать их с аргументами). Можно определять функции и без параметров, но круглые скобки при этом все равно нужно указывать. Завершается строка заголовка **обязательным** двоеточием. Далее следует тело функции, представляющее собой обычный блок программного кода, который будет выполняться каждый раз при вызове функции. Если тело функции состоит лишь из одной простой инструкции (одной строки программного кода), то такую инструкцию можно записывать вслед за двоеточием в строке заголовка. В других случаях инструкции тела функции записываются со следующей строки после заголовка функции с отступами в соответствии с общепринятыми правилами (отступ имеет четыре пробела).

Помимо обычных инструкций в любом месте тела функции может содержаться необязательная инструкция «return», которая предназначена для выхода из функции и возврата результата вызывающей программе в место вызова функции. В случае отсутствия инструкции «return» функция будет завершать свою работу по достижении последней строки тела функции. Однако технически функция все равно будет неявно возвращать результат в виде объекта «None», который обычно просто игнорируется. Однако рекомендуется всегда завершать функцию инструкцией «return», в этом случае код программы более понятный и лучше читается.

Для завершения работы функции можно также использовать специальную инструкцию «yield», которая превращает обычную функцию в функцию-генератор. Однако здесь мы

заострять внимание на ней не будем, поскольку данный вопрос будет подробно рассмотрен в самостоятельном разделе.

После того, как функция будет определена, ее можно начинать использовать, многократно вызывая практически в любое время и в любом месте тела программы. Следует иметь в виду, что функция становится доступной для использования только после того, как она была определена. Если попытаться вызвать функцию до ее определения в коде программы, то интерпретатор выдаст ошибку.

Как было сказано выше, в случаях, когда функция определяется с аргументами, их имена перечисляются через запятую в круглых скобках после имени функции. Далее, в моменты вызовов функции, все определенные в ней параметры автоматически становятся ее локальными переменными, которым присваиваются переданные в функцию значения. При этом по умолчанию сопоставление аргументов производится слева направо в соответствии с их позициями, а самих аргументов допускается передавать ровно столько, сколько имен было указано в заголовке функции при ее определении.

Если в теле функции всего одна инструкция, то допускается оформление такой функции в одной строке. Простое выражение, которое вычисляется в функции, может располагаться следом за инструкцией «return». Параметры функции принимают тот тип данных, которые переданы в аргументах. Пример такой функции приведен в листинге 4.4.

Листинг 4.4. Однострочная функция

```
# Листинг 4.4

# Определяем функцию
def func_1(a, b, c): return a + b + c

x = 1
y = 2
z = 3
rez1 = func_1(x, y, z)
print('Аргументы числа, rez1 = ', rez1)

x = 'Иванов '
y = 'Семен '
z = 'Петрович'
rez2 = func_1(x, y, z)
print(print('Аргументы символы, rez2 = ', rez2))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.4. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.7).

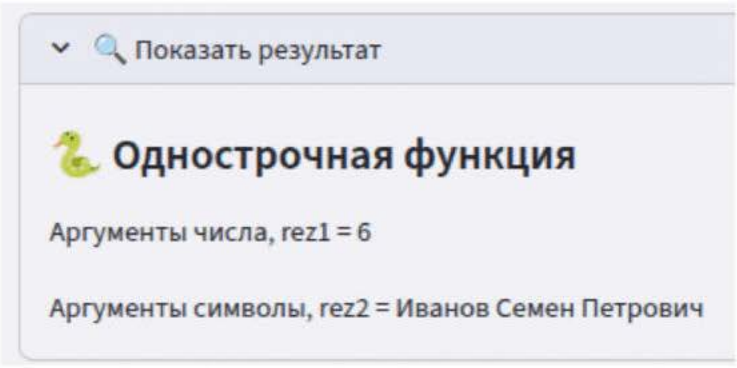


Рис. 4.7. Однострочная функция

В этом примере вся функция записана одной строкой. Инструкции в теле функции отсутствуют, а действие сложения параметров находятся после инструкции «return». В этом случае инструкции «return» последовательно выполняет две операции: производит соложение параметров, возвращает результат в место вызова. При первом обращении к функции ей было передано три аргумента, имеющих тип – целое число. Параметры функции то же стали числами и была получена их сумма - rez1=6. При втором обращении к функции ей было передано три аргумента, имеющих тип – символьная строка. Параметры функции то же стали символьными строками и была получена сумма строк - rez2=Иванов Семен Петрович.

Количества передаваемых в функцию аргументов должно соответствовать количеству параметров функции. Если такого совпадения нет, то это приведет к ошибке при работе программы. Однако в подобных ситуациях можно исключить появление ошибок, используя один из доступных в Python специальных режимов сопоставления аргументов с параметрами функции, которые мы сейчас и разберем. Рассмотрим несколько возможных вариантов позиционирования аргументов при обращениях к функциям.

Вариант 1. Определение функции без поименованных позиционных параметров:

```
def func(par_1, par_2, ...).
```

Во время вызова такая функция ожидает получить строго определенное количество аргументов, сопоставление которых будет происходить либо по их позициям, либо по именам. При этом в случае использования в вызове функции именованных аргументов их разрешается указывать в любом порядке, но только после того, как будут переданы все имеющиеся позиционные аргументы, которые в любом случае будут сопоставляться слева направо в соответствии с их позициями.

Пример программы, показывающей варианты передачи аргументов при таком заголовке функции, приведен в листинге 4.5.

Листинг 4.5. Определение функции без поименованных позиционных параметров

```
# Листинг 4.5
print('Функция с тремя позиционными параметрами def func(x, y, z):')
def func(x, y, z):
    rez = x + y - z
    return rez
```

```
print('-'*60)
print('Вызов функции с разными вариантами позиционния аргументов')
print('func(8, 2, 5) Результат ->', func(8, 2, 5))
print('func(x=8, y=2, z=5) Результат ->', func(x=8, y=2, z=5))
print('func(y=2, x=8, z=5) Результат ->', func(y=2, x=8, z=5))
print('func(z=5, x=8, y=2) Результат ->', func(z=5, x=8, y=2))
print('func(8, y=2, z=5) Результат ->', func(8, y=2, z=5))
print('func(8, z=5, y=2) Результат ->', func(8, z=5, y=2))
print('func(8, 2, z=5) Результат ->', func(8, 2, z=5))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.5. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.8).

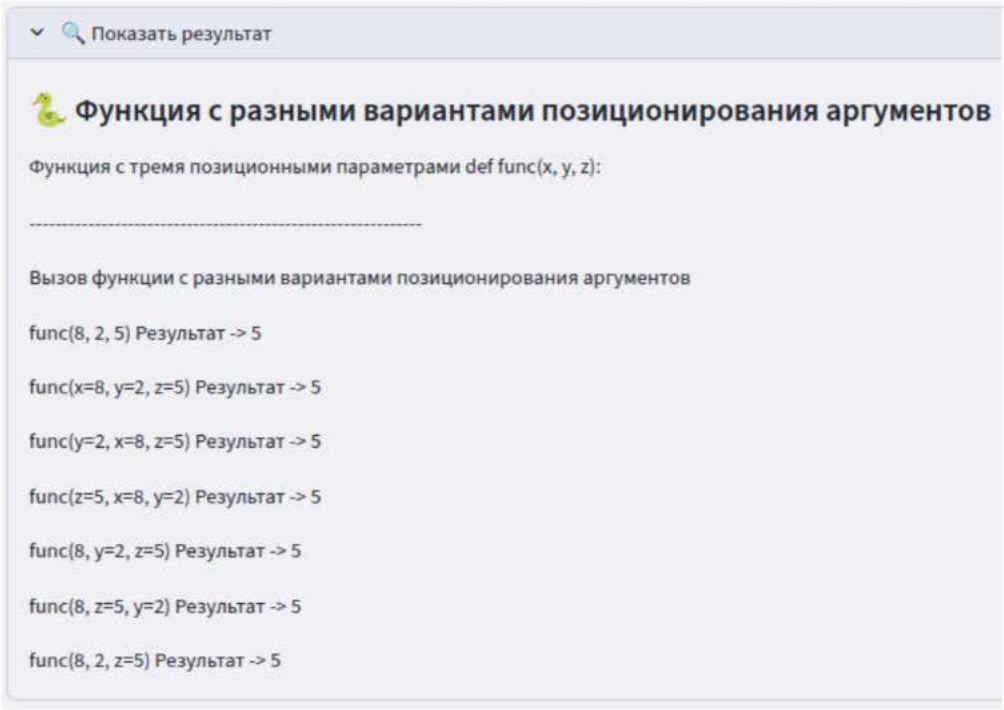


Рис. 4.8. Определение функции без поименованных позиционных параметров

Как видно из данного примера, даже при разной последовательности расположения аргументов функция вернула один и тот же результат.

Вариант 2. Определение функции, у которой есть поименованные параметры со значением по умолчанию:

def func(par_1, par_2, ..., par_3=val_1, par_4=val_2, ...).

В таких случаях поименованные параметры нужно перечислять после обычных позиционных параметров (если они имеются). Наличие поименованных параметров позволяет вызывать функцию без передачи им значений аргументов, в этом случае функция будет просто использовать значения параметров по умолчанию (val_1, val_2). Во всем остальном для таких функций справедливы правила стандартного режима передачи аргументов.

Пример программы, показывающей варианты использования функций с поименованными параметрами, приведен в листинге 4.6.

Листинг 4.6. Определение функции с поименованными параметрами со значением по умолчанию

```
# Листинг 4.6
print('Функция с одним позиционными параметром def func(x, y=1):')
def func(x, y=1):
    rez = x / y
    return rez

print('-'*60)
print('Вызов функции с разными вариантами передачи аргументов')
print('func(8, 2) Результат ->', func(8, 2))
print('func(x=8, y=2) Результат ->', func(x=8, y=2))
print('func(8, y=2) Результат ->', func(8, y=2))
print('func(2) Результат ->', func(2))
print('func(x=2) Результат ->', func(x=2))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.6. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.9).

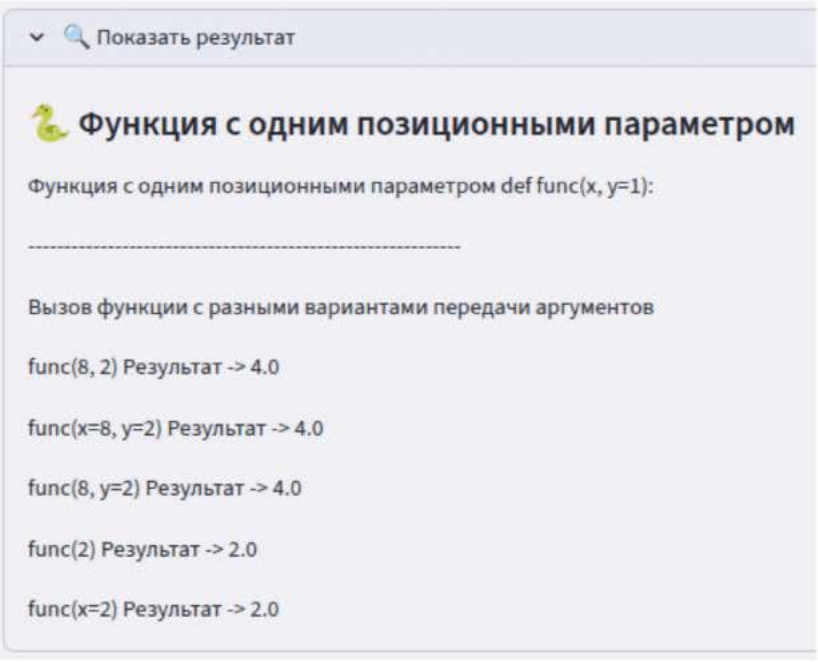


Рис. 4.9. Определение функции с поименованными параметрами со значением по умолчанию

Из данного примера видно, что в зависимости от того, в какой последовательности и в каком виде аргументы были переданы в функцию, она вернула разные результаты.

4.4. Функции с приемом переменного количества аргументов

Мы уже знаем, что существует два вида параметров функций: позиционные и поименованные. При обращении к функциям им необходимо передать соответствующее количество позиционных и поименованных аргументов. Однако программисты часто сталкиваются с ситуациями, когда в функцию нужно передать аргументы, количество которых заранее неизвестно, так как они формируются в процессе работы программного модуля. Вот тут на помощь придет волшебный оператор звездочка «*». Оператор звездочка чаще всего ассоциируется у людей с операцией умножения, но в Python он имеет и другой смысл. Этот оператор используется в функциях и позволяет «распаковывать» объекты, внутри которых хранятся некие элементы.

В Python передать в функцию переменное (заранее неизвестное) количество аргументов можно двумя способами:

- *args - для неименованных аргументов;
- **kwargs - для именованных аргументов.

Здесь «*args» — это сокращение от анг. «arguments» (аргументы), а **kwargs — сокращение от анг. «keyword arguments» (именованные аргументы). Каждая из этих конструкций используется для распаковки аргументов соответствующего типа, позволяя вызывать функции со списком аргументов переменной длины.

Если функция должна принять разное количество неименованных аргументов, то перед именем ее принимающего параметра ставится звездочка - «*», например «*args». В этом случае все принятые функцией данные будут преобразованы в кортеж.

Если функция должна принять разное количество именованных аргументов, то перед именем ее принимающего параметра ставится две звездочки - «**», например «**kwargs». В этом случае все принятые функцией данные будут преобразованы в словарь.

Чтобы код программы был более понятным и удобно читаемым, рекомендуется в качестве имен таких параметров использовать переменные «*args» и «**kwargs». Тогда в инструкциях тела функции эти последовательности данных получат идентификаторы «args» и «kwargs». Но это вовсе не обязательно, Например, если в заголовке функции обозначить такие параметры «*num_1» и «**num_2», то в инструкциях тела функции эти последовательности данных получат идентификаторы «num_1» и «num_2».

При использовании такой конструкции в заголовке функции сначала перечисляются позиционные параметры, и только после них параметры с переменным количеством аргументов. При обращении к таким функциям все аргументы, которые не вошли в список позиционных параметров, будут объединяться в кортеж или в словарь.

Рассмотрим на конкретном примере, в каких случаях возникает потребность использования инструкции «*args» (листинг 4.7).

Листинг 4.7. Определение функции с использованием инструкции «*args»

```
# Листинг 4.7
print('Фукция с фиксированным количеством параметров - def my_fun_1(x,
y):')
def my_fun_1(x, y):
    print('Тип переменной y ->', type(y))
    rez = x + y
    return rez

rez_1 = my_fun_1(1, 2)
print('Результат my_fun_1(1, 2) ->', rez_1)
print()

print('Фукция с одним из параметров переменной длины - def my_fun_1(x,
*y):')
def my_fun_2(x, *y):
    print('Тип переменной y ->', type(y))
    rez = x + sum(y)
    return rez

rez_2 = my_fun_2(1, 2, 3)
print('Результат my_fun_2(1, 2, 3) ->', rez_2)
print()

print('Фукция с параметром переменной длины - my_fun_3(*args):')
```

```
def my_fun_3(*args):
    rez = sum(args)
    return rez

print('Результат my_fun_3() ->', my_fun_3())
print('Результат my_fun_3(1) ->', my_fun_3(1))
print('Результат my_fun_3(1, 2) ->', my_fun_3(1, 2))
print('Результат my_fun_3(1, 2, 3) ->', my_fun_3(1, 2, 3))
print('Результат my_fun_3(1, 2, 3, 4) ->', my_fun_3(1, 2, 3, 4))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.7. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.10).

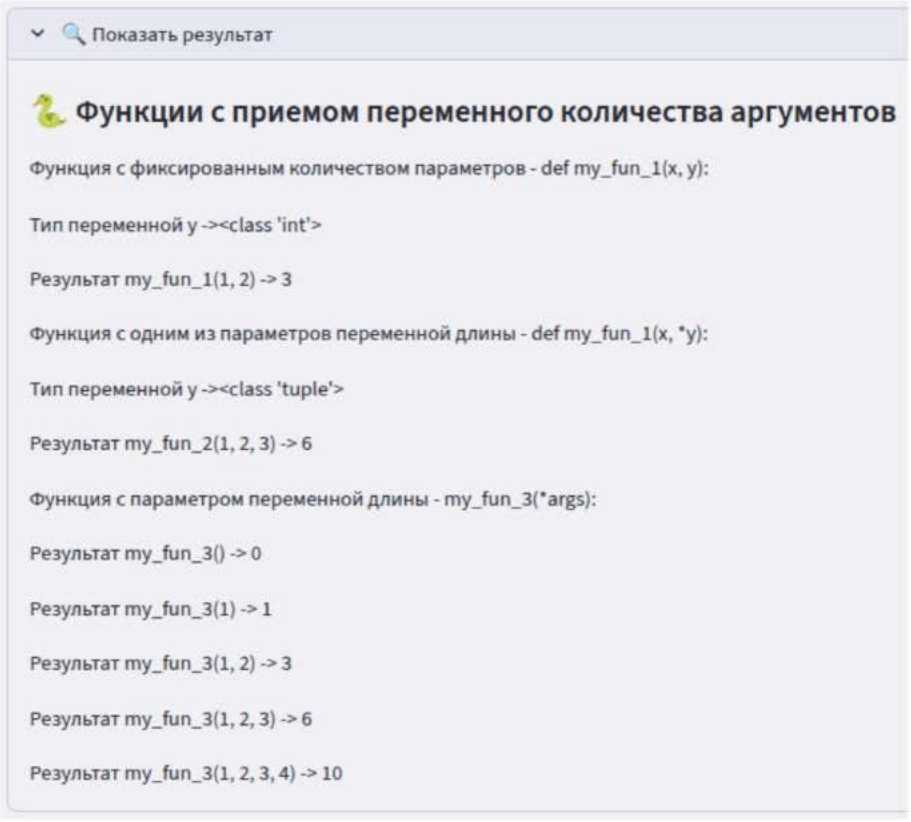


Рис. 4.10. Определение функции с использованием инструкции «*args»

В данном примере на первом шаге была создана функция «def my_fun_1(x, y):». У этой функции два позиционных параметра «x» и «y». Такая функция может принять два (и только два) аргумента. В примере, приведенном выше, при вызове функции my_fun_1 ей было передано два числа (1 и 2), и получена сумма этих чисел – 3. Если мы попытаемся вызвать данную функцию с одним аргументом, или с тремя и более аргументами, то интерпретатор выдаст ошибку. Обратите внимание, что аргументы, переданные в функцию, являются числами. Контрольный вывод в теле функции типа для переменной «y» показал, что она является целыми числом.

На втором шаге была создана функция «def my_fun_2(x, *y):». У этой функции те же два параметра «x» и «*y», но звездочка указывает, что второй параметр будет распаковывать принятые данные и преобразовывать их в кортеж. Соответственно в теле нашей функции элементы кортежа нужно суммировать встроенной в Python функцией - sum(). В примере, приведенном выше, при вызове функции my_fun_2 ей было передано три числа (1, 2, 3), и получена сумма этих чисел – 6. Функция приняла единицу как целое число, в вот двойку и тройку преобразовал в кортеж, после чего получила их полную сумму. Контрольный вывод в теле функции типа для переменной «y» показал, что она действительно является кортежем. Теперь в данную функцию можно передавать сколько угодно аргументов. При этом первый аргумент будет преобразован в число, а все последующие в кортеж.

На третьем шаге была создана функция «def my_fun_3(*args):». У этой функции всего один параметр «*args», и звездочка указывает, что этот параметр будет преобразовывать все принятые данные в кортеж. Соответственно в теле нашей функции элементы кортежа нужно суммировать встроенной в Python функцией - sum(). Теперь количество аргументов, которые будут приняты функцией, ничем не ограничено. В примере, приведенном выше, при каждом вызове функции my_fun_3 ей передавалось разное количество аргументов: ни одного числа (или пусто), одно, два, три, четыре числа. При этом функция принимала все данные, преобразовывала их в кортеж и находила сумму этих чисел.

Рассмотрим несколько возможных вариантов использования функций в тех случаях, когда при каждом обращении к ним количество передаваемых аргументов разное, или заранее неизвестно.

Вариант 1. Определение функции, которая принимает позиционные аргументы и разное количество не именованных аргументов:

```
def func(par_1, ..., par_n, *args)
```

Пример программы использования функции с позиционными параметрами, один из которых преобразует принятые данные в кортеж, приведен в листинге 4.8.

Листинг 4.8. Определение функции с позиционными аргументами и разным количеством не именованных аргументов

```
# Листинг 4.8
print('Функция с параметром, принимающим кортеж def func(x, *tup):')


def func(x, *tup):
    rez = x + sum(tup)
    return rez
```

```
print('-'*60)
print('Вызов функции с разными вариантами передачи аргументов')
print('func(5) Результат ->', func(5))
print('func(5, 3) Результат ->', func(5, 10))
print('func(5, 10, 15, 20, 30) Результат ->', func(5, 10, 15, 20, 30))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.8. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.11).

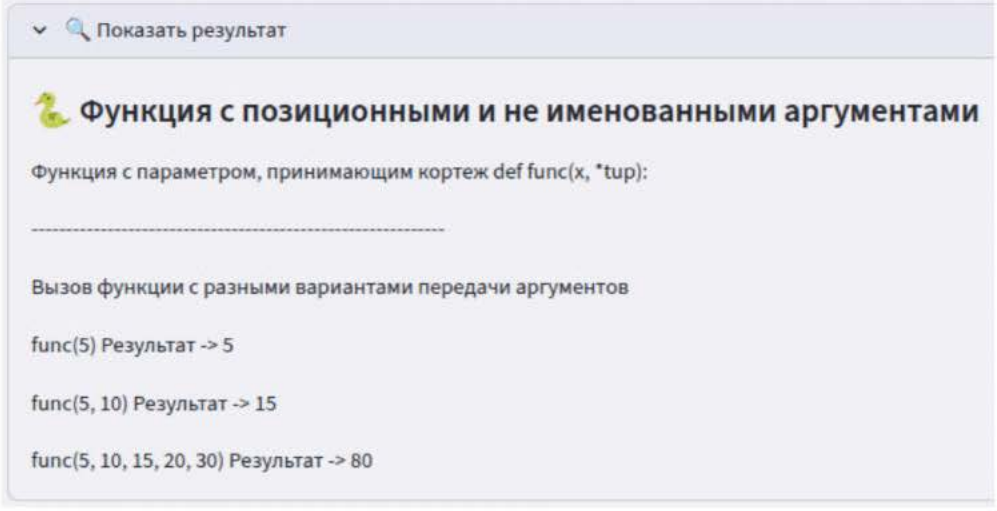


Рис. 4.11. Функция с позиционными аргументами и разным количеством не именованных аргументов

Как видно из данного примера, при обращении к такой функции количество аргументов может быть достаточно большим. Все аргументы, которые не вошли в перечень позиционных параметров, будут объединены в один параметр функции в виде кортежа.

Вариант 2. Определение функции, у которой есть позиционные параметры и параметр - кортеж, который находится в середине списка параметров:

```
def func(par_1, ..., par_n, *par, name_ par_1, ..., name_ par_m)
```

В данном варианте определения функции параметр, преобразующий принятые данные в кортеж, находится в середине списка. При обращении к такой функции все аргументы, которые оказались за пределами списка позиционных параметров, будут объединяться в кортеж с именем «par». А все последующие аргументы должны передаваться функции только в виде именованных аргументов. Если в списке аргументов не окажется элементов, которые можно объединить в кортеж, то соответствующий ему параметр-кортеж, объявленный в заголовке функции, в теле функции будет пустым.

Пример программы, в которой параметр, принимающий кортеж находится в середине списка параметров, приведен в листинге 4.9.

Листинг 4.9. Определение функции, у которой параметр, принимающий кортеж находится в середине списка параметров

```
# Листинг 4.9
print('Функция с параметром кортеж в середине списка параметров def
func(x, *y, z):')
def func(x, *y, z):
    rez = x + sum(y) + z
    return rez

print('-'*60)
print('Вызов функции с разными вариантами передачи аргументов')
print('func(5, 7, 8, z=3) Результат ->', func(5, 7, 8, z=3))
print('func(5, 6, 7, 8, 9, z=3) Результат ->', func(5, 6, 7, 8, 9, z=3))
print('func(5, z=3) Результат ->', func(5, z=3))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.9. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.12).

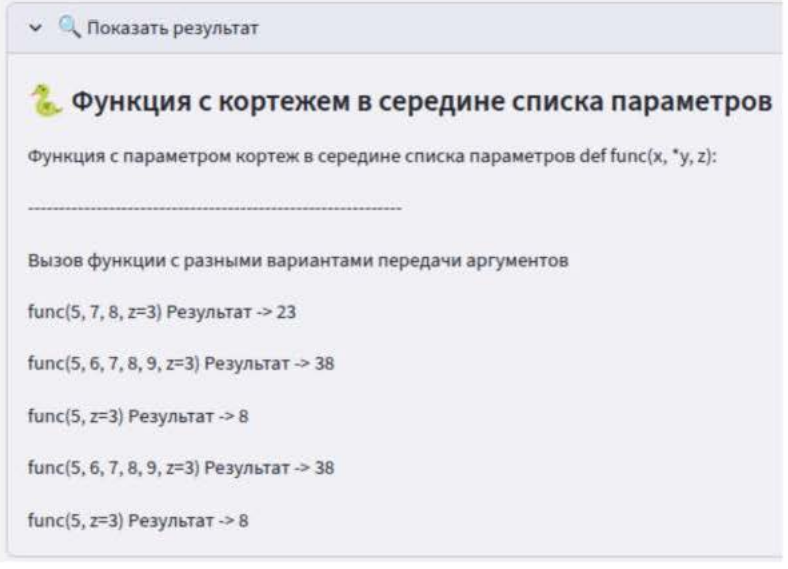


Рис. 4.12. Функция, с параметром, принимающим кортеж, в середине списка параметров

Как видно из данного примера, при третьем обращении к функции задано всего два аргумента (5 и z=3). Когда они были приняты функцией, то переменная «x» получила значение – 5, переменная «z» значение – 3, а кортеж «*y» оказался пустым. При этом программа отработала корректно и не выдала ошибок.

Вариант 3. Теперь рассмотрим возможность передачи в функцию разного количества поименованных аргументов с использованием инструкции «**kwargs». В этой инструкции имя переменной может быть любым, важно, чтобы перед ней стояло две звездочки «**». Однако чтобы код программы был более понятным и удобно читаемым, рекомендуется в качестве имени такого параметра использовать переменную «**kwargs». Благодаря такой инструкции в функции создается словарь, в котором будут содержаться именованные аргументы, переданные функции в момент вызова.

Пример программы, в которой параметр принимает поименованные аргументы, приведен в листинге 4.10.

Листинг 4.10. Определение функции, у которой параметр принимает поименованные аргументы с использованием инструкции «kwargs»**

```
# Листинг 4.10
# Функция принимает именованные аргументы в словарь
def my_fun(**data):
    print("Тип данных ->", type(data))
    print('--------- Получен следующий словарь ---------')
    print(data)
    print('-'*60)
    print('       Вывод содержания словаря')
    for key, value in data.items():
        print(key, ':', value)

print('######## Это первое обращение к функции ##############')
my_fun(Фамилия="Иванов", Имя="Петр", Возраст=22)
print()

print('######## Это второе обращение к функции ##############')
my_fun(Имя="Виктор", Фамилия="Семенов", Email="vik_123@nomail.com",
Страна="Россия", Возраст=25)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.10. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
######## Это первое обращение к функции ##############
Тип данных -> <class 'dict'>
--------- Получен следующий словарь ---------
{'Фамилия': 'Иванов', 'Имя': 'Петр', 'Возраст': 22}
------------------------------------------------------------
      Вывод содержания словаря
Фамилия : Иванов
Имя : Петр
Возраст : 22
```

```
######## Это второе обращение к функции ###############
Тип данных -> <class 'dict'>
--------- Получен следующий словарь ---------
{'Имя': 'Виктор', 'Фамилия': 'Семенов', 'Email': 'vik_123@nomail.com', 'Страна': 'Россия',
'Возраст': 25}
------------------------------------------------------------
      Вывод содержания словаря
Имя : Виктор
Фамилия : Семенов
Email : vik_123@nomail.com
Страна : Россия
Возраст : 25
```

В данном примере была создана функция «my_fun(**data):». В этой функции всего один параметр (**data)», который будет преобразовывать все принятые данные в словарь. В теле функции имеется несколько инструкций:

- вывод сведений о типе полученных данных - type (data);
- вывод всего принятого словаря - type (data);
- в цикле вывод элементов принятого словаря в виде – «ключ:значение».

Затем было выполнено два обращения к этой функции, при этом ей передавалось разное количество поименованных аргументов. В обоих случаях функция корректно приняла эти аргументы и преобразовала их в тип данных – словарь.

Вариант 4. Операторы «*» и «**» могут находиться не только в заголовке функции, но и перед именами аргументов при вызове функций. Пример программы, в которой аргументы при обращении к функциям имею звездочки, приведен в листинге 4.11.

Листинг 4.11. Операторы «*» и «» в заголовке функции и перед именами аргументов при вызове функций**

```
# Листинг 4.11
print('Функция с наличием звездочек в параметрах def func_1(*x, **y):')
def func_1(*x, **y):
    print('x ->', x)
    print('x ->', y)
    return

print('Обращение к функции func_1')
func_1(1, 2, 3, 4, Сотрудник='Иванов А.В.', Должность='Программист')
print('-'*60)

print('Функция с обычными парами def func_2(x, y):')
def func_2(x, y):
    print('x ->', x)
    print('x ->', y)
    return
```

```
print('Первое обращение к функции func_2')
func_2(*[111, 222])
print('-'*60)

print('Второе обращение к функции func_2')
func_2(**{'x': 'Иванов', 'y': 'Петров'})
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.11. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.13).

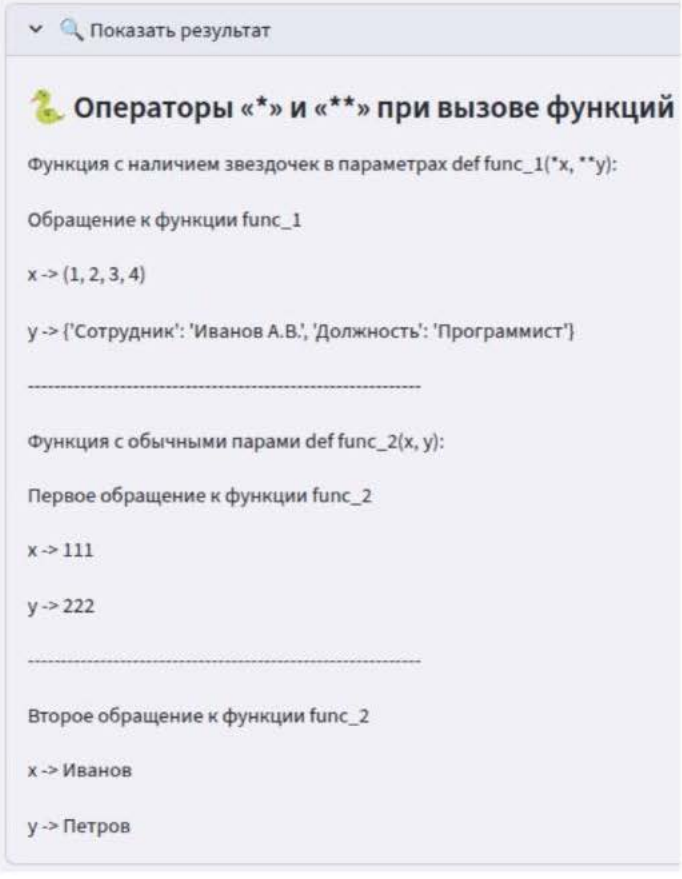


Рис. 4.13. Операторы «*» и «**» в заголовке функции и перед именами аргументов при вызове функций

В этой программе на первом шаге создана функция - def func_1(*x, **y). Она имеет два параметра:

*x – параметр, который может принимать любое количество позиционных аргументов;

**y - параметр, который может принимать любое количество поименованных аргументов.

Такие параметры могут принимать произвольное количество позиционных и поименованных аргументов. В данном примере при обращении к данной функции было передана некая последовательность чисел (позиционные аргументы), и некая последовательность элементов словаря (поименованные аргументы). Все они были корректно приняты функцией, значит, в теле функции над этими данными можно производить любые допустимые операции.

На втором шаге была создана функция - def func_2(x, y). Она имеет два параметра:

- x – параметр, который может принимать только один аргумент;
- y - параметр, который может принимать только один аргумент.

По правилам синтаксиса Python такой функции можно передавать только два аргумента. Однако операторы «*» и «**» позволяют, как бы обойти эти правила, и передать в такую функцию всего один аргумент. В данной программе мы обратились к этой функции дважды:

- func_2(*[111, 222]) – передали в функцию список (из двух элементов);
- func_2(**{'x': 'Иванов', 'y': 'Петров'}) – передали в функцию словарь (из двух элементов).

При передаче в функцию списка перед ним была поставлена звездочка «*». А из предыдущей главы мы знаем, что этот оператор означает распаковку списка. А поскольку в списке два элемента, то их значения после распаковки будут присвоены двум параметрам функции, то есть они примут значения: x=111, y=222.

При передаче в функцию словаря перед ним было поставлено две звездочки «**». А из предыдущей главы мы знаем, что этот оператор означает распаковку словаря. В нашем случае словарь содержит два элемента типа «ключ:значение»:

9. 'x': 'Иванов';
10. 'y': 'Петров'.

Эти два элемента словаря в качестве ключа содержат идентификаторы параметров функции (x, y), а в качестве символьной переменно – значения этих ключей. По сути это именованные аргументы, которые передаются в функцию. А поскольку в словаре два элемента, то их значения после распаковки будут присвоены двум параметрам функции, то есть они примут значения: x=Иванов, y=Петров.

Вариант 5. Определение функции, у которой есть позиционные параметры и параметры со звездочками:

```
def func(par,  * args, ** kwargs)
```

Это наиболее общая форма определения функции с позиционными и именованными аргументами: при вызовах такой функции все дополнительные позиционные аргументы будут объединяться в кортеж с именем «args», а все дополнительные именованные аргументы будут объединяться в словарь с именем «kwargs».

Пример программы, в которой используются все три типа приведенных выше параметров функций, приведен в листинге 4.12.

Листинг 4.12. Определение функции, у которой есть позиционные параметры и параметры со звездочками

```
# Листинг 4.12
def func(x, *y, **z):
    print("Тип данных x ->", type(x))
    print('x->', x)
    print("Тип данных y ->", type(y))
    print('y->', y)
    print("Тип данных z ->", type(z))
    print('z->', z)
    return

func(111, 222, 333, y=1, a=2, b=3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.12. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.14).

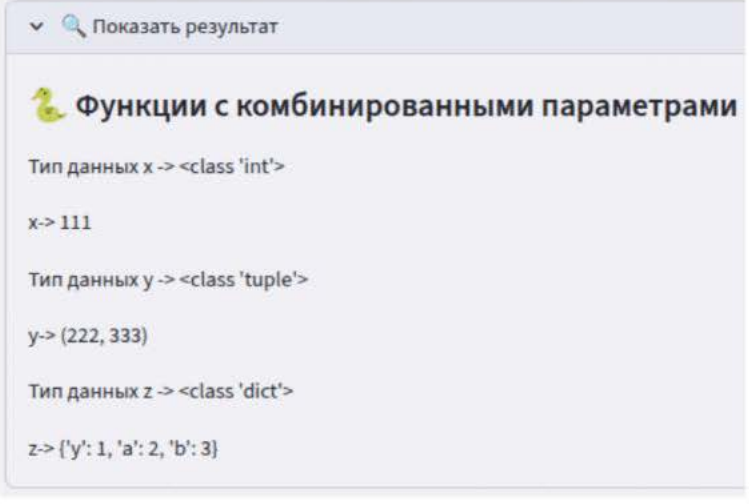


Рис. 4.14. Определение функции, у которой есть позиционные параметры и параметры со звездочками

При обращении к этой функции ей было передано 6 аргументов - три позиционных и три поименованных:

```
func(111, 222, 333, y=1, a=2, b=3)
```

Поскольку в заголовке функции всего один позиционный параметр «x», то ему было присвоено первое значение из списка аргументов x=111 (целое число). Далее следует параметр с одной звездочкой «*y», он принял данные из оставшихся позиционных аргументов и сформировал кортеж y-> (222, 333). Последний параметр функции в качестве префикса имеет две звездочки «**z». Он из всех переданных поименованных аргументов создал словарь z-> {'y': 1, 'a': 2, 'b': 3}.

Первое знакомство с возможными режимами сопоставления аргументов с параметрами функций может вызвать некоторые сложности, но это только на первый взгляд. На самом деле все значительно проще, нужно просто придерживаться нескольких простых правил при определении функций и при обращении к функциям.

Определение функций. При определении функций в заголовке определяемой функции параметры должны указываться в следующем порядке:

- позиционные параметры (arg_1, arg_2, ...);
- параметры со звездочкой для сбора дополнительных позиционных аргументов в кортеж (*args);
- именованные параметры со значениями по умолчанию (name_arg_1=val_1, name_arg_2=val_2, ...);
- параметры с двумя звездочками для сбора дополнительных именованных аргументов в словарь (**name_args).

Вызов функций. При вызове функций порядок передачи аргументов должен быть следующим:

- обычные позиционные аргументы (arg_1, arg_2, ...);
- позиционные аргументы в форме *args (будут распакованы интерпретатором в значения arg_1, arg_2, ...);
- именованные аргументы со значениями (name_arg_1=val_1, name_arg_2=val_2, ...);
- именованные аргументы в форме **name_args, которые будут распакованы интерпретатором в словари в виде пар «ключ: значение».

Конечно, Python предоставляет богатый выбор вариантов передачи аргументов в функции, но не стоит перегружать программный код сложными инструкциями. Чем проще программа, тем она лучше читается, надежнее работает и ее удобнее сопровождать.

4.5. Области видимости переменных в Python

В Python все имена переменных становятся доступны для использования только после того, как им будет присвоено какое-нибудь значение. Так вот, место, где переменной присваивается значение, как раз и определяет область ее видимости. Но поскольку значения переменным могут быть присвоены в трех разных местах программного модуля, то и областей видимости выделяют тоже три.

·8 Если переменная получает значение в теле основного модуля программы, то она становится глобальной для всего этого программного модуля. В общем случае глобальная переменная будет доступна внутри любой функции текущего модуля (в этих функциях не должно быть переменной с таким же именем).

·9 Если переменная получает значение в теле инструкции «def», определяющей функцию, то переменная становится локальной для этой функции. Это касается как параметров функции, так и переменных, созданных в теле функции. Локальные переменные доступны только внутри своей функции. Локальные переменные недоступны за пределами той функции, в которой они были созданы.

·10 Функция в своем теле может иметь определение для другой функции. В этом случае можно говорить о внешней функции и вложенной функции. Если переменная получает

значение в теле инструкции «def» внешней функции, то она становится доступной, как в теле внешней функции, так и в теле вложенной функции (при отсутствии во вложенной функции переменной с таким же именем). Если же переменная получает значение в теле инструкции «def» вложенной функции, то она становится доступной только в теле вложенной функции.

Пример программы, в которой показана видимость переменных в основном модуле, во внешней и вложенной функции, приведен в листинге 4.13.

Листинг 4.13. Видимость переменных в основном модуле, во внешней и вложенной функции

```
# Листинг 4.13
# Это начало тела основного модуля
glob_var_0 = 0.001
print('Задано в начале модуля: glob_var_0 =', glob_var_0)
print('-' * 45)

def func_1(par_1):  # Объявляем внешнюю функцию
    # Это тело func_1, объявляем локальную переменную
    loc_var_1 = 111
    print('Задано в функции func_1: loc_var_1 =', loc_var_1)

    # Это тело func_1, выводим доступные переменные
    print('Доступно в функции func_1: glob_var_0 ->', glob_var_0)
    print('Доступно в функции func_1: par_1 ->', par_1)
    print('-' * 45)

    def func_2():  # Объявляем вложенную функцию
        # Это тело func_2, объявляем локалную переменную
        loc_var_2 = 222
        print('Задано в функции func_1: loc_var_2 =', loc_var_2)

        # Это тело func_2, выводим доступные переменные
        print('Доступно в функции func_2: glob_var_0 ->', glob_var_0)
        print('Доступно в функции func_2: loc_var_1 ->', loc_var_1)
        print('Доступно в функции func_2: par_1 ->', par_1)
        print('-' * 45)
        return  # Конец тела func_2

    # Продолжение тела func_1
    func_2()  # Вызываем вложенную функцию
    return  # Конец тела func_1
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.13. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.15).

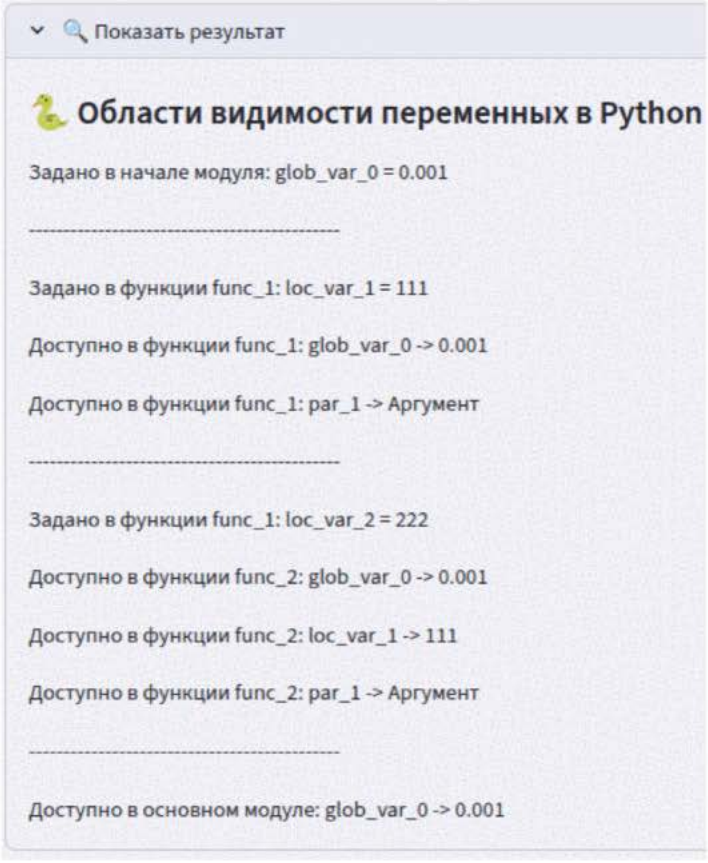


Рис. 4.15. Видимость переменных в основном модуле, во внешней и вложенной функции

Как видно из данного примера, глобальная переменная glob_var_0 доступна и в основном модуле программы, и во внешней функции, и во вложенной функции. Локальная переменная loc_var_1 доступна в теле внешней функции func_1, и в теле вложенной функции func_2 (для нее она является нелокальной переменной). А вот локальная переменная loc_var_2 доступна только в теле вложенной функции func_2. Если мы попробуем использовать локальные переменные функций в основном модуле, то наверняка получим ошибку, так как локальные переменные доступны только в своей области видимости, то есть в инструкциях тела функций.

Наличие обособленных локальных областей видимости и приоритет в них локальных переменных над глобальными с теми же именами позволяет избежать конфликта имен. Поэтому в различных функциях можно использовать одни и те же имена переменных, и они не будут конфликтовать между собой. При этом основной программный модуль может иметь те же имена переменных.

Пример программы, в которой показана возможность использования одних и тех же имен переменных в основном модуле и функциях, приведен в листинге 4.14.

Листинг 4.14. Использование одних и тех же имен переменных в основном модуле и функциях

```
# Листинг 4.14
# Это начало тела основного модуля
var = 0.001
print('В начале программного модуля: var =', var)

# Объявляем 1-ю функцию
def func_1():
    var = 111
    print('В функции func_1: var =', var)
    return

# Объявляем 2-ю функцию
def func_2():
    var = 222
    print('В функции func_2: var =', var)
    return

# Это продолжение тела основного модуля
func_1()  # Вызываем функции func_1
func_2()  # Вызываем функции func_2

print('В конце программного модуля: var =', var)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.14. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.16).



Рис. 4.16. Использование одних и тех же имен переменных в основном модуле и функциях

Как видно из данного примера, в основном модуле и в теле функций используются одна и та же переменная – «var», и в разных частях программы ей были присвоены разные

значения. Ни одна из функций не испортила значение переменной, которое ей было задано в начале программы. Такое разделение видимости переменных облегчают работу программистов, особенно при создании больших программных модулей. Вероятность пересечения имен, а значит и искажение результатов работы программ, сведена к минимуму.

Однако надо помнить, что глобальную переменную нельзя напрямую изменить внутри функции, если только эта переменная не определена в ней с помощью инструкции «global», внутри самой этой функции. Переменную, определенную в родительской функций, так же нельзя напрямую изменить внутри вложенной функции, если только она не определена с помощью инструкции «nonlocal» внутри самой вложенной функции.

Инструкции «global» и «nonlocal» используются в теле функций в тех случаях, когда требуется изменить область видимости и доступности переменных. При этом инструкцию «global» можно использовать, как в простых функциях, так и во вложенных функциях, а вот инструкцию «nonlocal» используются только во вложенных функциях.

С помощью инструкции «global» можно определить глобальную переменную в области видимости функции, при этом такая переменная станет доступной и в других частях программы, включая основной модуль.

При использовании инструкции «nonlocal», можно изменять значения переменных, которые определены в родительской функции (в родительской функции такие переменные должны существовать).

Пример программы, в которой показана возможность использования инструкций «global» и «nonlocal», приведен в листинге 4.15.

Листинг 4.15. Использование инструкций «global» и «nonlocal»

```
# Листинг 4.15
Glob_var = 0.001
print('     Начало работы основного модуля')
print('Значение Glob_var в начале модуля: Glob_var =', Glob_var)
print('-'*50)

def func_1():  # Объявляем внешнюю функцию
    # Начало тела внешней функции
    print('     Начало работы функции func_1()')
    # Glob_var = 100  # Так нельзя- получим ошибку

    global Glob_var  # Даем возможность менять Glob_var в func_1 и
func_2
    # Теперь можно изменять значение Glob_var
    # в функциях func_1 и func_2
    Glob_var = 100
    print('Значение Glob_var в func_1: Glob_var =', Glob_var)
    Loc_var = 11111  # Создаем локальную переменную в func_1
    print('Значение в func_1: Loc_var=', Loc_var)

    def func_2():  # Объявляем вложенную функцию
        # Тело вложенной функции
```

```
        print('     Начало работы функции func_2()')
        Glob_var = 200
        print('Значение Glob_var в func_2: Glob_var =', Glob_var)

        # Loc_var = 222  # Так нельзя- получим ошибку
        nonlocal Loc_var  # Даем возможность менять Loc_var в func_2
        Loc_var = 22222   # Изменяем значение loc_var
        print('Значение Loc_var в func_2: Loc_var =', Loc_var)
        print('-' * 50)
        return

    # Продолжение тела внешней функции
    func_2()  # Вызываем вложенную функцию func_2
    print('     Продолжение работы функции func_1()')
    print('Значение Glob_var по выходу из func_2: Glob_var =', Glob_var)
    print('Значение Loc_var по выходу func_2: Loc_var =', Loc_var)
    print('-' * 50)
    return

func_1()  # Вызываем внешнюю функцию
print('     Завершение работы основного модуля ')
print('Значение Glob_var в конце модуля: Glob_var =', Glob_var)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.15. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
   Начало работы основного модуля
Значение Glob_var в начале модуля: Glob_var = 0.001
--------------------------------------------------
   Начало работы функции func_1()
Значение Glob_var в func_1: Glob_var = 100
Значение в func_1: Loc_var= 11111
   Начало работы функции func_2()
Значение Glob_var в func_2: Glob_var = 200
Значение Loc_var в func_2: Loc_var = 22222
--------------------------------------------------
   Продолжение работы функции func_1()
Значение Glob_var по выходу из func_2: Glob_var = 100
Значение Loc_var по выходу func_2: Loc_var = 22222
--------------------------------------------------
   Завершение работы основного модуля
Значение Glob_var в конце модуля: Glob_var = 100
```

В этой программе определены две переменная Glob_var и Local_var. Переменная Glob_var получила значение в самом начале программного модуля, значит, она будет доступна и в основном модуле, и во всех функциях программы. Однако при этом в функциях значение этой переменной изменить нельзя. Переменная Local_var получила значение в функции func_1(), значит, она будет доступна и в данной функции, и во вложенной функции func_2(). Однако при этом во вложенной функции func_2() значение этой переменной изменить нельзя. Посмотрим, как менялись значения этих переменных по ходу работы программы.

Переменная Glob_var. В начале программного модуля этой переменной присвоено значение 0.001. В функции func_1() была попытка изменить это значение на 100, но интерпретатор этого сделать не даст и сгенерирует ошибку (поэтому данная строка в программе закомментирована). Чтобы дать возможность вносить изменение в данную переменную нужно использовать инструкцию «global» (global Glob_var). После этого она станет доступной для изменения, как в функции func_1(), так и во вложенной функции func_2(). В этих функциях переменная Local_var получила следующие значения:

15. В начале основного модуля Glob_var =0.001;
16. в func_1() - Glob_var =100;
17. в func_2() - Glob_var =200;
18. после выхода из функции в func_2() - Glob_var =100;
19. после выхода из функции в func_1() в основном модуле - Glob_var =100.

Обратите внимание, что при выходе из вложенной функции значение переменной осталось таким же, как и во внешней функции, где она была определена - Glob_var =100. По выходу из внешней функции в основном модуле эта переменная сохранила свое значение - Glob_var =100. Отсюда можно сделать вывод, что переменная, которая объявлена как global и создана в функции, будет сохранять свое значение, как в данной функции, так и в основном модуле. При этом во вложенной функции можно использовать переменную с таким же именем, но в ней она будет локальной и может принимать любые значения.

Переменная Local_var. В теле функции func_1() этой переменной присвоено значение 11111. В функции func_2() была попытка изменить это значение на 22222, но интерпретатор этого сделать не даст и сгенерирует ошибку (поэтому данная строка в программе закомментирована). Эта переменная является локальной и может использоваться только во внешней функции. Чтобы дать возможность вносить изменение в данную переменную во вложенной функции нужно использовать инструкцию «nonlocal» (nonlocal Local_var). После этого она станет доступной для изменения, как в функции func_1(), так и во вложенной функции func_2(). В этих функциях переменная Local_var получила следующие значения:

5 в func_1() - Local_var =11111;

6 в func_2() - Local_var =22222.

После выхода из вложенной функции func_2() переменная Local_var сохранила свое последнее значение - Local_var =22222. Отсюда можно сделать вывод, что переменная, которая объявлена как «nonlocal» и создана во внешней функции, будет доступна и во вложенной функции, где можно менять ее значение.

Из всего выше сказанного следует простое правило поиска имен, которое использует интерпретатор Python, когда внутри функции выполняется обращение к неизвестному имени переменной:

- интерпретатор пытается отыскать имя переменной в локальной области видимости (в теле функции);

- если переменная не найдена, то она ищется в локальной области любой объемлющей инструкции (внешней функции «def» или в выражении «lambda»);
- если переменная не найдена, то она ищется в глобальной области видимости основного модуля программы;
- если искомого имени не обнаружено, то интерпретатор выведет сообщение об ошибке.

В Python получить сведения о локальных и глобальных переменных можно при помощи следующих встроенных функций.

- globals() – возвращает словарь со значениями переменных из текущей глобальной области видимости модуля, в котором ключами служат имена этих переменных. Стоит заметить, что функция возвращает такой словарь даже в тех случаях, когда она вызывается из локальных областей видимости функций и методов.
- locals() – обновляет и возвращает словарь со значениями переменных, использующихся в текущей локальной области видимости функции или метода, в котором ключами служат имена этих переменных.

Пример программы, в которой показана возможность использования этих функций, приведен в листинге 4.16.

Листинг 4.16. Получение сведений о локальных и глобальных переменных

```
# Листинг 4.16
glob_var_1 = 0.1  # Инициализируем 1-ю глобальную переменную

# Объявляем внешнюю функцию
def func_1():
    loc_var_1 = 1  # Локальная переменная для func_1
    print('    Это печать из внешней функции - func_1')
    print('Словарь доступных локальных переменных для func_1')
    print(locals())
    print('Словарь глобальных переменных')
    print(globals())
    print('-'*60)

    # Объявляем вложенную функцию
    def func_2():
        loc_var_2 = 2  # Локальная переменная для func_2
        print('    Это печать из вложенной функции - func_2')
        print('Локальные переменные для func_2 ->', locals())
        print('-' * 60)
        return

    func_2()  # Вызываем вложенную функцию func_2.
    return
```

```
func_1()  # Вызываем функцию -func_1()

# Словарь глобальных переменных
print('    Это печать из основного модуля')
print('Словарь глобальных переменных')
print(globals())
print('Словарь локальных переменных')
print(locals())
print('-' * 60)

glob_var_2 = 0.2  # Инициализируем 2-ю глобальную переменную
print('Обновленный словарь глобальных переменных')
print(globals())
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.16. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
   Это печать из внешней функции - func_1
Словарь доступных локальных переменных для func_1
{'loc_var_1': 1}
Словарь глобальных переменных
{… , 'glob_var_1': 0.1, 'func_1': <function func_1 at 0x00000000021A0DC8>}
------------------------------------------------------------
   Это печать из вложенной функции - func_2
Локальные переменные для func_2 -> {'loc_var_2': 2}
------------------------------------------------------------
   Это печать из основного модуля
Словарь глобальных переменных
{… , 'glob_var_1': 0.1, 'func_1': <function func_1 at 0x00000000021A0DC8>}
Словарь локальных переменных
{…, 'glob_var_1': 0.1, 'func_1': <function func_1 at 0x00000000021A0DC8>}
------------------------------------------------------------
Обновленный словарь глобальных переменных
{… 'glob_var_1': 0.1, 'func_1': <function func_1 at 0x00000000021A0DC8>, 'glob_var_2':
0.2}
```

Поскольку список глобальных переменных довольно большой, то в данной распечатке приведена только часть словаря с глобальными переменными. Когда вы получите его на своем компьютере, он будет гораздо больше, но здесь нам важен конец этого словаря, где присутствуют значения глобальных переменных, которые именно мы объявили в программном модуле. Обратите внимание, что эти функции для основного программного модуля выдают одинаковый перечень значений для локальных и глобальных переменных, а для функций он разный.

4.6. lambda-выражения или анонимные функции

Поскольку в Python инструкции не могут использоваться внутри выражений, для этих целей в язык была введена дополнительная возможность в виде специальных конструкций, называемых lambda-выражениями, которые позволят создавать объекты-функций там, где интерпретатор ожидает встретить выражение. Эти конструкции еще называют анонимными функциями, безымянными функциями или lambda-функциями. В отличие от инструкций «def» такие функции не имеют имени.

В общем виде lambda-выражение состоит из ключевого слова lambda, за которым перечисляются один или более аргументов и после двоеточия записывается выражение, использующее эти аргументы:

lambda arg_1, arg_2, ..., arg_n: <выражение>.

Здесь стоит заметить, что после двоеточия записывается именно выражение, а не блок инструкций. По сути, такое выражение похоже на то, которое помещается в инструкцию «return» при определении обычной функции. Это делает lambda-выражения менее универсальными по сравнению с инструкциями «def». Однако такая реализация предусмотрена намеренно, так как lambda-выражения дают возможность создавать простые функции в рамках всего одной строки программного кода.

Пример программы, в которой показано использование lambda-выражения, приведен в листинге 4.17.

Листинг 4.17. Использование lambda-выражения

```
# Листинг 4.17
# Обычная функция с именем
def my_func(x, y, z):
    rez = x + y + z
    return rez

sum_0 = my_func(5, 10, 15)
print('Результат sum_0-> ', sum_0)

# lambda функция
# явно присваиваем его переменной
lambda_1 = lambda x, y, z: x + y + z
sum_1 = lambda_1(5, 10, 15)
print('Результат sum_1-> ', sum_1)

# Используем значения по умолчанию
lambda_2 = lambda x=8, y=4: x / y
rez_2 = lambda_2()
rez_3 = lambda_2(12)
print('rez_2 =', rez_2)
print('rez_3 =', rez_3)

# Собираем аргументы в кортеж
lambda_3 = lambda *nums: sum(nums)
```

```
sum_2 = lambda_3(2, 3)
sum_3 = lambda_3(1, 2, 3, 4, 6, 6)
print('sum_2 =', sum_2)
print('sum_3 =', sum_3)

# Используем простейшую логику
lambda_4 = lambda x, y: x / y if y != 0 else None
log_1 = lambda_4(20, 5)
log_2 = lambda_4(20, 0)
print('log_1 =', log_1)
print('log_2 =', log_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.17. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.17).

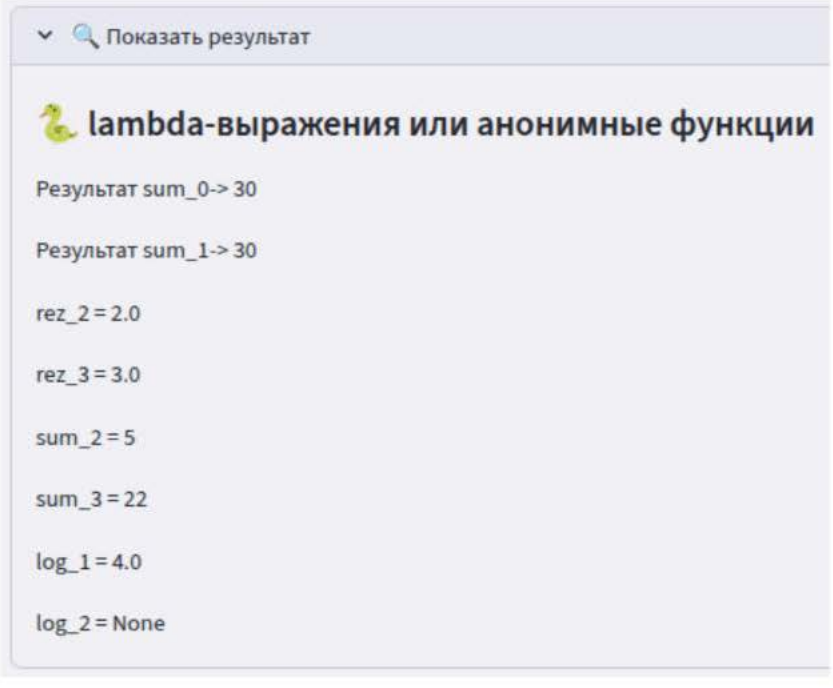


Рис. 4.17. Использование lambda-выражения

В этой программе сначала создана традиционная функция с именем my_func(), а затем несколько объектов на основе lambda-выражений. Затем были выполнены обращения к обычной функции и к lambda-функциям. Здесь lambda-выражения присваивались переменным, и уже через них выполнялось обращение к lambda-функциям и передача им аргументов.

Как видно из данного примера, lambda-выражения действительно очень похожи на обыкновенные функции: в них также действуют специальные режимы сопоставления аргументов, имеется тело с кодом, а на выходе получается результат в виде объекта. В теле lambda-выражений может находиться только одно единственное выражение, а использование даже простейших инструкций вообще запрещено. Кроме того, возвращаемый объект функции нужно либо сразу вызывать при помощи круглых скобок, либо явно присваивать переменной. Зато lambda-выражения можно смело использовать внутри других выражений, что делает программный код очень компактным.

Пример программы, в которой показаны разные варианты использование lambda-выражений и передачи им аргументов, приведен в листинге 4.18.

Листинг 4.18. Варианты использование lambda-выражений и передачи им аргументов

```
# Листинг 4.18
print('lambda-функция с одним аргументом')
f1 = lambda x: x * x + x**3 - x / 2
rez = (22 + f1(2)) / 2 + f1(3) - f1(4)
print('rez =', rez)
print('-'*15)

print('lambda-функция с двумя аргументами')
f2 = lambda x, y: x ** y - x + (x * y) / 2 + y / x
print('f2 =', f2(5, 3))
print('f2 =', f2(8, 2))
print('-'*15)

print('Список из двух lambda-функций')
list_2 = [lambda x: x**2, lambda x: x**3]
print('list_2 =', list_2[0](3))
print('list_2 =', list_2[1](3))
print('-'*15)

print('И lambda-функция, и ее аргументы внутри выражения')
rez_1 = 5 + (lambda x, y: x**y)(5, 2) - 15
print('rez_1 =', rez_1)
print('-'*15)

print('lambda-выражение с вложенным lambda-выражением')
rez_2 = (lambda x: 7 + (lambda a, b: a**b)(x, 2))(5)
print(rez_2)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.18. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
lambda-функция с одним аргументом
rez = -27.0
---------------
lambda-функция с двумя аргументами
f2 = 128.1
f2 = 64.25
---------------
Список из двух lambda-функций
list_2 = 9
list_2 = 27
---------------
И lambda-функция, и ее аргументы внутри выражения
rez_1 = 15
---------------
lambda-выражение с вложенным lambda-выражением
32
```

Вкладывать одно lambda-выражение в другое конечно можно, но это уже не упрощает, а усложняет программный код, поскольку здесь нужно учитывать область видимости переменных во внешних и вложенных функциях. Поэту для удобства читаемости программного кода такие конструкции лучше не использовать.

4.7. Рекурсивные функции

Рекурсивная функция – это функция, которая содержит код вызова самой себя в целях организации циклического процесса. Рекурсивные функции являются вполне обычными функциями со всеми присущими им особенностями. Разница лишь в том, что внутри тела такие функции содержат вызов самой себя.

Пример программы, в которой реализована рекурсивная функция, приведен в листинге 4.19.

Листинг 4.19. Рекурсивная функция

```
# Листинг 4.19
# Рекурсивная функция
def sum_n(n):
    if n == 0:
        return n
    else:
        return n + sum_n(n - 1)

print('Сумма чисел - 0 ->', sum_n(0))
print('Сумма чисел от 0 до 1 ->', sum_n(1))
print('Сумма чисел от 0 до 2 ->', sum_n(2))
print('Сумма чисел от 0 до 3 ->', sum_n(3))
print('Сумма чисел от 0 до 10 ->', sum_n(10))
print('Сумма чисел от 0 до 20 ->', sum_n(20))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.19. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.18).

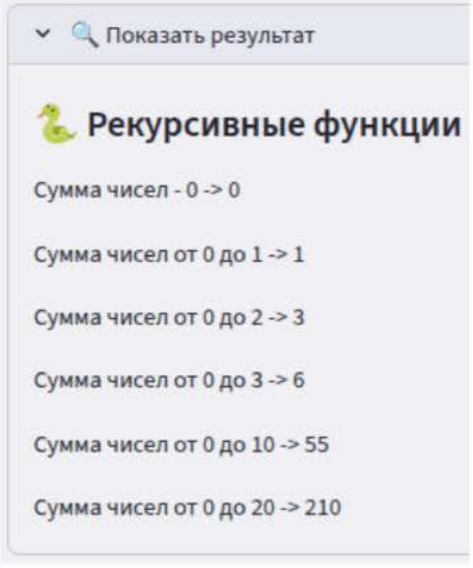


Рис. 4.18. Рекурсивная функция

В нашем примере создана функция с именем «sum_n», которая подсчитывает сумму чисел от 0 до n. Здесь в заголовке функции «n» – это параметр, который принимает значение максимального числа, до которого нужно в цикле суммировать все предыдущие числа. В теле функции имеется инструкция, которая вызывает функцию «sum_n», то есть функция вызывает сама себя. То есть в функции организован бесконечный цикл, и если не создать инструкцию выхода из такого цикла, то программа не остановится, пока не будет переполнена память. Поэтому тело функции содержит условную инструкцию «if», которая остановит цикл в том случае, когда значение переданного ей аргумента будет равно нулю. Если же при текущем вызове функции значение аргумента будет больше нуля, то в блоке «else» условной инструкции «if» будет вычислено выражение n + sum_n(n-1), в котором функция вызывает себя. Эта функция, прежде, чем вернуть конечный результат в точку вызова из основной программы, будет вызывать себя до тех пор, пока в выражении для возврата не останется незавершенных вызовов, то есть пока параметр «n» не станет равным нулю.

Изменим код предыдущей программы, в которой реализована рекурсивная функция, и посмотрим, как в цикле изменяется параметр «n» (листинг 4.20).

Листинг 4.20. Рекурсивная функция

```
# Листинг 4.20
# Рекурсивная функция
def sum_n(n):
    print('n = ', n)
    if n == 0:
       return n
    else:
        return n + sum_n(n - 1)

print('Сумма чисел от 0 до 4 ->', sum_n(4))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.20. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.19).

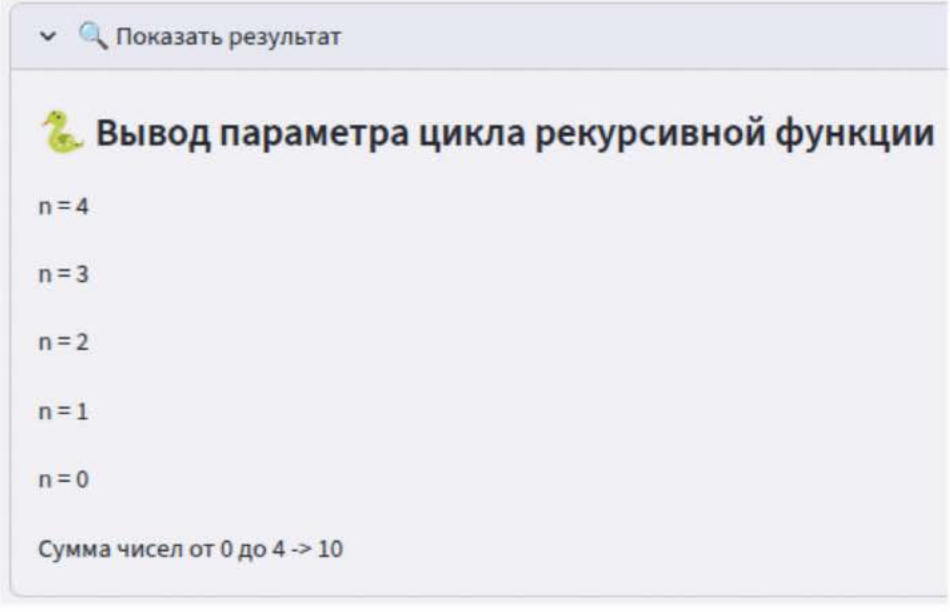


Рис. 4.19. Рекурсивная функция (параметры цикла)

Как видно из данного примера, рекурсивная функция начала работать с момента, когда при вызове ей было передано число – 4. Затем в цикле этот параметр уменьшался до нуля, после чего в точку вызова функции была возвращена сумма чисел от 0 до 4.

Следует заметить, что в языке Python рекурсивные функции применяются не так часто, так как имеется инструкция для организации цикла «while», которая более естественно подходит для решения аналогичных задач.

4.8. Атрибуты функций

Поскольку функции в Python являются обычными объектами, то можно получать доступ к предопределенным атрибутам функций, а также добавлять, изменять или удалять собственные атрибуты. Когда вы сами проектируете систему и пишите собственный программный код, то вы, конечно же, имеете полное представление о том, какие параметры функция принимает и что возвращает. Но когда вы используете сторонние библиотеки, то такой информации у вас нет. Или когда через какое то время нужно модифицировать собственный программный код, особенно если в нем десятки модулей, то возникает потребность в получении информации, какая функция что принимает и в каком виде, и что отдает. Здесь на помощь придут атрибуты функций. У функций есть предопределенные атрибуты, которые можно использовать, Однако если их не хватает, то можно для функций создать собственные пользовательские атрибуты.

Пример программы, в которой можно оперировать с атрибутами функции, приведен в листинге 4.21.

Листинг 4.21. Оперирование с атрибутами функции

```
# Листинг 4.21
# Объявляем функцию и получаем объект
def func(n):
    print('Итог вычисления n**2 =', n**2)
    func.count += 1
    return

print('Тип объекта:', func)
print('Все встроенные атрибуты функции')
print(dir(func))
print('-'*30)
print('Имя функции:', func.__name__)
print('Размер функции в байтах:', func.__sizeof__())
print('-'*30)

func.count = 0  # Создаем свой атрибут
print('Начальное значение пользовательского атрибута count ->',
func.count)

# Используем пользовательский атрибут
func(2)
print('Количество обращений к функции count ->', func.count)  # вывод
значения атрибута
print('-'*30)

func(4)
print('Количество обращений к функции count ->', func.count)  # вывод
значения атрибута
print('-'*30)
```

```
func(6)
print('Количество обращений к функции count ->', func.count)  # вывод
значения атрибута
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.21. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Тип объекта: <function func at 0x0000000002870AF8>
Все встроенные атрибуты функции
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__',
'__getattribute__', '__globals__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__',
'__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
------------------------------
Имя функции: func
Размер функции в байтах: 112
------------------------------
Начальное значение пользовательского атрибута count -> 0
Итог вычисления n**2 = 4
Количество обращений к функции count -> 1
------------------------------
Итог вычисления n**2 = 16
Количество обращений к функции count -> 2
------------------------------
Итог вычисления n**2 = 36
Количество обращений к функции count -> 3
```

В этой программе была создана функция с именем func() и пользовательский атрибут этой функции func.count. Этот атрибут в данной программе используется для того, чтобы узнать – сколько обращений к данной функции было в ходе работы программы. В теле функции выполняется всего два действия: рассчитывается квадрат полученного числа, увеличивается значение счетчика вызова функции.

В ходе работы программы были получены и выведены на экран различные атрибуты функции, а также получено количество обращений к функции с использованием пользовательского атрибута.

Как видно из данного примера, для получения списка всех предопределенных атрибутов функции нужно просто воспользоваться встроенной (стандартной) функцией Python - «dir». Если же необходим доступ к конкретному атрибуту, следует указать его имя в формате «func.attr», где «func» – имя функции, а «attr» – имя атрибута. Что касается пользовательских атрибутов, то они создаются и изменяются простым присваиванием требуемого значения имени атрибута, который также записывается после имени функции через точку в формате «func.attr». Собственные атрибуты позволяют эффективно и просто

хранить информации о состоянии непосредственно в объекте функции, отказавшись от использования других приемов, таких как применение глобальных или нелокальных переменных и классов. Огромным плюсом при этом является то, что в отличие от нелокальных переменных, атрибуты функции доступны в любом месте программы, где доступна сама функция.

4.9. Аннотирование и документирование функций

Начиная с версии Python 3.0, у объектов-функций появился новый встроенный атрибут __annotations__. Он предназначен для хранения краткого описания (аннотаций) аргументов функции, и ее возвращаемого значения. В ходе работы программы интерпретатор Python не обращает внимания на аннотации, и они не влияют на работу программы. Аннотирование вообще не является обязательным при создании функций. Но если аннотирование присутствует, то такая функция является аннотированной, и всегда можно получить значения атрибутов, воспользовавшись синтаксической конструкцией func.__annotations__.

Аннотации параметров функции представляют собой необязательные выражения, следующие за именем самого параметра, например:

```
def my_fun (a: выражение_1, b: выражение_2) -> выражение_3:
```

Здесь в скобках перечислены параметры функции, а через двоеточие следуют выражения. Выражениями могут быть строки с любым содержанием, или указатели типа данных (например, int, float). После скобок стоит аннотирование возвращаемого значения, для этого используется синтаксис «->». Допустим мы аннотируем функцию следующим заголовком:

```
def my_fun(p_1, fio: 'Это ФИО сотрудника', adg: int) -> float:
```

В этом случае аннотации параметров будут иметь следующий смысл:

- p_1 – этот параметр не имеет аннотации;
- fio – этот параметр будет принимать фио сотрудника в виде символьной строки;
- age – этот параметр будет принимать целое значение;
- float – функция будет возвращать значение в виде числа с плавающей точкой.

Пример программы, в которой создана аннотируемая функция, приведен в листинге 4.22.

Листинг 4.22. Аннотируемая функция

```
# Листинг 4.22
# Объявляем аннотированную функцию
def my_func(name: 'Это имя', age: int) -> str:
    '''Это функция "my_func", она формирует nic пользователя'''
    nick = name + '_' + str(age)  # Составляем ник пользователя
    return nick  # Возвращаем его

print('Обращаемся к функции')
print(my_func('Иванов_Петр', 35))

print('----- Получаем сведения о параметрах функции -----')
```

```
print(my_func.__doc__)
print(my_func.__annotations__)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.22. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.20).

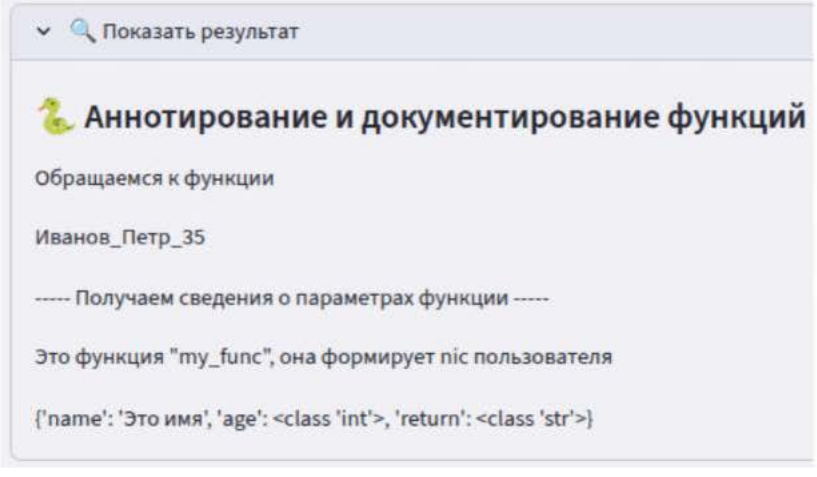


Рис. 4.20. Аннотируемая функция

Как видно из данного примера, аннотации функции находятся в заголовке инструкции «def» в виде произвольных выражений, которые ассоциируются с ее аргументами и возвращаемым значением. Для аргументов аннотации указываются через двоеточие сразу после имени аргумента, а для возвращаемого значения – после пары символов «->» вслед за списком аргументов. Все аннотации, если конечно они присутствуют в объявлении функции, интерпретатор собирает в словарь, который затем присоединяет к объекту созданной функции. Имена аргументов в этом словаре становятся ключами, а сами аннотации их значениями. При этом для возвращаемого значения аннотация сохраняется под ключом «return».

Что касается строки документирования, то она должна записываться в начале тела функции в тройных кавычках. Интерпретатор автоматически присоединит эту строку к объекту функции, и она станет доступна в качестве значения метода функции «__doc__». В этой программе в строке документирования функции написана фраза «Это функция "my_func", она формирует nic пользователя». Что касается созданных в заголовке функции аннотаций параметров и возвращаемого значения, то они могут быть получены вызовом метода функции «.__annotations__».

Стоит обратить внимание на еще один важный момент: в аннотированных аргументах, которые используют значения по умолчанию, аннотация записывается перед значением аргумента по умолчанию в формате:

```
def func(arg_1: annot_1 = val_1, arg_2: annot_2 = val_2, ...) -> return_annot:
```

Пример программы, в которой создана аннотируемая функция, имеющая аргументы со значением по умолчанию, приведен в листинге 4.23.

Листинг 4.23. Аннотируемая функция, имеющая аргументы со значением по умолчанию

```
# Листинг 4.23
# Объявляем аннотированную функцию
def my_func(cena: 'Цена - руб', nalog: 'Налог -%' = 20) -> float:
    nds = cena * nalog / (100 + nalog)
    tovar = cena - nds
    return tovar, nds

print('      Получаем сведения о параметрах функции')
print(my_func.__annotations__)
print('-'*60)

cena = 200  # цена товара с НДС
print('Цена товара с НДС - ', cena)
print('-'*30)

print('     Расчет при ставке НДС 20%')
tovar, nds = my_func(cena)
itogo = round(tovar, 2) + round(nds, 2)
print('Стоимость товара - ', round(tovar, 2))
print('НДС от стоимость товара - ', round(nds, 2))
print('Итого - ', itogo)
print('-'*30)

print('     Расчет при ставке НДС 10%')
tovar, nds = my_func(cena, 10)
itogo = round(tovar, 2) + round(nds, 2)
print('Стоимость товара - ', round(tovar, 2))
print('НДС от стоимость товара - ', round(nds, 2))
print('Итого - ', itogo)
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.23. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.21).

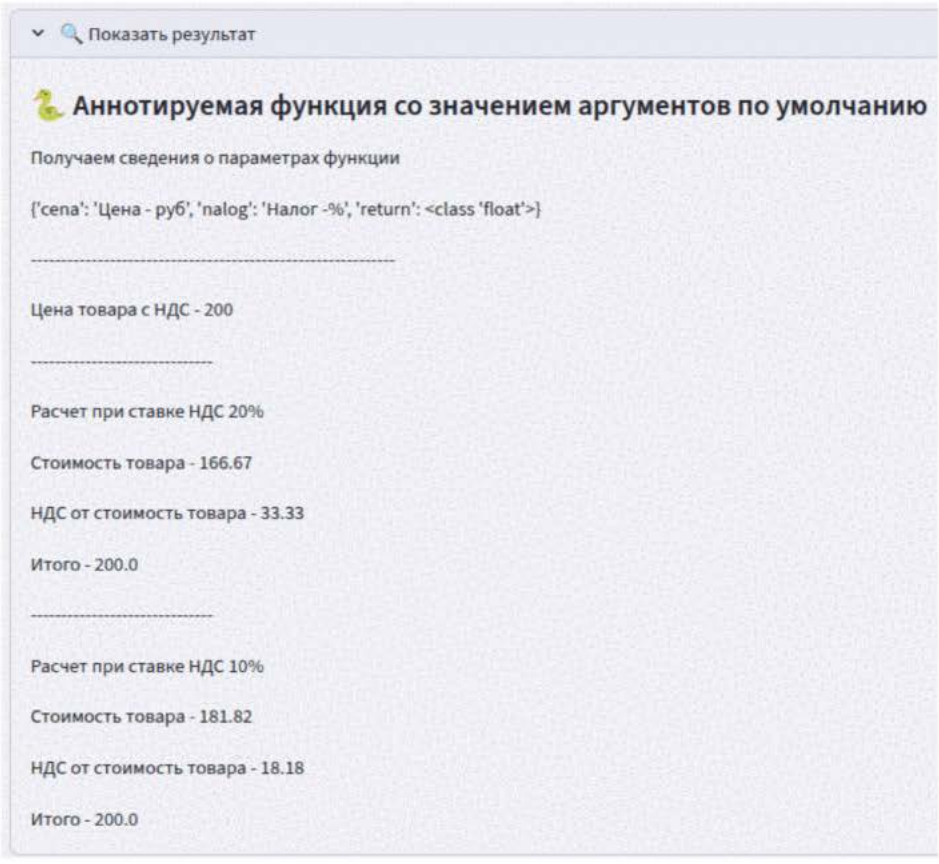


Рис. 4.21. Аннотируемая функция, имеющая аргументы со значением по умолчанию

В этой программе создана функция, которая позволяет из стоимости товара выделить налог НДС. В заголовке функции присутствуют аннотируемые параметры:

6. cena – параметр, который принимает цену товара вместе с налогом НДС (имеет аннотацию - 'Цена');
7. nalog - параметр, который принимает налог НДС (имеет аннотацию 'Налог' и значение налога по умолчанию – 20);
8. float – тип данных, которые будут возвращены данной функцией.

На первом шаге мы в качестве справки получили сведения о параметрах функции в виде словаря:

```
{'cena': 'Цена - руб', 'nalog': 'Налог -%', 'return': <class 'float'>}
```

Из этого словаря видно, что при обращении к данной функции в первом аргументе нужно указывать цену товара в рублях, во втором аргументе величину налога НДС в процентах, а возвращать функция будет данные с типом float – числа с плавающей точкой. Причем функция будет возвращать два значения: стоимость товара без НДС (tovar), и величину налога (nds).

На втором шаге была задана цена товара с НДС (cena) – 200 руб.

На третьем шаге мы обратились к функции и передали ей в качестве аргумента только цену товара. В этом случае величина налога (20%) функция получит из второго параметра, который имеет значение по умолчанию.

На третьем шаге мы обратились к функции и передали ей значение двух аргументов: цена товара и величину НДС (10% - некоторые товары социального значения облагаются пониженным налогом). В этом случае величина налога (10%) будет передана второму аргументу и значение по молчанию (20) будет заменено на значение -10.

Самое главное, что нужно подчерпнуть из данного примера, это то, что для аргументов, которые имеют значение по умолчанию, аннотация располагается между именем аргумента и его значением.

4.10. Функции декораторы

Функции декораторы — один из самых полезных инструментов в Python, однако новичкам их работа может показаться непонятной, потому что они основаны на инструкциях, выходящих за рамки традиционного процедурного программирования.

Декоратор — это функция, которая позволяет обернуть (вставить в свое тело) другую функцию для расширения ее функциональности (без изменения кода вставленной функции), и вернуть результат работы в точку вызова.

Иными словами декоратор это дополнительная функция, которая расширяет функциональные возможности другой (базовой) функции без изменения программного кода в самой базовой функции. Фактически в тело одной функции (декоратора) как бы внедряется тело другой функции, и возвращается результат работы сразу двух функций.

Предположим, что у вас есть программный модуль, в котором присутствует функция с длинным кодом и сложной логикой. Это может быть собственная программа, которую вы когда то написали, или сторонняя программа из библиотеки с открытым исходным кодом. Вы хотите расширить возможности этой функции, но боитесь испортить логику ее работы. Вот тут и потребуется функция-декоратор. Нужно просто создать новую функцию-декоратор, вложить в нее базовую функцию и дописать новый программный код, который расширит функционал базовой функции. При этом не нужно в функцию - декоратор физически помещать полный код базовой функции, достаточно при вызове функции - декоратора в качестве аргумента просто передать ей имя базовой функции. На первый взгляд эти рассуждения кажутся сложными и запутанными, поэтому для большей ясности перейдем к примерам и, шаг за шагом, распутаем этот «клубок».

Пример программы, в которой создана базовая функция, а потом дополнена новыми инструкциями традиционным способом, приведен в листинге 4.24.

Листинг 4.24. Функции декораторы

```
# Листинг 4.24
# Это базовая функция
def base_func():
    print("Это отработал код базовой функции (он не меняется никогда)")
    return

base_func()  # Вызов базовой функции
```

```
print('-' * 60)

# Это новая функция
def new_func():
    print("Это отработал код, добавленный перед базовой функцией -
base_func()")
    print("Это отработал код базовой функции (он не меняется никогда)")
    print("Это отработал код, добавленный после базовой функции -
base_func()")
    return

new_func()  # Вызов новой функции
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.24. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.22).



Рис. 4.22. Две функции без использования декоратора

В этом примере создана базовая функция – base_func(), которая имеет инструкцию всего из одной строчки. Для того чтобы ее модифицировать и добавить новый функционал традиционным способом, нужно вносить изменения непосредственно в программный код базовой функции. В данном примере к коду базовой функции добавлено две новые инструкции: одна перед ранее существовавшей инструкцией, вторая после нее. Для наглядности мы это сделали не в самой базовой функции, а создали новую функцию с именем – new_func().

А теперь выполним те же действия с использованием функции – декоратора. Пример программы, в которой создана базовая функция, а потом дополнена новыми инструкциями через функцию - декоратор, приведен в листинге 4.25.

Листинг 4.25. Функции декораторы

```
# Листинг 4.25
# Это базовая функция
def base_func():
    print("Это отработал код базовой функции (он не меняется никогда)")

# Это функция декоратор
def fun_decor(par_base_func):  # Сюда принимаем базовую функцию
    def wrap_func():     # это фунция - обертка для принятой функции
        print("Это отработал код, добавленный перед базовой функцией -
base_func()")
        par_base_func()  # Здесь выполняется код базовой функции
        print("Это отработал код, добавленный после базовой функции -
base_func()")
    return wrap_func  # Возвращается результат работы двух функций


base_func()  # Вызов базовой функции
print('-' * 60)

# Создаем новый объект - функцию
new_func = fun_decor(base_func)  # Функции декоратору передаем имя
базовой функции
new_func()  # Вызов новой функции
print('-' * 60)

base_func()  # Повторный вызов базовой функции
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.25. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.23).

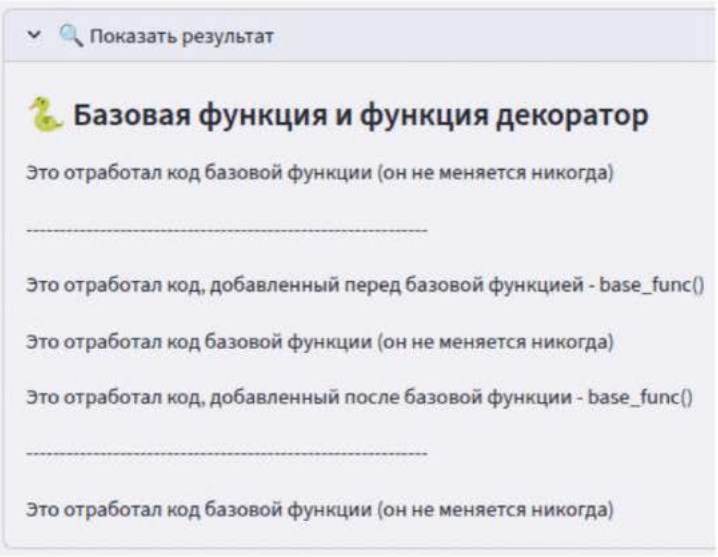


Рис. 4.23. Функции декораторы

В этом примере создана базовая функция – base_func(), которая имеет инструкцию всего из одной строчки. Для того чтобы добавить в нее новый функционал, была создана функция декоратор с именем – fun_decor(). Параметр этой функции - «par_base_func» будет принимать имя базовой функции в момент обращения. В теле функции декоратора fun_decor() находится другая вложенная функция с именем – wrap_func(), ее называют функцией – оберткой. Она как бы обертывает (заворачивает в себя) тело принятой функции и, одновременно, содержит ряд инструкций, которые дополняют функционал принятой функции. Функция обертка возвращает в точку вызова результат работы сразу двух функций: результаты выполнение инструкций принятой функции, и результаты работы собственных инструкций.

Поскольку в Python любая функция это объект, то на базе двух созданных функций (base_func и fun_decor) теперь можно создать новую функцию. Мы создали такую функцию с именем new_func() с помощью инструкции:

```
new_func = fun_decor(base_func)
```

В этой инструкции мы вызываем функцию декоратор (fun_decor), а в качестве параметра указываем имя базовой функции (base_func). Фактически мы вызываем одну функцию и передаем ей другую функцию. В программном коде, приведенном выше, такой вызов выполняется с помощью инструкции – new_func(). Что происходит после вызова функции декоратора:

- из точки вызова в функцию декоратор передается имя базовой функции;
- параметр, который находится в заголовке функции декоратора, принимает имя базовой функции;
- функция-обертка, которая находится в теле функции декоратора, выполняет собственные инструкции и инструкции полученной базовой функции;
- результаты работы функции-обертки возвращаются в точку вызова.

Таким образом, был дополнен функционал базовой функции, при этом ее программный код не менялся. Это видно из результатов работы вышеприведенного примера. В нем мы сначала вызвали базовую функцию, затем функцию декоратор, и снова базовую функцию. В первом случае отработали инструкции только базовой функции, во втором случае отработали расширенные инструкции базовой функции, и, в заключение, снова отработали инструкции только базовой функции. На первый взгляд такой поход кажется сложным, однако в том случае, когда программный код базовой функции имеет сложную структуру, это лучший вариант.

В предыдущем примере для того, чтобы одна функция вызвала другую, использовалась следующая инструкция:

```
new_func = fun_decor(base_func)
```

Она вроде бы логически и понятна, но выглядит как то неуклюже. В Python есть более простой способ указать на то, что функция имеет декоратор – это использование символа - @. Пример программы, в которой показано использование символа - @ для декорирования функций, приведен в листинге 4.26.

Листинг 4.26. Функции декораторы

```
# Листинг 4.26
# Это функция декоратор
def fun_decor(par_base_func):  # Сюда принимаем базовую функцию
    def wrap_func():  # это фунция - обертка для принятой функции
        print("Это отработал код, добавленный перед базовой функцией")
        par_base_func()  # Здесь выполняется код базовой функции
        print("Это отработал код, добавленный после базовой функции")

    return wrap_func  # Возвращается результат работы двух функций

# Это обычная функция
def base_func():
    print("Это инструкция из base_func")

base_func()  # Первый вызов базовой функции
print('-' * 60)

# Это та же функция, имеющая декоратор с обновлениями
@fun_decor
def base_func():
    print("Это инструкция из base_func")

base_func()  # Второй вызов базовой функции
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.26. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.24).

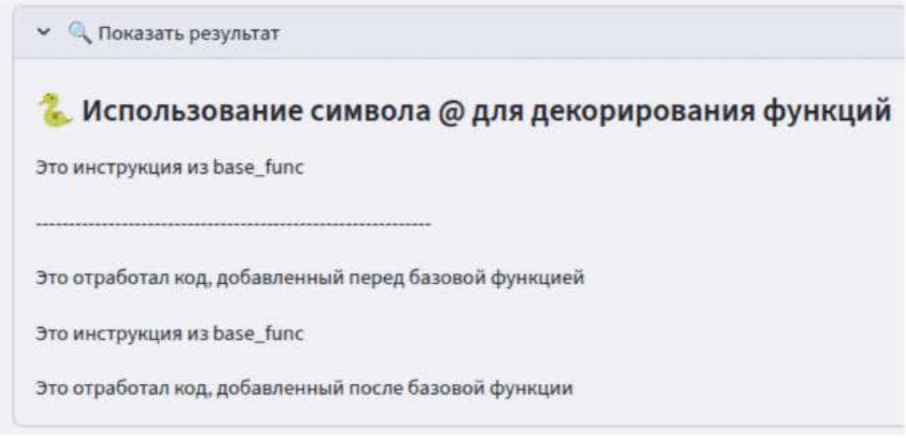


Рис. 4.24. Функции декораторы с символом @

В этой программе создана функция декоратор с именем – fun_decor(). Затем мы создали обычную функцию с именем base_func(), в которой всего одна строчка программного кода, и вызвали ее на исполнение. В результате на печать была выведена всего одна строка.

Затем мы создали туже функцию, с тем же именем, но перед ней написали строку с инструкцией - @fun_decor. Этим самым мы указали, что данная функция имеет дополнения, которые содержаться в другой функции с именем fun_decor, то есть эти дополнения находятся в функции декораторе. Теперь, когда мы обратились к функции с тем же именем, мы получили совершенно другой результат: отработал код этой функции, и код функции декоратора. В результате на печать была выдана не одна, а три строки.

Таким образом, для того, чтобы дополнить какую либо (базовую) функцию новыми инструкциями (не трогая ее программный код), достаточно создать функцию – декоратор, а перед именем этой (базовой) функции добавить строку с символом @, за которым следует имя функции декоратора. В принципе все достаточно просто.

Функция декоратор может выполнять и еще одну важную роль - в ней можно поместить программный код, который необходимо повторять сразу в нескольких уже существующих функциях. Пример такой программы приведен в листинге 4.27.

Листинг 4.27. Функции декораторы

```
Листинг 4.27
# Это функция декоратор
def fun_decor(par_base_func):  # Сюда принимаем базовую функцию
    def wrap_func():  # это фунция - обертка для принятой функции
        print("Это отработал код, добавленный перед базовой функцией")
        par_base_func()  # Здесь выполняется код базовой функции
        print("Это отработал код, добавленный после базовой функции")
    return wrap_func  # Возвращается результат работы двух функций

@fun_decor
def func_1():
```

```
    print("111111 Это инструкция из func_1 1111111")

func_1()  # Повторный вызов базовой функции
print('-' * 60)

@fun_decor
def func_2():
    print("222222 Это инструкция из func_2 22222")

func_2()  # Повторный вызов базовой функции
print('-' * 60)

@fun_decor
def func_3():
    print("333333 Это инструкция из func_3 333333")

func_3()  # Повторный вызов базовой функции
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 4», и в выпадающем списке боковой панели выбрать раздел «Листинг 4.27. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 4.25).

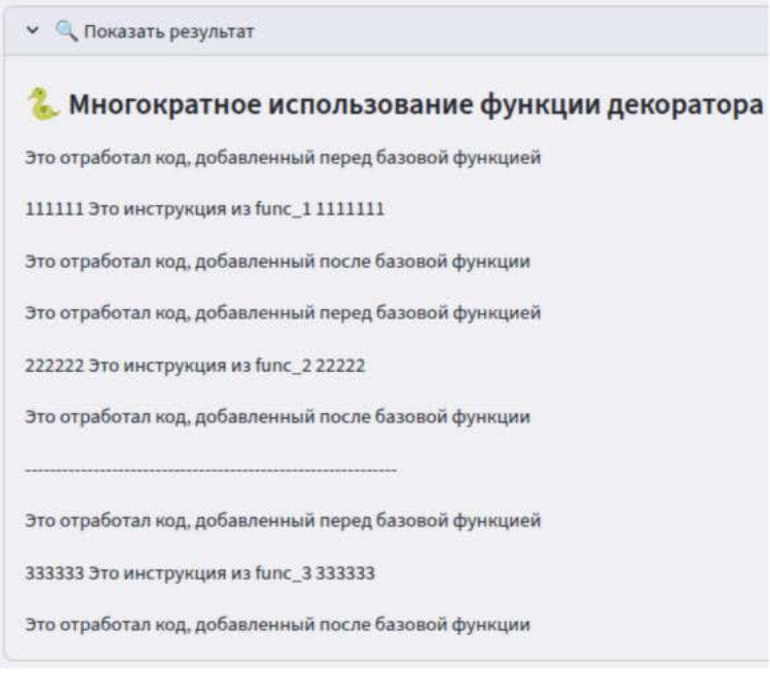


Рис. 4.25. Функции декораторы (многократное использование)

В этой программе создана функция декоратор с именем – fun_decor(). Затем мы создали три функции: func_1, func_2 и func_3. Каждая функция имеет собственный код с инструкциями выполняемых действий. Но перед именем каждой функции имеется инструкция - @fun_decor, которая говорит о том, что эти функции должны выполнять и некие дополнительные действия, которые запрограммированы в функции декораторе с именем fun_decor().

При вызове каждой из этих трех функций были выполнены содержащиеся в них инструкции, и в дополнение к ним инструкции функции декоратора.

Кроме функций декораторов, которые может создавать пользователь, Python имеет и ряд встроенных функций декораторов. Они применяются и для обычных функций, и для функций, созданных в классах. Однако на данном этапе мы не будем углубляться в эти тонкости.

4.11. Рекомендации при проектировании функций

Теперь, когда мы получили достаточно ясное представление о функциях в Python, хотелось бы подытожить этот раздел несколькими полезными рекомендациями.

Каждая функция должна иметь одно единственное назначение и решать только одну простую задачу, и при этом иметь относительно небольшой размер. Если разработанная функция начинает занимать несколько экранов – это явный признак того, что ее нужно разбить на несколько более простых частей. Используйте для передачи значений функции аргументы, а для возврата результатов – инструкцию «return». Функция должна быть максимально независимой от того, что происходит за ее пределами.

Старайтесь не использовать глобальные переменные внутри тела функции. Это поможет избежать зависимости от внешних факторов и существенно облегчит отладку программных модулей.

Следите за тем, чтобы ваши функции не меняли значений, передаваемых ей аргументов (если вызывающая программа не предполагает этого). Ведь получая, например, списки или словари, функции могут их изменить, что может исказить результаты основной программы.

4.12.Краткие итоги главы 4

6. Функция – это блок программного кода, который определяется один раз и далее может быть использован многократно. Функции позволяют существенно уменьшить время и трудозатраты на разработку приложений, а также их дальнейшее сопровождение за счет следующих преимуществ:
    9. функции дают возможность разбить сложную систему на небольшие и легко управляемые части, каждая из которых может разрабатываться отдельно;
    10. функции обеспечивают многократное использование программного кода;
    11. функции уменьшают его избыточность, повышают читабельность исходного кода, упрощают его редактирование.
7. Для создания функций в Python предназначена инструкция «def», которая создает объект функции и связывает его с именем. В скобках заголовка функции через запятую перечисляются принимаемые параметры, а заголовок завершается обязательным двоеточием. После заголовка с отступом следует тело функции из одной или нескольких

инструкций. Завершается функция инструкцией «return». Инструкция «return» не является обязательной. При ее отсутствии функция будет завершать свою работу по достижении потоком управления конца тела функции. При этом она неявно будет возвращать результат в виде объекта «None».

8. При работе с функциями используются такие понятия, как параметры и аргументы. Параметры функции представляют собой локальные переменные, которым присваиваются значения в момент ее вызова. Иногда параметрам задают значения, используемые по умолчанию. Конкретные значения, которые передаются в функцию при ее вызове, называются аргументами.
9. Осуществляются вызовы функций только после их определения при помощи имени функции, за которым следуют скобки. В скобках через запятую перечисляются аргументы, передаваемы в функцию. В Python имеется ряд специальных режимов сопоставления передаваемых в функцию аргументов с принимающими параметрами функций.
10. В заголовке функции ее параметры, которые принимают аргументы, должны указываться в следующем порядке:
    - позиционные параметры (p_1, p_2, ...);
    - позиционные параметры со значениями по умолчанию (p_1=val_1, p_2=val_2, ...);
    - параметры со звездочкой для сбора дополнительных позиционных аргументов в кортеж (*args), именованные параметры (name_p_1, name_p_2, ...);
    - именованные параметры со значениями по умолчанию (name_p_1=val_1, name_p_2=val_2, ...) и параметры с двумя звездочками для сбора дополнительных именованных аргументов в словарь (**kwargs).
11. При вызове функции порядок передачи аргументов должен быть следующим:
    - 15обычные позиционные аргументы (arg_1, arg_2, ...),
    - 16позиционные аргументы в форме *args (будут распакованы интерпретатором в значения arg_1, arg_2, ...),
    - 17обычные именованные аргументы (name_arg_1=val_1, name_arg_2=val_2, ...)
    - 18именованные аргументы в форме **name_args (будут распакованы интерпретатором в пары name_arg_1=val_1, name_arg_2=val_2, ...).
12. В пределах модуля имена переменных могут находиться в трех основных областях видимости, которые определяются местом инициализации переменных:
    - Если присваивание переменной выполняется внутри инструкции «def», переменная становится локальной для этой функции. Это касается как аргументов функции, так и переменных, созданных в теле функции. Локальные переменные доступны только внутри своей функции, то есть в текущей локальной области видимости, и недоступны за ее пределами.
    - Если присваивание производится в пределах объемлющей инструкции «def» (внешней функции), переменная становится локальной для данной объемлющей функции и нелокальной для вложенной функции. В общем случае такая переменная будет доступна внутри любой вложенной функции при отсутствии в ней локальной переменной с таким же именем.

- Если присваивание производится в главном модуле (файла скрипта) за пределами всех инструкций «def», переменная становится глобальной для всего модуля. В общем случае глобальная переменная будет доступна внутри любой функции текущего модуля при отсутствии в ней локальной переменной с таким же именем.
- 7.Получить информацию о глобальных и локальных переменных можно при помощи встроенных функций globals() и locals(), которые возвращают словари со значениями соответствующих переменных, имена которых используются в словарях в качестве ключей.

13. Поиск имен интерпретатором осуществляется следующим образом: когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается сначала отыскать его в локальной области видимости, затем в локальной области любой объемлющей инструкции «def» или в выражении «lambda» и далее в глобальной области видимости. Как только первое подходящее имя будет найдено, поиск завершается. Если искомого имени обнаружено не будет, интерпретатор выведет сообщение об ошибке.
14. Для создания простых функций непосредственно внутри выражений в Python используются lambda-выражения, которые еще называют анонимными или безымянными функциями, так как они создают функции, не имеющих имени. При этом важно помнить, что в теле анонимной функции может содержаться только одно выражение и никаких инструкций.
15. В Python допускается использование рекурсивных функций, которые позволяют осуществлять вызов самих себя до выполнения особого условия, останавливающего дальнейшие рекурсии.
16. Функции в Python являются обычными объектами, у них имеются предопределенные атрибуты, а также могут добавляться, изменяться или удаляться пользовательские атрибуты. В общем случае доступ к атрибутам можно получить при помощи имени атрибута, указанного через точку после имени функции.
17. Начиная с версии Python 3.0, у объектов функций появился новый встроенный атрибут __annotations__, используемый для хранения аннотаций к параметрам функции и возвращаемому значению.
18. Помимо аннотаций в функции разрешается добавлять строки документирования, в которые можно записывать информацию, указывающую назначение функции. Такая строка должна записываться в начале тела функции в тройных кавычках. Интерпретатор автоматически присоединит ее к объекту функции. Далее созданная документация к функции станет доступна в качестве значения его предопределенного атрибута __doc__.
19. В ходе проектирования функций следует придерживаться следующих основных правил:
    - функция должна быть максимально короткой и выполнять одну простую задачу;
    - значения в функцию должны передаваться через аргументы, а приниматься параметрами;
    - результат возвращаться через инструкцию «return»;
    - глобальные переменные должны использоваться внутри функции только в случае крайней необходимости;
    - функции не должны воздействовать на изменяемые аргументы, если вызывающая программа не предполагает этого.

4.13. Вопросы для самоконтроля к главе 4



- Какой из представленных вариантов определения функции не содержит ошибок:
  - def fun(x, y): return x*y;
  - dfn fun(x, y): return x*y;
  - def fun[x, y]: return x*y;
  - def fun{x, y}: return x*y.
- Имеется функция – «def my_fun(a, b, c): return a + b + c». В какой строке программы выполнено корректное обращение к данной функции:
  9. rez = my_fun(a, b, c);
  10. rez = my_fun(a, b);
  11. rez = my_fun(b, c);
  12. rez = my_fun(a, b, c, d).
- Имеется функция – «def my_fun(a, b=3, c=4): return a + b + c». В какой строке программы выполнено корректное обращение к данной функции:
  - rez = my_fun(3);
  - rez = my_fun(1, 2, d=5);
  - rez = my_fun(a=1, 2, 5);
  - rez = my_fun(a=1, 2, c=5).
- Имеется функция – «def my_fun(x, *y): return x + sum(y)». Инструкция обращения к этой функции имеет вид rez = my_fun(3). Какое значение получит переменная rez:
  - rez = 3;
  - rez = 1;
  - rez = 2;
  - rez = 4.
- Имеется функция – «def my_fun(x, *y): return x + sum(y)». Инструкция обращения к этой функции имеет вид rez = my_fun(1, 2). Какое значение получит переменная rez:
  - rez = 3;
  - rez = 1;
  - rez = 2;
  - rez = 4.
- Имеется функция – «def my_fun(x, *y): return x + sum(y)». Инструкция обращения к этой функции имеет вид rez = my_fun(1, 2, 3). Какое значение получит переменная rez:
  - rez = 6;
  - rez = 1;
  - rez = 2;
  - rez = 5.
- Имеется функция – «def my_fun(x, *y): return x + sum(y)». Инструкция обращения к этой функции имеет вид rez = my_fun(1, 2, 3, 4). Какое значение получит переменная rez:
  - rez = 10;
  - rez = 6;
  - rez = 5;
  - rez = 4.

- Имеется функция – «def my_fun(x, y, z): return x + y + z». В какой инструкция обращения к этой функции нет ошибок:
  - ➲ rez = my_fun(*[1, 2, 3]);
  - ➲ rez = my_fun([1, 2, 3]);
  - ➲ rez = my_fun({1, 2, 3});
  - ➲ rez = my_fun[*[1, 2, 3]].
- Имеется lambda функция – lam_fun = lambda x=1, y=2: x + y. Инструкция обращения к этой функции имеет вид rez = lam_fun(). Какое значение получит переменная rez:
  - rez = 3;
  - rez = 1;
  - rez = 2;
  - rez = 0.
- Имеется lambda функция – lam_fun = lambda x=1, y=2: x + y. Инструкция обращения к этой функции имеет вид rez = lam_fun(4). Какое значение получит переменная rez:
  14. rez = 6;
  15. rez = 4;
  16. rez = 3;
  17. rez = 2.
- Имеется lambda функция – lam_fun = lambda x=1, y=2: x + y. Инструкция обращения к этой функции имеет вид rez = lam_fun(4, 5). Какое значение получит переменная rez:
  - 4 rez = 9;
  - 5 rez = 6;
  - 6 rez = 3;
  - 7 rez = 2.
- Для чего используются функции декораторы?
  5. Для того чтобы одна функция расширила действия другую функции;
  6. Для того чтобы одна функция полностью подменила собой действие другой функции;
  7. Для декорирования вывода на печать работы функции;
  8. Для повышения скорости работы функций.

Глава 5. Объектно-ориентированное программирование в Python (классы и объекты)



При разработке программного кода используется два подхода:

- процедурное программирование;
- объектно-ориентированное программирование (ООП).

Процедурное программирование — это подход создания программных модулей, при котором инструкции (строки программы) выполняются последовательно одна за другой. При этом можно создавать подпрограммы, из которых затем собирать более крупные единицы программного кода.

Объектно-ориентированное программирование (ООП) является методологией разработки программного обеспечения, в основе которой лежит понятие класса и объекта. При этом программный код формируется из совокупности объектов, созданных на основе классов. Каждый объект является экземпляром некоторого класса. При таком подходе есть возможность создать набор классов, а потом из них, как из кирпичиков, формировать программные модули.

Python — это одновременно и процедурно-ориентированный, и объектно-ориентированный язык программирования, поскольку в нем можно использовать оба подхода. При этом абсолютно все, чем манипулирует Python, это объекты. Объектами являются: числа, строки, списки, кортежи, множества, словари, функции, и даже сами классы.

Класс (от англ. class) – это шаблон кода, который используется для описания структуры и создания объектов, то есть экземпляров этого класса.

Все данные в Python то же создаются на основе классов. Например, в инструкциях a = 'abc' и b = 5' переменная «a» становится экземпляром (объектом) встроенного класса «string», а переменная «b» экземпляром (объектом) встроенного класса «int».

Если говорить совсем просто, то класс можно сравнить с чертежом, по которому создаются объекты. Достаточно создать всего лишь один класс (чертеж) и далее можно будет порождать любое количество его экземпляров, изменяя и дополняя их по мере необходимости.

Из материалов данной главы вы узнаете:

- ➲ об основных принципах объектно-ориентированного программирования;
- ➲ о классах и объектах и как их создать;
- ➲ как получить доступ к атрибутам классов и объектов;
- ➲ как использовать принципы наследования и композиции в классах;
- ➲ что такое инкапсуляция в классах.

5.1.Основные принципы объектно-ориентированного программирования

Благодаря использованию классов Python обладает всеми преимуществами абстрактного подхода в программировании. В частности ему присущи: абстракция, полиморфизм, инкапсуляция, наследование и композиция (агрегирование).

Абстракция (от англ. abstraction) - принцип ООП, согласно которому объект характеризуется свойствами, которые отличают его от всех остальных объектов и при этом четко определяют его концептуальные границы. Абстракция позволяет представить сложную структуру объекта в более простой форме. При абстрагировании отбрасывается все лишнее, и выделяются только важные черты группы объектов (рис. 5.1).

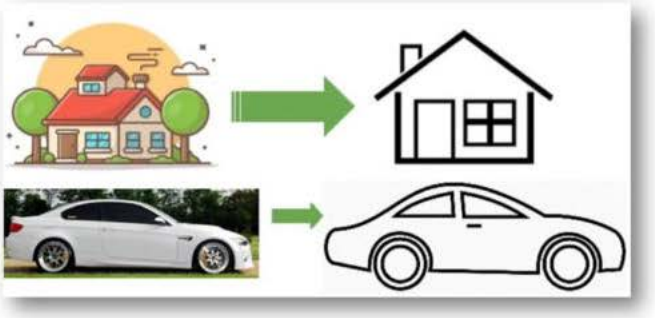


Рис. 5.1. Абстрагирование – выделение характерных свойств объектов

Полиморфизм (от англ. polymorphism) – способность функций и методов обрабатывать данные разных типов. Простыми примерами полиморфизма может служить: встроенная функция len(s), которая может работать с разными типами встроенных последовательностей, возвращая количество их элементов.

Полиморфизм позволяет перегружать одноименные методы родительского класса в классы-потомки (рис. 5.2).

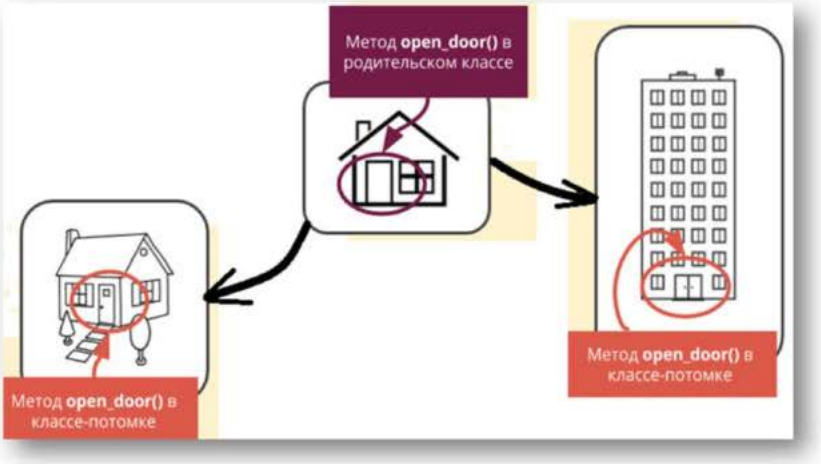


Рис. 5.2. Полиморфизм – методы класса передаются потомкам

Инкапсуляция (от англ. encapsulation) – механизм, который позволяет объединять данные и методы, работающие с этими данными, в единый объект, скрывая при этом детали реализации от пользователя. Грамотно написанный класс должен взаимодействовать с пользователем только с помощью своего интерфейса.

Инкапсуляция не дает возможности взглянуть на внутреннюю реализацию сложной структуры. При этом можно просто пользоваться сложным механизмом, не вникая, как он устроен, и как в нем протекают сложные процессы (рис. 5.3).

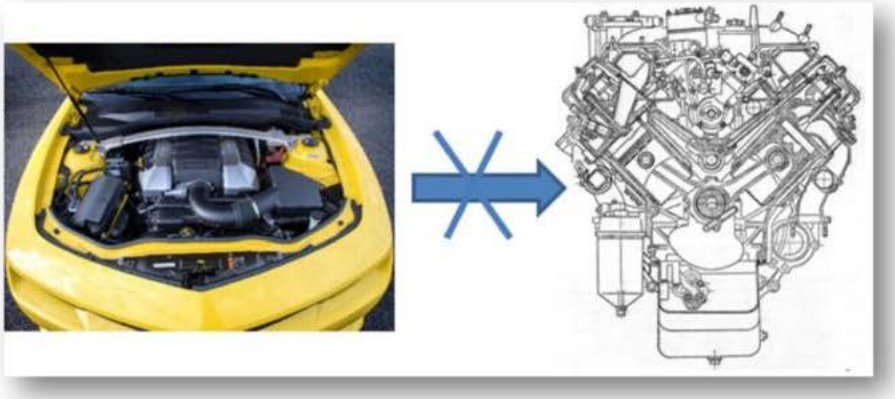


Рис. 5.3. Инкапсуляция – скрытие сложных механизмов работы класса

Наследование (от англ. inheritance) – еще одна концепция объектно-ориентированного программирования, которая позволяет на основе одного суперкласса создавать множественные подклассы, заимствующие его данные и функциональность с возможностью их изменения и добавления своих данных и функциональности.

Иными словами можно создать новый класс на основе существующего класса, при этом можно к свойствам и методам класса-родителя и добавить собственные свойства и методы (рис. 5.4).

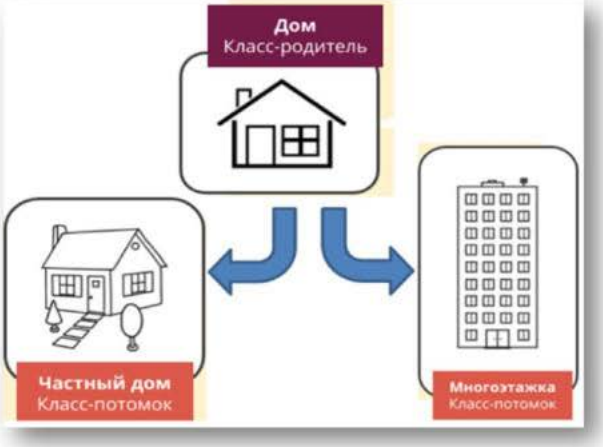


Рис. 5.4. Наследование – передача свойств и методов класса потомкам

Композиция (агрегирование) (от англ. composition) – возможность создания классов, включающих в себя вызовы уже существующих классов. В результате при создании объектов такого класса они будут состоять или содержать объекты других классов (рис. 5.5).



Рис. 5.5. Композиция – создание класса на основе существующих классов

Благодаря всем этим преимуществам ООП у программистов появляется возможность разработки отдельных независимых модулей в виде классов, способных скрывать детали своего внутреннего устройства. Данный процесс можно наглядно представить на примере автосборочного завода, когда отдельные агрегаты изготавливаются разными производителями, а затем из них собирается готовый продукт. При этом один и тот же двигатель, может быть без всяких переделок устанавливаться на десятки моделей автомобилей.

5.2. Создание классов и объектов

Для создания классов в Python используется инструкция «class», которая в общем виде может быть представлена в следующем формате:

```
class Class_name(Superclass_name_1, ..., Superclass_name_n):
      '''Строка документирования класса'''
      # Данные класса (доступны и в теле класса, и вне класса)
      class_data = value_all
      # Конструктор класса
      def __init__(self, init_arg_1, init_arg_2, ..., init_arg_n):
            # Данные, доступные только в теле класса
            self.data_class = data_value_class
            # Методы класса
            def class_method(self, arg_1, arg_2, ..., arg_n):
                  # Атрибут данных конкретного метода
                  self.data_method = value_ method
```

Заголовок инструкции начинается со служебного слова «class», затем через пробел указывается имя класса (по принятому в Python соглашению имя класса начинается с заглавной буквы). Далее в скобках перечисляются имена наследуемых классов (их еще называют суперклассами) и завершается заголовок двоеточием. Если создаваемый класс не

наследует других классов, то заголовок инструкции можно записать в формате class Class_name: (без скобок).

После заголовка с отступом можно написать блок инструкций, в которых можно создать строку «документации класса» и набор данных (переменных). Получить доступ к этим переменным можно через инструкцию «имя_класса.имя_переменной», как внутри класса, так и за его пределами.

Далее в классе формируются методы класса. Методы – это набор инструкций, в которых запрограммированы некие действия. Фактически это функции, которые находятся внутри класса и имеют один обязательный параметр «self» (собственный). Этот параметр как бы говорит интерпретатору, что данная функция является собственностью этого класса, и никто другой не имеет права ею пользоваться.

Первой функцией, или методом класса, может быть метод, который называют конструктором класса или методом инициализации класса. Конструктор класса может отсутствовать, но если он есть, он должен быть самым первых среди прочих методов. Инструкция, которая создает конструктор класса – это функция с зарезервированным именем «__init__» (здесь двойное подчеркивание), которая имеет следующий заголовок:

```
def __init__(self, par_1, par_2, par_3, …):
```

В этой инструкции:

- __init__ - зарезервированное имя конструктора класса;
- self – ключевое слово, которое говорит, что этот метод принадлежит только данному классу;
- par_1, par_2, par_3 – параметры данной функции.

У функции параметры могут быть, а могут и отсутствовать. Параметры функций называют еще свойствами класса, так как они являются некими характеристиками объектов, которыми оперирует класс. Эти свойства могут иметь (а могут и не иметь) значения по умолчанию. Если свойства класса не имеют значений по умолчанию, то эти значения должны быть обязательно заданы и переданы в класс тогда, когда на основе класса создается новый объект.

Наконец далее следуют инструкции, которые создают методы класса. В этих методах содержатся инструкции, связанные с обработкой данных. Эти данные содержаться либо в самом классе, либо передаются классу из модулей за пределами класса. Методы представляют собой обычные функции, определенные внутри класса, которые первым параметром имеют ключевое слово «self». Каждый метод, как и любая функция, может иметь свой набор параметров.

Переменные, которые определены в функции (в методе), видны только в теле этой функции. Таким образом, переменные с одинаковыми именами, используемые в разных методах никак не пересекаются и не портят друг друга. Когда необходимо в разных методах использовать значение одной и той же переменной, перед ее именем ставится префикс «self» с последующей точкой, например, self.my_name.

После того, как класс будет определен, можно приступать к созданию его отдельных экземпляров (объектов класса), сохраняя их в переменных. Для создания объекта (экземпляра) на основе класса нужно указать имя класса с круглыми скобками, перечислив в скобках начальные данные для инициализации создаваемого экземпляра (свойствам класса передать значения).

После этой вводной части можно от теории перейти к практике и создать свой первый класс. В первом примере создадим некий абстрактный класс – «Автомобиль». Для начала представим эту задачу в виде схемы (рис. 5.6).



Рис. 5.6. Абстрактный класс – «Автомобиль» и его объекты

Если представит автомобиль абстрактно (в общем, как класс), то это некий объект, который имеет набор свойств и действия, которые присущи абсолютно всем автомобилям. В нашем примере такими свойствами автомобилей могут быть:

- марка (модель)
- вес;
- мощность двигателя;
- цвет.

Так же автомобили могут выполнять какие-то действия, характерные для всего класса. Эти действия есть методы класса, и выглядеть они могут вот так:

- двигаться прямо;
- повернуть направо;
- повернуть налево;
- остановиться;
- подать звуковой сигнал и т.п.

Начинка класса готова. Если подытожить, то класса в реальности не существует, это всего лишь описание неких общих параметров, которые присущи группе объектов.

Теперь можно переходить к понятию объектов класса. Каждый конкретный автомобиль, который существует в реальности, имеет свой, индивидуальный набор свойства (конкретную марку, вес, мощность двигателя, определенный цвет). При этом абсолютно все автомобили могут выполнять одни и те же действия (двигаться, поворачивать, останавливаться и т.п.).

Завод изготовитель, имея описание класса (перечень агрегатов, их чертежи, указания последовательности сборки), может теперь делать конкретные автомобили, или, на основе описания класса, создавать новые объекты. При этом каждый такой объект может иметь свой набор свойств (например, цвет).

Итак, перейдем к созданию своего первого класса на примере автомобилей (листинг 5.1).

Листинг 5.1. Создание класса на примере автомобилей

```
# Листинг 5.1
class Car:
    '''Автор программы Иванов А.В.'''
    class_obj = "Автомобили"  # Объекты класса
    max_speed = 90  # Максимально допустимая скорость на дорогах

    def __init__(self, brand='Ford', weight=900, power=150,
colour='Красный'):
        self.brand = brand      # Марка, модель автомобиля
        self.weight = weight    # Вес автомобиля
        self.power = power      # Мощность двигателя
        self.colour = colour    # Мощность двигателя
        self.signal = 'Би-Бип'  # Имитация звука сигнала

    def drive(self):  # Метод двигаться прямо
        print("Поехали, двигаемся прямо!")
        print("Скорость движения не более -", Car.max_speed)

    def righ(self):  # Метод повернуть на право
        print("Едем, поворачиваем руль направо!")

    def left(self):  # Метод повернуть на лево
        print("Едем, поворачиваем руль налево!")

    def brake(self):  # Метод тормозить
        print("Стоп, активируем тормоз")

    def beep(self):  # Метод подать звуковой сигналь
        print("Подан звуковой сигнал")
        print(self.signal)


print('--- Выводим сведения о классе Car ---')
print('Объекты класса->', Car.class_obj)
print(Car.__doc__)
print('-' * 55)

print('Создаем объект MyCar_1 (значения свойств по умолчанию)')
MyCar_1 = Car()
print('Бренд->', MyCar_1.brand)
print('Вес->', MyCar_1.weight, 'кг.')
print('Мощность->', MyCar_1.power, 'лс.')
print('Цвет->', MyCar_1.colour)
print('-' * 55)
```

```
print('Создаем объект MyCar_2 и меняем значения свойств')
MyCar_2 = Car('Мерседес', 1200, 250, 'Черный')
print('Бренд->', MyCar_2.brand)
print('Вес->', MyCar_2.weight, 'кг.')
print('Мощность->', MyCar_2.power, 'лс.')
print('Цвет->', MyCar_2.colour)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.1. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.7).

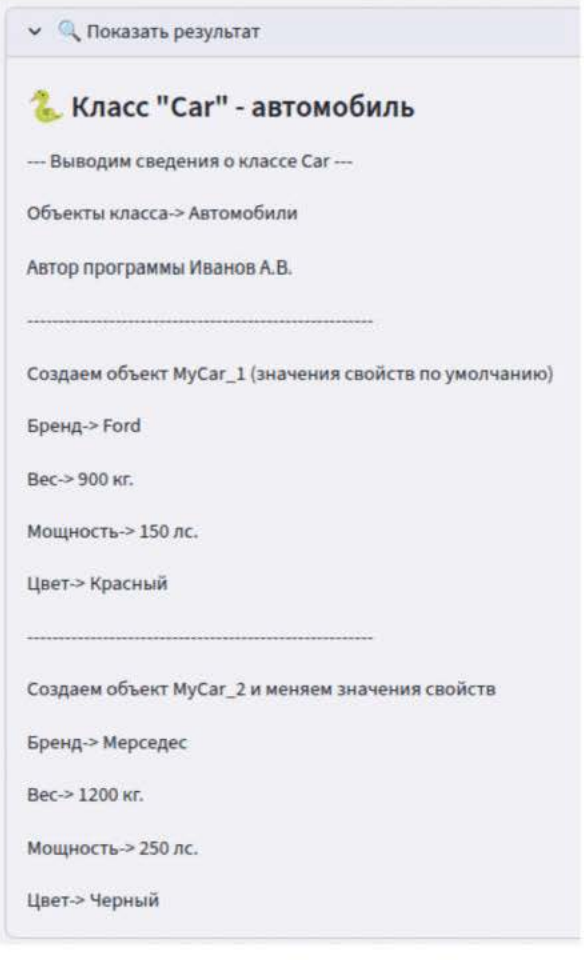


Рис. 5.7. Создание класса на примере автомобилей

В этой программе создан класс с именем – «Car». Затем, в строке документирования класса создана строка, в которую внесены сведения об авторе данного программного модуля. Эта строка не являются обязательной, она никак не влияет на работу программы и может отсутствовать. Получить доступ к данной строке вне класса можно с использованием инструкции «Car.__doc__». Далее созданы две переменные, в одну внесены сведения об объектах, которые можно создавать на основе данного класса, во вторую максимально разрешенная скорость движения автомобиля. Доступ к значениям этих переменных можно получить, как внутри класса, так и за его пределами с использованием инструкций: «Car.class_obj» и «Car.max_speed».

Затем создан метод инициализации класса, который имеет зарезервированное имя (__init__):

```
def __init__(self, brand='Ford', weight=900, power=150, colour='Красный'):
```

В этом методе перечислены параметры, через которые класс принимает внешние данные, и заданы значения этим параметрам (эти значения будут использоваться по умолчанию). Каждый такой параметр является свойством класса. Первым параметром обязательно указывается ключевое слово self. Это говорит интерпретатору о том, что данная функция (метод) является собственностью этого класса, и может использоваться только внутри данного класса. Следующим параметрам данной функции (свойствам класса) присвоены значения, которые будут использоваться по умолчанию. Это делать не обязательно, но желательно. Если все параметры (свойства) имеют значения по умолчанию, то при создании объектов на основе этого класса с пустыми аргументами ошибок не будет. Если в данной функции параметрам не заданы значения по умолчанию, то эти значения нужно обязательно передать в качестве аргумента при создании объекта на основе такого класса (иначе программа выдаст сообщение об ошибке).

В теле метода __init__ сначала создаются локальные переменные (по факту объекты), которые могут использоваться в остальных методах класса, и этим переменным присваиваются значения. Такие переменные имею префикс self. Это говорит интерпретатору о том, что эти переменные (объекты) являются локальными (собственностью этого класса), и могут использоваться только внутри данного класса. Все данные, которые класс получил извне через параметры функции __init__, должны быть перенесены в локальные переменным с префиксом self. Это будет гарантировать то, что полученные значения не будут изменены в каких либо других инструкциях вне класса. В нашем случае для использования внутри класса были созданы следующие локальные переменные (объекты):

12. self.brand = brand - марка, модель автомобиля;
13. self.weight = weight - вес автомобиля;
14. self.power = power - мощность двигателя;
15. self.colour = colour – цвет кузова;
16. self.signal = 'Би-Бип' - имитация звука сигнала.

Это не что иное, как свойства класса.

Ну и наконец, далее следуют методы класса. Это уже известные нам обычные функции, которые могут иметь свой перечень параметров. Единственным отличительным признаком этих функций (методов) является наличие на первом месте параметра self. Это говорит

интерпретатору о том, что эти функции (методы) являются собственностью этого класса, и их нельзя путать с функциями вне класса, которые имеют такие же имена.

В данном примере после того, как класс был создан, написано несколько инструкций, демонстрирующих элементарные способы создания объектов на основе класса. Сначала мы получили некоторые сведения о классе Car с использованием встроенных в класс методов, для этого используется синтаксис: <имя_класса.свойство_класса>.

Затем на основе класса Car было создано два объекта:

- MyCar_1 - со свойствами объекта по умолчанию;
- MyCar_2 - с пользовательским набором свойств.

В заключение мы получили некоторые сведения о свойствах этих объектов с использованием синтаксиса: <имя_объекта.свойство_объекта>.

Как видно из данного примера, было создано два объекта с совершенно разными характеристиками на основе одного класса. Таких объектов может быть создано сколько угодно, они будут иметь совершенно разные свойства, но обладают способностью выполнять одни и те же действия. А эти действия запрограммированы в методах класса.

Пример программы, которая демонстрирует использование методов класса на примере автомобилей, приведен в листинге 5.2.

Листинг 5.2. Создание класса на примере автомобилей

```
# Листинг 5.2
class Car:
    '''Автор программы Иванов А.В.'''
    class_obj = "Автомобили"  # Объекты класса
    max_speed = 90  # Максимально допустимая скорость на дорогах

    def __init__(self, brand='Ford', weight=900, power=150,
colour='Красный'):
        self.brand = brand      # Марка, модель автомобиля
        self.weight = weight    # Вес автомобиля
        self.power = power      # Мощность двигателя
        self.colour = colour    # Мощность двигателя
        self.signal = 'Би-Бип'  # Имитация зука сигнала

    def drive(self):  # Метод двигаться прямо
        print("Поехали, двигаемся прямо!")
        print("Скорость движения не более -", Car.max_speed)

    def righ(self):  # Метод повернуть на право
        print("Едем, поворачиваем руль направо!")

    def left(self):  # Метод повернуть на лево
        print("Едем, поворачиваем руль налево!")

    def brake(self):  # Метод тормозить
        print("Стоп, активируем тормоз")
```

```
    def beep(self):  # Метод подать звуковой сигналь
        print("Подан звуковой сигнал")
        print(self.signal)

def drive_car(MyCar):
    print('----- Поехал автомобиль', MyCar.brand, '-----')
    MyCar.drive()  # Двигается прямо
    MyCar.righ()  # Поворачиваем направо
    MyCar.drive()  # Двигается прямо
    MyCar.left()  # Поворачиваем налево
    MyCar.drive()  # Двигается прямо
    MyCar.beep()  # Подаем звуковой сигнал
    MyCar.brake()  # Тормозим
    print('-' * 50)
    return

MyCar_1 = Car()
print('Создан автомобиль', MyCar_1.brand, 'цвет-', MyCar_1.colour)
drive_car(MyCar_1)

MyCar_2 = Car('Мерседес', 1200, 250, 'Черный')
print('Создан автомобиль', MyCar_2.brand, 'цвет-', MyCar_2.colour)
drive_car(MyCar_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.2. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Создан автомобиль Ford цвет- Красный
----- Поехал автомобиль Ford -----
Поехали, двигаемся прямо!
Скорость движения не более - 90
Едем, поворачиваем руль направо!
Поехали, двигаемся прямо!
Скорость движения не более - 90
Едем, поворачиваем руль налево!
Поехали, двигаемся прямо!
Скорость движения не более - 90
Подан звуковой сигнал
Би-Бип
Стоп, активируем тормоз
--------------------------------------------------
Создан автомобиль Мерседес цвет- Черный
----- Поехал автомобиль Мерседес -----
```

```
Поехали, двигаемся прямо!
Скорость движения не более - 90
Едем, поворачиваем руль направо!
Поехали, двигаемся прямо!
Скорость движения не более - 90
Едем, поворачиваем руль налево!
Поехали, двигаемся прямо!
Скорость движения не более - 90
Подан звуковой сигнал
Би-Бип
Стоп, активируем тормоз
--------------------------------------------------
```

В данном примере создана функция с именем drive_car. В этой функции выполняются обращения к различным методам класса Car. Для вызова метода объекта используется синтаксис: <имя_объекта.метод_объекта>.

Затем было создано два объекта - автомобиля с совершенно разными свойствами на основе одного класса, а с использованием функции drive_car эти автомобили совершали определенные действия. Как видно из данного примера, не смотря на то, что объекты имеют абсолютно разные свойства, они способны совершать одни и те же действия, которые запрограммированы в методах класса.

5.3. Доступ к атрибутам классов и объектов

Доступ к открытым атрибутам, которые доступны вне класса (или объекта), может быть получен с помощью имени этого атрибута, указанного через точку после имени класса (или объекта) в формате «obj.attr». Можно не только получать и менять значения этих атрибутов, но так же добавлять новые атрибуты, или даже удалять существующие. Для добавления нового атрибута (или изменения значения уже существующего) используется привычная инструкция присваивания «obj.new_attr = value» или, соответственно, «obj.attr = new_value». Для удаления атрибута используется инструкцией «del obj.attr». Обратите внимание, что здесь речь идет об общедоступных атрибутах, которые находятся между инструкцией объявления класса «class» и методом инициализации класса «__init__».

Пример программы, которая демонстрирует возможности модификации общедоступных атрибутов класса, приведен в листинге 5.3.

Листинг 5.3. Демонстрация возможности модификации общедоступных атрибутов класса

```
# Листинг 5.3
class Car:
    max_speed = 90  # Максимально допустимая скорость на дорогах

    def __init__(self, brand='Ford', weight=900, power=150,
colour='Красный'):
        self.brand = brand  # Марка, модель автомобиля
        self.weight = weight  # Вес автомобиля
        self.power = power  # Мощность двигателя
```

```
        self.colour = colour  # Мощность двигателя
        self.signal = 'Би-Бип'  # Имитация зука сигнала

# Изменение общих атрибутов класса
print('Допустимая скорость в исходном классе ->', Car.max_speed)
Car.max_speed = 110  # Меняем атрибутов класса
print('Новая скорость в исходном классе ->', Car.max_speed)

MyCar = Car()  # Создаем объект
print('Допустимая скорость объекта ->', MyCar.max_speed)
MyCar.max_speed = 120  # Изменение атрибута объектов
print('Новая скорость у объекта ->', MyCar.max_speed)
print('Скорость в исходном классе не изменилась->', Car.max_speed)
print('-'*30)

# Добавление - удаление общих атрибутов класса
Car.brake_dist = 50  # Добавить атрибут классу (тормозной путь 50 м.)
print('Тормозной путь -', Car.brake_dist, 'м.')

del Car.max_speed  # Удаляем существующий атрибут класса
print('Максимальная скорость -', hasattr(Car, 'max_speed'))
```

Это программа повторяет листинг предыдущего примера, только в ней удалены строки с инструкциями методов класса, поскольку здесь демонстрируются только возможности модификации общедоступных атрибутов класса за его пределами (в основном модуле программы). Ниже приведены результаты работы этого программного модуля.

> ***Примечание.***
> В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.3. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.8).

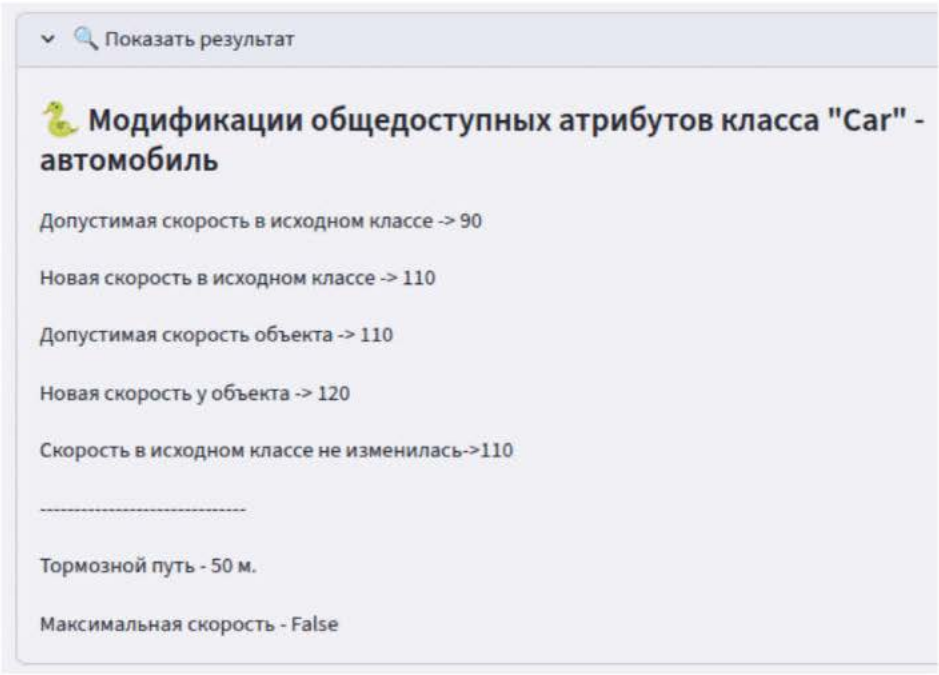


Рис. 5.8. Демонстрация возможности модификации общедоступных атрибутов класса

Здесь сначала выведена на печать максимально допустимая скорость автомобиля, которая задана в теле класса (по умолчанию она имеет значение 90 км/час). Затем этому атрибуту было задано значение – 110 км/час. Когда на основе этого класса был создан объект MyCar, то он унаследовал именно это значение допустимой скорости – 110. Далее мы поменяли значение этого атрибут у объекта MyCar со 110 на 120. Как видно из распечатки, значение этого атрибута у объекта действительно изменилось. При этом значение скорости в исходном классе не поменялось. Это говорит о том, что объект после создания, является абсолютно самостоятельным объектом и значения его атрибутов не зависимы от атрибутов базового класса.

Помимо синтаксиса с использованием точки, операции над атрибутами объектов могут выполняться и при помощи предназначенных для этого встроенных функций:

20. hasattr(object, name) – функция проверяет наличие в объекте object атрибута с именем name, возвращая True в случае успеха, и False в противном случае. В качестве аргументов ей передаются объект и строка с именем искомого в нем атрибута.
21. getattr(object, name[, default]) – возвращает значение атрибута name переданного объекта object, если он существует. В противном случае функция возвращает значение по умолчанию default, если оно было передано, или возбуждает исключение AttributeError. В качестве обязательных аргументов функции передаются объект и строка с именем искомого в нем атрибута.
22. setattr(object, name, value) – устанавливает новое значение value атрибуту с именем name переданного объекта object, если он существует. В противном случае интерпретатор создает новый атрибут с указанным именем и присваивает ему

переданное значение. В качестве первых двух аргументов функции передаются объект и строка с именем искомого в нем атрибута.

23. delattr(object, name) – удаляет из объекта object атрибут с именем name, если объект позволяет это сделать. В качестве аргументов ей передаются объект и строка с именем искомого в нем атрибута.

Пример программы с использованием этих функций приведен в листинге 5.4.

Листинг 5.4. Демонстрация использования функций работы с атрибутами объектов

```
# Листинг 5.4
class Car:
    max_speed = 90  # Максимально допустимая скорость на дорогах

    def __init__(self, brand='Ford', weight=900, power=150,
colour='Красный'):
        self.brand = brand  # Марка, модель автомобиля
        self.weight = weight  # Вес автомобиля
        self.power = power  # Мощность двигателя
        self.colour = colour  # Мощность двигателя
        self.signal = 'Би-Бип'  # Имитация зука сигнала

# Изменение общих атрибутов класса
print('Допустимая скорость в исходном классе ->', getattr(Car,
'max_speed'))

setattr(Car, 'max_speed', 110)  # Меняем атрибутов класса
print('Новая скорость в исходном классе ->', Car.max_speed)

MyCar = Car()  # Создаем объект
print('Допустимая скорость обьекта ->', getattr(MyCar, 'max_speed'))
setattr(MyCar, 'max_speed', 120)  # Изменение атрибута объектоа
print('Новая скорость у обьекта ->', getattr(MyCar, 'max_speed'))
print('Скорость в исходном классе не изменилась->', Car.max_speed)
print('-'*30)

# Добавление - удаление общих атрибутов класса
setattr(Car, 'brake_dist', 50)  # Добавить атрибут классу (тормозной
путь 50 м.)
print('Тормозной путь -', Car.brake_dist, 'м.')

delattr(Car, 'max_speed')  # Удаляем существующий атрибут класса
print('Максимальная скорость -', hasattr(Car, 'max_speed'))
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.4. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.9).

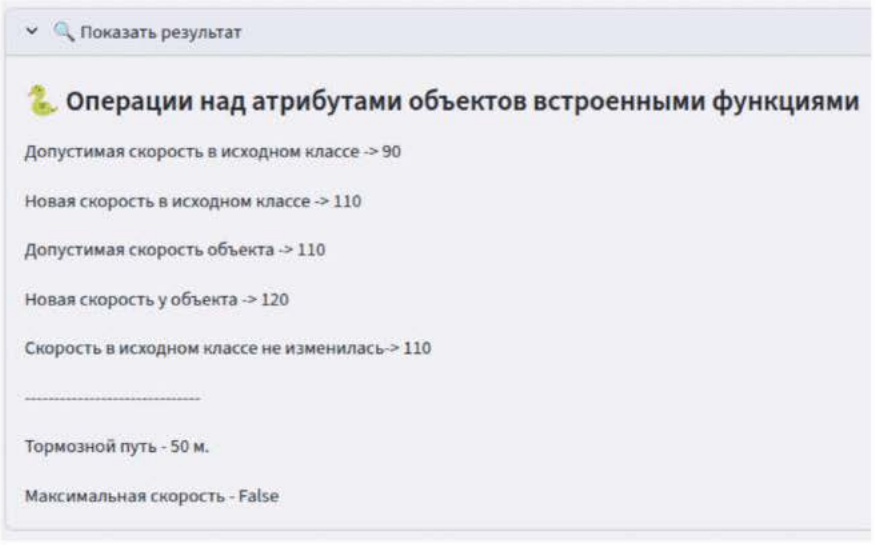


Рис. 5.9. Демонстрация использования функций работы с атрибутами объектов

Здесь важно отметить один нюанс – при обращении к атрибуту объекта, созданного на основе класса, интерпретатор производит поиск этого атрибута сначала в объекте класса, а затем, при его отсутствии, во всех классах, расположенных выше в дереве наследования. Это наглядно демонстрирует следующий пример (листинг 5.5.)

Листинг 5.5. Демонстрация использования функций работы с атрибутами объектов

```
# Листинг 5.5
# Создаем простейший класс (без методов)
class Simpl_class:
    attr_class = 0  # Атрибут класса (доступен всем экземплярам)

my_obj = Simpl_class()  # Создаем объект(экземпляр класса)
print('Начальное значение атрибута ->', my_obj.attr_class)
my_obj.attr_class = 111  # Изменяем значение атрибута у объекта
print('Измененное значение атрибута ->', my_obj.attr_class)

del my_obj.attr_class  # Удаляем атрибут из объекта (но не класса)
print("Наличие атрибута у объекта ->", hasattr(my_obj, 'attr_class'))
print('Значение атрибута объекта->', my_obj.attr_class)
print('Значение атрибута класса->', Simpl_class.attr_class)

# Здесь получим ошибку, т.к. удалять в объекте уже нечего
# del my_obj.attr_class
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел

«Листинг 5.5. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.10).

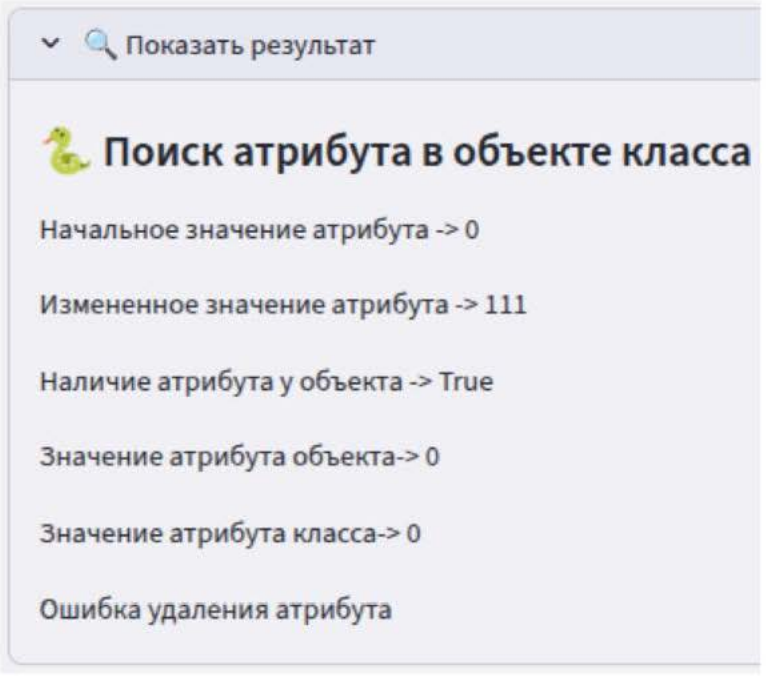


Рис. 5.10. Демонстрация использования функций работы с атрибутами объектов

Как видно из данного примера, даже при удалении атрибута «attr_class» из объекта, интерпретатор, при попытке напечатать его значение, не выдал ошибки. Дело в том, что не найдя удаленный атрибут в объекте, интерпретатор отыскал его в базовом классе, и уже оттуда вернул его значение для вывода на печать. А вот повторная попытка удалить атрибут «attr_class» из объекта уже не получится. В этом случае интерпретатор выдаст ошибку (в связи с этим данная строка в примере закомментирована).

Получить доступ и посмотреть текущий набор атрибутов класса или объекта, созданного на его основе, можно при помощи словаря атрибутов object.__dict__, который динамически изменяется в процессе изменения, как количества доступных атрибутов, так и их значений.

Пример программы использования словаря атрибутов «__dict__» для получения доступа к атрибутам класса и объектов, созданных на основе класса, приведен в листинге 5.6.

Листинг 5.6. Использование словаря «__dict__» для получения доступа к атрибутам класса и объектов

```
# Листинг 5.6
# Создаем класс
class My_Class:
    attr_1 = 1  # Атрибут данных класса

    # Метод класса
    def set_attr(self):
        self.attr_2 = 2

print('Словарь с атрибутами класса My_Class')
print(My_Class.__dict__)
print('-'*60)

My_obj = My_Class()  # Создаем объект (экземпляр класса)
print('     Изменение словарь атрибутов объекта My_obj')
print('До вызова метода set_attr:', My_obj.__dict__)

My_obj.set_attr()  # Установим атрибут объекту
print('После вызова метода set_attr:', My_obj.__dict__)

My_obj.attr_2 = 222  # Изменим значение атрибута объекта
My_obj.attr_3 = 333  # Добавим еще один атрибут объекту
print('После изменения атрибута за пределами класса', My_obj.__dict__)

My_obj.__dict__['attr_3'] = 99999  # Изменим значение атрибута через
словарь
del My_obj.__dict__['attr_2']  # Удалим атрибут через словарь
print('После изменения атрибута через словарь:', My_obj.__dict__)
print('-'*60)

print('Вывод значение атрибута объекта двумя способами')
print('Через словарь объекта ->', My_obj.__dict__['attr_3'])
print('Через имя объекта ->', My_obj.attr_3)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.6. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.11).

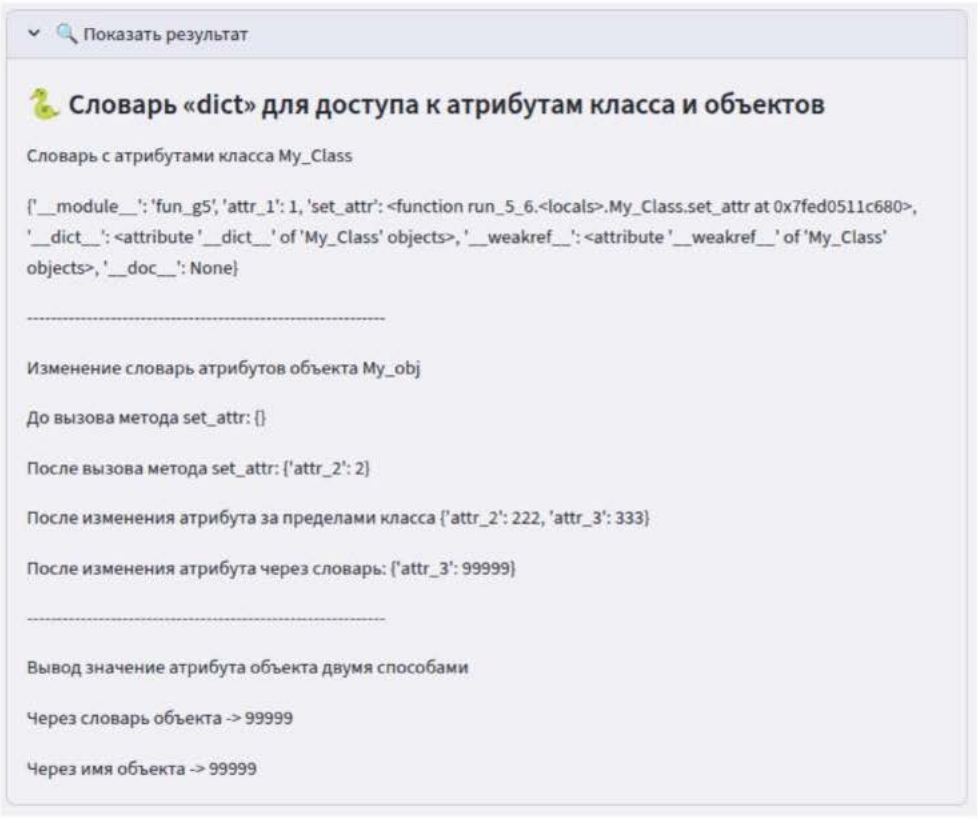


Рис. 5.11. Доступ к атрибутам классов и объектов

Поскольку в Python объекты присутствуют повсюду, то словарь атрибутов __dict__ имеется не только у классов, но и у других объектов, например, у функций.

5.4. Наследование и композиции в классах

Как упоминалось в начале данного раздела, наследование позволяет создавать новый класс, называемый подклассом, на основе уже существующего класса, называемого суперклассом. Более того, в Python поддерживается механизм композиции классов (множественного наследования), при котором подкласс может наследовать атрибуты и методы сразу нескольких суперклассов. Все что нужно, это перечислить наследуемые суперклассы в порядке их приоритета в скобках в заголовке инструкции «class».

Пример программы, в которой создан новый класс на основе двух существующих, приведен в листинге 5.7.

Листинг 5.7. Создание нового класса на основе двух существующих

```
# Листинг 5.7
# Создаем 1-й суперкласс
class Super_1:
```

```
    attr_0 = 0.001  # Атрибут данных класса
    attr_1 = 11111   # Атрибут данных класса

    def __init__(self):  # Конструктор 1-го суперкласса
        self.p_1 = '«Конструктор 1-го суперкласса»'

    def method_1(self):   # Метод 1-го суперкласса.
        print('Это инструкция метода-1 суперкласса-1')

# Создаем 2-й суперкласс
class Super_2:
    attr_0 = 0.002  # Атрибут данных класса
    attr_2 = 22222  # Атрибут данных класса

    def __init__(self):   # Конструктор 2-го суперкласса.
        self.p_2 = '«Конструктор 2-го суперкласса»'

    def method_2(self):  # Метод 2-го суперкласса
        print('Это инструкция метода-2 суперкласса-2')

# Создаем подкласс, наследуя оба суперкласса
class Sub_3(Super_1, Super_2):
    attr_0 = 0.003  # Атрибут данных класса
    attr_3 = 3333  # Атрибут данных подкласса

    def method_3(self):  # Метод подкласса
        print('Это инструкция метода-3 подкласса - 3')

# Создаем объект на основе подкласса Sub_3
obj = Sub_3()
print('------ Атрибуты obj (обьекта наследника) --------')
print('Атрибут унаследован от 1-го суперкласса (obj.attr_1) ->',
obj.attr_1)
print('Атрибут унаследован от 2-го суперкласса (obj.attr_2) ->',
obj.attr_2)
print('Значение собственного атрибута (obj.attr_3) ->', obj.attr_3)
print('Значение атрибута, с одинаковыми именами во всех классах')
print('Атрибут (obj.attr_0) ->', obj.attr_0)
print('-'*60)
obj.method_1()  # Вызов метода от 1-го суперкласса
obj.method_2()  # Вызов метода от 3-го суперкласса
obj.method_3()  # Вызов собственного метода
# Можно вызвать только один конструктор
print('Конструктор 1-го суперкласса (obj.p_1) ->', obj.p_1)
# Получим ошибку - наследуется только один конструктор
# print('obj.p_2 ->', obj.p_2)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.7. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.12).

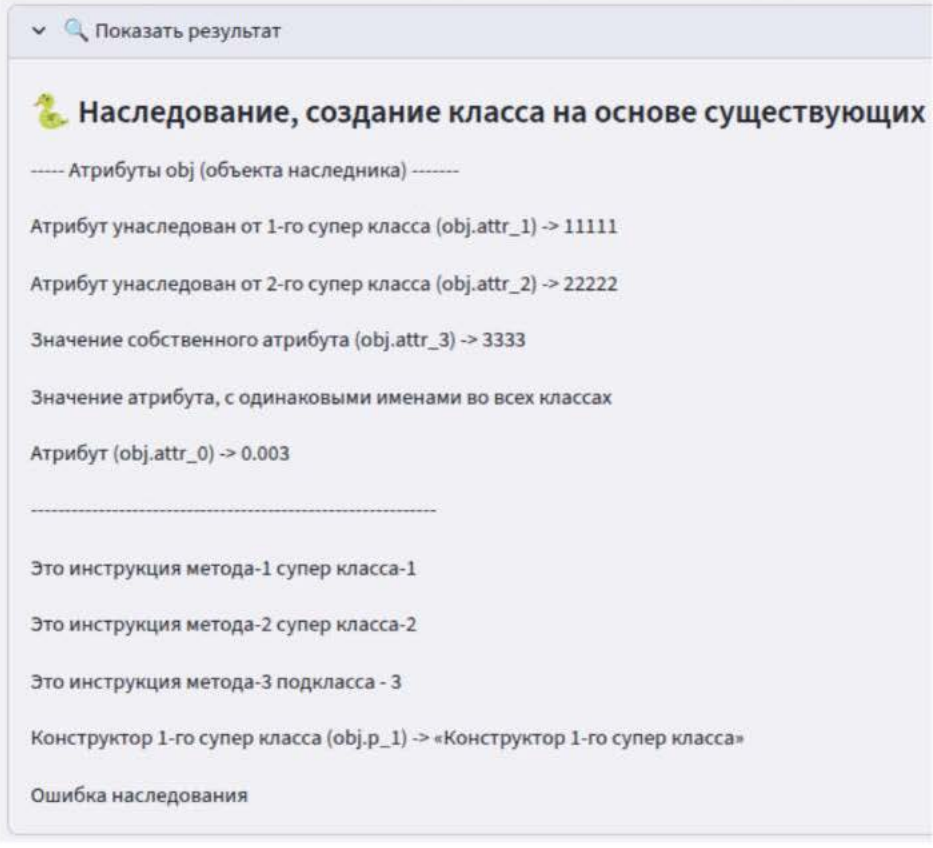


Рис. 5.12. Создание нового класса на основе двух существующих

В нашем примере подкласс Sub_3 наследует атрибуты сразу двух суперклассов Super_1 и Super_2. Поскольку все классы имеют атрибут с одинаковым именем attr_0, то интерпретатор для подкласса Sub_3 выбрал значение этого атрибута (0.003), которое определено именно в этом подклассе. Если бы подкласс не имел атрибута с именем attr_0, то его значение было унаследовано от атрибута с этим именем суперкласса Super_1. В Python при наследовании поиск атрибутов ведется сначала в самом подклассе, а затем в списке наследуемых суперклассов слева направо. Если бы нам понадобилось отдать приоритет атрибутам второго суперкласса, его пришлось бы перечислить в скобках первым в инструкции, которая создает подкласс Sub_3.

Как видно из результатов данного примера, подкласс Sub_3, не имея некоторых собственных атрибутов и методов, тем не менее, получил их в «наследство» от суперклассов и пользовался ими как своими.

Помимо того, что подклассы наследуют данные и методы своих суперклассов, они могут определять и свои собственные атрибуты. Так в подклассе Sub_3 были определены собственный атрибут данных attr_3 и метод method_3.

Но и это еще не все. При необходимости подклассы могут изменять (специализировать) наследуемые атрибуты или даже полностью замещать их.

Пример программы, в которой класс наследник меняет и дополняет атрибуты базового класса, приведен в листинге 5.8.

Листинг 5.8. Класс наследник меняет и дополняет атрибуты базового класса

```
# Листинг 5.8
# Создаем суперкласс
class Super_class:
    attr_1 = 111  # Атрибут данных класса

    def __init__(self):  # Конструктор суперкласса
        self.p_1 = '«Конструктор суперкласса»'

    def method_1(self):  # 1-й метод суперкласса
        print('«Это 1-й метод суперкласса»')

    def method_2(self):  # 2-й метод суперкласса
        print('«Это 2-й метод суперкласса»')

# Создаем подкласс - наследник суперкласса
class Sub_class(Super_class):
    attr_1 = 222  # Меняем значение атрибута в подклассе
    attr_3 = 333  # Определяем собственный атрибут данных подкласса

    def __init__(self):  # Переопределяем (расширяем) конструктор
        Super_class.__init__(self)  # Вызываем конструктор суперкласса
        self.p_2 = '«Конструктор подкласса»'  # Добавляем  свой атрибут

    def method_2(self):  # Переопределяем (расширяем) метод суперкласса
        Super_class.method_1(self)  # Вызываем второй метод суперкласса
        Super_class.method_2(self)  # Вызываем второй метод суперкласса
        print('Это расширение 2-х методов родителя в подклассе')  #
Добавляем собственную инструкци

    def method_3(self):   # Создаем собственный метод подкласса
        print('Собственный метод подкласса (method_3)')

obj = Sub_class()  # Создаем объект на основе Sub_class
print('Этот атрибут переопределен в подклассе (obj.attr_1)->',
```

```
obj.attr_1)
print('-'*60)
print('Это конструктор суперкласса (obj.p_1)->', obj.p_1)
print('-'*60)
print('Это Конструктор подкласс (obj.p_2) ->', obj.p_2)
print('-'*60)

obj.method_1()  # Метод унаследован от суперкласса
print('-'*60)

obj.method_2()  # Методы суперкласса расширен в подклассе
print('-'*60)

obj.method_3()  # Собственный метод подкласса
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.8. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.13).

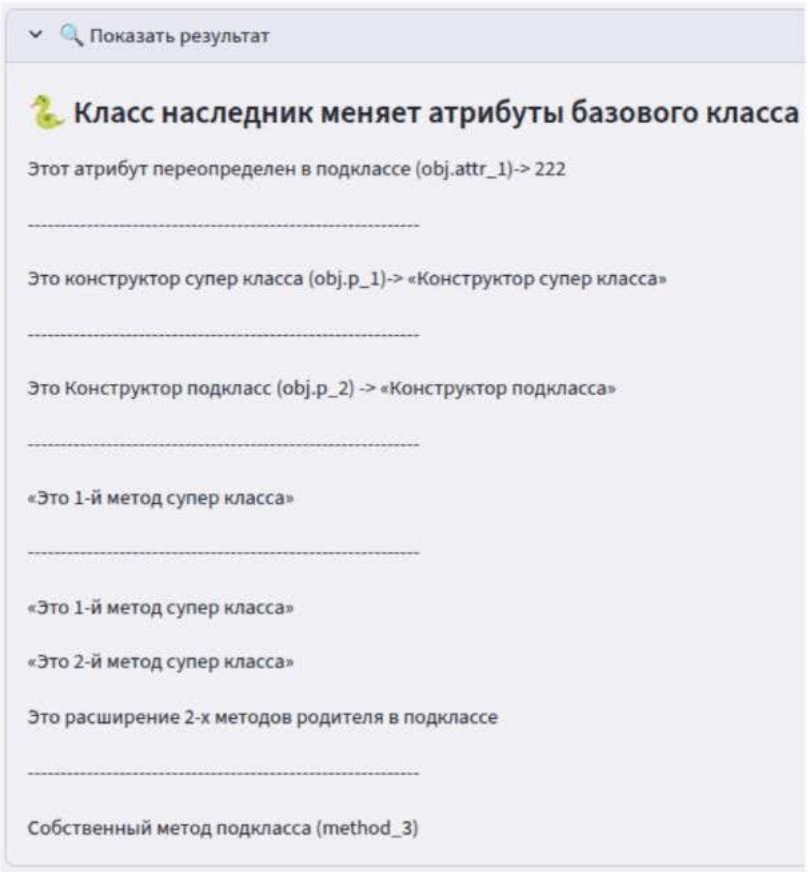


Рис. 5.13. Класс наследник меняет и дополняет атрибуты базового класса

Следует заметить, что в ходе изменения (специализации) методов в подклассах внутри замещающего метода необходим вызов исходной версии методов родительского суперкласса. Это позволяет выполнить предусмотренный в суперклассе набор действий по умолчанию (без повторного написания уже имеющегося кода суперкласса). Так в подклассе Sub_class нашего примера мы расширили его конструктор, и метод method_2.

Python позволяет создать для классов цепочки. Допустим, создана иерархия: родительский класс «A», на его основе дочерний класс «B». При этом может быть получен доступ к методам родительского и дочернего класса не только через их имена, но и через встроенную функцию super().

Функция super() – это встроенная функция языка Python. Она позволяет получить доступ из класса наследника к методам класса-родителя в том случае, если наследник переопределил эти методы. Эта функция имеет синтаксис:

```
super(type, object-or-type)
```

Параметры данной функции имеют следующий смысл:

- type - тип, от которого начинается поиск объекта-посредника (необязательный параметр);

- object-or-type - тип или объект, который определяет порядок разрешения метода для поиска (необязательный параметр).

В версиях Python 3.x можно использовать функцию super() без передачи ей двух вышеуказанных параметров. Фактически функция super(), возвращает объект (объект-посредник), который делегирует вызовы метода родительскому классу, или другому родственному классу, указанного «type» типа. Это полезно для доступа к унаследованным методам, которые были переопределены в классе.

Пример программы, в которой класс наследник меняет и дополняет атрибуты базового класса с использованием функции super(), приведен в листинге 5.9.

Листинг 5.9. Класс наследник меняет и дополняет атрибуты базового класса с использованием функции super()

```
# Листинг 5.9
# Создаем суперкласс
class A:
    def method_0():  # Метод суперкласса - не будем менять
        print('-> 0000  Это method_0 в суперклассе  A  <-')

    def method_1(self):  # 1-й метод суперкласса
        print('-> AAAA  Это method_1 в суперклассе A  <-')

    def method_2(self):  # 2-й метод суперкласса
        print('-> AAAA  Это method_2 в суперклассе A  <-')

# Подкласс - наследник суперкласса A
class B(A):
    def method_1(self):                # Расширяем 1-й метод суперкласса
A
        print('-> BBBB  Это собственный method_1 в классе B»')  # Своя
функциональность
        super().method_2()  # Используем method_2 суперкласса A (вариант
1)
        super(B, self).method_2()  # Используем method_2 суперкласса A
(вариант 2)

# Создаем объект из подкласса B
obj = B()

# Вызываем методы классов
obj.method_1()  # Сработает method_1 класса B
print('-'*45)
obj.method_2()  # Сработает method_2 класса A
print('-'*45)
super(B, obj).method_1()  # Сработает method_1 класса A
print('-'*45)
A.method_0()  # Сработает method_0 класса A
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.9. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.14).

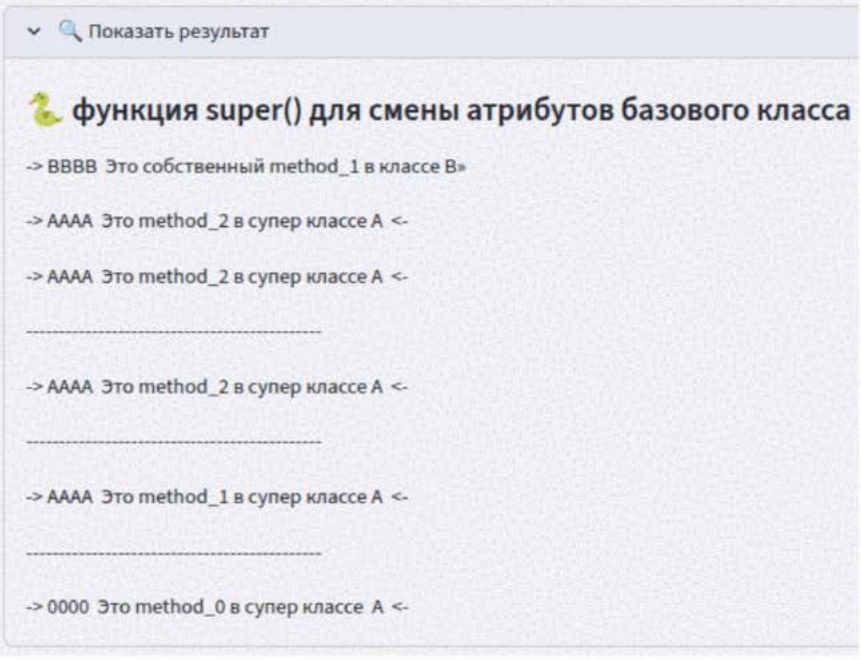


Рис. 5.14. Класс наследник меняет и дополняет атрибуты базового класса с использованием функции super()

В этой программе класс-родитель имеет метод с именем metod_1, и класс наследник имеет метод с именем metod_1. При этом класс наследник в этом методе имеет не только свой функционал (собственные инструкции), но и использует функционал метода - metod_2 класса-родителя, который включен с использованием функции super(). Причем он был включен дважды (для примера показаны разный синтаксис использования функции super). Как видно из данного примера функцию super() можно использовать не только в теле класса, но и за его пределами в основном модуле программы для вызова на исполнение методов класса. Конечно, такая конструкция делает программный код несколько запутанным и плохо читаемым, но такую возможность Python предоставляет.

Python позволяет создавать для классов цепочку наследований. Допустим, создана иерархия: класс «A», на его основе создан класс «B», а затем на основе последнего класс «C». При этом может быть получен доступ к оригинальным методам суперкласса и дочерних классов не только через его имя, но и через встроенную функцию super(). С помощью данной функции можно из дочернего класса-«внука» обратиться к методам класса-родителя и к методам класса-дедушки. Конечно, создавать такую сложную иерархию – это запутывать себя. Тем не менее, в Pyton такое возможно. Пример такой программы приведен в листинге 5.10.

Листинг 5.10. Создание для классов цепочки наследований

```
# Листинг 5.10
# Создаем суперкласс
class A:
    def method_0():  # Метод суперкласса - не будем менять
        print('-> 0000  Это method_0 в суперклассе  A  <-')

    def method_1(self):  # 1-й метод суперкласса
        print('-> AAAA  Это method_1 в суперклассе A  <-')

    def method_2(self):  # 2-й метод суперкласса
        print('-> AAAA  Это method_2 в суперклассе A  <-')

# Первый подкласс - наследник суперкласса A
class B(A):
    def method_1(self):  # Расширяем 1-й метод суперкласса A
        # Используем метод суперкласса A.
        # Можно и через super().method_1().
        super(B, self).method_1()
        print('-> BBBB  Это method_1 в классе B»')  # Своя
функциональность

# Второй подкласс - наследник суперкласса B
class C(B):
    # Расширяем 2-й метод суперкласса A.
    def method_2(self):
        super(B, self).method_2()  # Используем метод суперкласса A
        print('-> CCCC Это method_2 в классе C  <-')   # Своя
функциональность

# Создаем обьект из подкласса C
obj = C()
# Вызываем методы классов
obj.method_1()
print('-'*45)
obj.method_2()
print('-'*45)
super(B, obj).method_1()
print('-'*45)
super(B, C).method_0()
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.10. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 5.15).

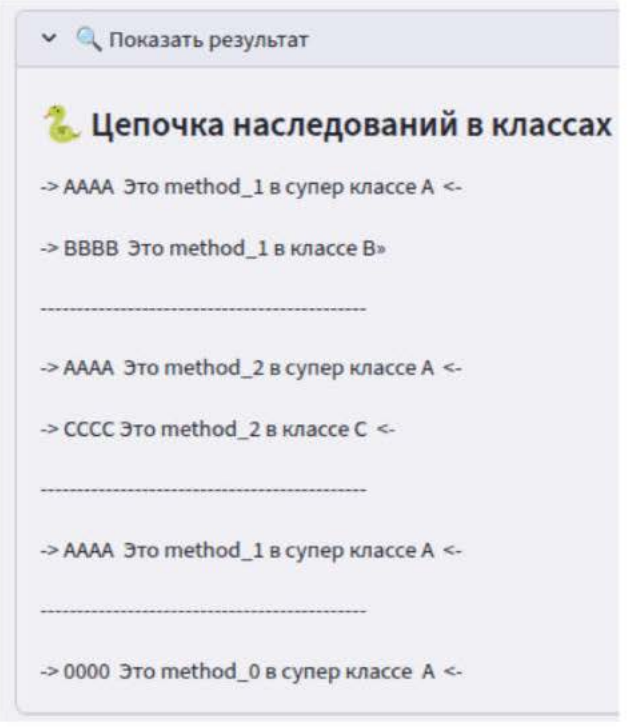


Рис. 5.15. Создание для классов цепочки наследований

Ниже приведены результаты работы этого программного модуля.

В этой программе создано три класса: класс «A», его наследник класс «B», и его наследник класс «C» (все классы родственники: «дедушка», «отец», «внук»). С использованием функции super() функционал некоторых методы классов-потомков был расширен за счет функционала классов-предков.

Как видно из данных примеров, функция super() может вызываться из подкласса без аргументов. В таком случае интерпретатор автоматически использует в качестве аргументов текущий подкласс и объект self. Кроме того, ее можно вызывать и вне классов, явно передавая требуемый тип и целевой объект.

Теперь рассмотрим подробнее ситуацию, когда подкласс наследует сразу несколько суперклассов с одинаковыми именами атрибутов. Выше мы уже указывали, что в таком случае подклассом будет унаследовано значение атрибута того суперкласса, который перечисляется в заголовке инструкции «class» первым. Но что делать, когда нужны значения атрибутов обоих классов? В этом случае, чтобы гарантировать принадлежность атрибута тому классу, который его использует, необходимо в начале имени атрибута поставить два символа подчеркивания везде, где оно используется этим классом. Программа, в которой продемонстрирована такая ситуация, приведена в листинге 5.11.

Листинг 5.11. Подкласс наследует несколько суперклассов с одинаковыми именами атрибутов

```python
# Листинг 5.11
# 1-й суперкласс
class A:
    # Устанавливаем экземпляру атрибуты
    def set_a(self):
        self.x = 111    # Обычный атрибут
        self.__y = 111  # y искажается в _A__y.

    # Получаем атрибуты
    def get_a(self):
        print(self.x)
        print(self.__y)

# 2-й суперкласс
class B:
    # Устанавливаем экземпляру атрибуты
    def set_b(self):
        self.x = 222     # Обычный атрибут
        self.__y = 222  # y искажается в _B__y

    # Получаем атрибуты
    def get_b(self):
        print(self.x)
        print(self.__y)

# Пустой класс, наследник 2-х суперклассов
class C(A, B):
    pass

# Создаем объект на основе класса C
obj = C()
print('Исходный словарь атрибутов класса "C" ')
print(obj.__dict__)
print('-'*60)
obj.set_a()  # Теперь obj.x == 1 и _A__y == 1.
print('Словарь атрибутов класса "C" после вызова метода obj.set_a()')
print(obj.__dict__)
print('-'*60)
obj.set_b()  # obj.x переопределен на 2 и _B__y == 2.
print('Словарь атрибутов класса "C" после вызова метода obj.set_b()')
print(obj.__dict__)
print('-'*60)
print('Значения атрибутов класса "C" при вызове метода obj.get_a()')
obj.get_a()
print('-'*60)
print('Значения атрибутов класса "C" при вызове метода obj.get_b()')
obj.get_b()
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.11. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Исходный словарь атрибутов класса "C"
{}
------------------------------------------------------------
Словарь атрибутов класса "C" после вызова метода obj.set_a()
{'x': 111, '_A__y': 111}
------------------------------------------------------------
Словарь атрибутов класса "C" после вызова метода obj.set_b()
{'x': 222, '_A__y': 111, '_B__y': 222}
------------------------------------------------------------
Значения атрибутов класса "C" при вызове метода obj.get_a()
222
111
------------------------------------------------------------
Значения атрибутов класса "C" при вызове метода obj.get_b()
222
222
```

В данном примере, благодаря наличию приставки из двух знаков подчеркивания, атрибуты остались доступны по короткому имени внутри класса, но стали недоступны вне класса. Это произошло потому, что имена этих атрибутов были дополнены интерпретатором именами их классов до _A__y и _B__y. Это наглядно и показал словарь атрибутов obj.__dict__ после вызова методов obj.set_a() и obj.set_b(). При этом атрибут «x», который использовался без специальной приставки, был методом obj.set_b() перезаписан.

Да, Python позволяет разными способами манипулировать объектами с одними и теми же именами в классах-родителях и в классах наследников. Однако, что не путаться в собственном программном коде, лучше его упрощать и в классах использовать уникальные имена. Это будет гарантией того, что инструкции одного класса не испортили работу инструкций другого родственного класса. Однако в тех случаях, когда требуется, чтобы родственные классы манипулировали общими данными, эти возможности Python можно использовать.

5.5. Инкапсуляция в классах

В программах на Python можно инкапсулировать (скрыть) атрибуты класса от других программных модулей. По умолчанию все атрибуты и методы класса являются открытыми или публичными (от англ. public). Это значит, что они доступны не только внутри класса, но и за его пределами. Для этого достаточно использовать синтаксис доступа к атрибутам при помощи точки (например, «obj.attr») или предназначенные для этого встроенные функции (например, «getattr(obj, 'attr')»). Однако бывают случаи, когда требуется ограничить прямой доступ к атрибутам классов из вызывающего кода во избежание

передачи им некорректных значений, случайного изменения или удаления. В таких случаях вполне логично делать атрибуты закрытыми или приватными (от англ. private). При этом следует помнить, что в Python инкапсуляция в классах (т.е. сокрытие данных) поддерживается лишь на уровне соглашения между программистами. Полностью скрыть реализацию класса от целенаправленных злонамеренных действий извне не получится.

Создать приватный атрибут довольно просто: нужно всего лишь указать в начале его имени два знака подчеркивания в формате self.__name. В результате такой манипуляции атрибут останется доступен внутри класса, но станет недоступен по такому имени за его пределами.

Пример программы, в которой продемонстрирована инкапсуляция, приведен в листинге 5.12.

Листинг 5.12. Инкапсуляция в классах

```
# Листинг 5.12
class Person:
    def __init__(self):
        self.open_atr = 'Демонстрации инкапсуляции'  # Открытый атрибут
        self.__name = None  # Имя (закрытый атрибут)
        self.__age = None   # Возраст (закрытый атрибут)

    def set_name(self, name):  # Метод - установить (принять) имя
        self.__name = name

    def set_age(self, age):  # Метод - установить (принять) возраст
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):  # Метод - получить (отдать) возраст
        return self.__age

    def get_name(self):   # Метод - получить (отдать) имя
        return self.__name

    def display_info(self):   # Метод - получить (отдать) информацию о
персоне
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")

my_per = Person()
print('Информация о человеке при инициализации класса')
my_per.display_info()
print('-'*25)

my_per.set_name('Иванов Иван Иванович')  # Передаем в класс имя
print('Информация о человеке после передачи в класс имени')
my_per.display_info()
print('-'*25)
```

```
print('Попытка передать в класс ошибочного возраста')
my_per.set_age(-3486)  # Передаем в класс недопустимый возраст
print('-'*25)

my_per.set_age(25)  # Передаем в класс возраст

print('Информация о человеке после передачи в класс возраста')
my_per.display_info()
print('-'*25)

print('Получить отдельно: имя, возраст')
print(my_per.get_name())
print(my_per.get_age())
print('-'*25)

print('Вывод открытых атрибутов класса')
print(my_per.open_atr)
# print(my_per.__name)  # Выдаст ошибку - закрытый атрибут
# print(my_per.__age)   # Выдаст ошибку - закрытый атрибут
print('='*50)

print('Но все же возможен вывод закрытых атрибутов класса')
print(my_per._Person__name)
print(my_per._Person__age)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 5», и в выпадающем списке боковой панели выбрать раздел «Листинг 5.12. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить этот модуль в редакторе Python, то будет получен следующий результат:

```
Информация о человеке при инициализации класса
Имя: None        Возраст: None
-------------------------
Информация о человеке после передачи в класс имени
Имя: Иванов Иван Иванович      Возраст: None
-------------------------
Попытка передать в класс ошибочного возраста
Недопустимый возраст
-------------------------
Информация о человеке после передачи в класс возраста
Имя: Иванов Иван Иванович      Возраст: 25
-------------------------
Получить отдельно: имя, возраст
Иванов Иван Иванович
25
```

```
-------------------------
Вывод открытых атрибутов класса
Демонстрации инкапсуляции
```

В данной программе был создан класс, в котором есть открытый атрибут (self.open_atr), который доступен из любой точки базового программного модуля. И есть два закрытых атрибута (self.__name и self.__age), которые доступны только в теле самого класса. В инструкциях класса им можно присваивать значения, получать от них значения, но обращение к данным атрибутам вне класса вызовет ошибку. В данной программе мы смогли передать значения этим атрибутам из внешнего программного модуля, и получить значения этих атрибутов. Однако это было сделано через методы, открытые внешнему «миру». И только методы в инструкциях тела класса оперировали закрытыми атрибутами. Попытка получить доступ к закрытым атрибутом в инструкциях вне класса вызовет ошибку (поэтому такие строки в данном примере закомментированы).

Тем не менее, такой подход не обеспечивает абсолютно полного сокрытия данных, так как, зная имя вмещающего (базового) класса, всегда можно обратиться к таким атрибутам по их расширенному имени из любой точки программы. Для этого используется синтаксис - obj_Class__attr. Элементы этого синтаксиса имею следующий смысл:

- obj – объект, который создан на основе базового класса (экземпляр класса);
- _Class – имя базового класса;
- __attr – скрытый атрибут базового класса.

В приведенном выше примере это было сделано с использованием следующих инструкций:

- my_per._Person__name;
- my_per._Person__age.

5.6.Краткие итоги главы 5

13. Язык программирование Python поддерживает объектно-ориентированное программирование (ООП), это позволяет создавать классы, а на основе классов объекты. Такой подход обеспечивает создание абсолютно изолированных фрагментов программ с возможностью использования функционала этих фрагментов в различных модулях и приложениях.
14. Каждый объект, созданных на основе класса, может иметь абсолютно разный набор свойств, но способен использовать все функциональные возможности, которые реализованы в методах класса.
15. Классы имеют набор атрибутов, доступ к которым можно получить из инструкций вне класса. Через эти атрибуты каждому объекту класса можно задать собственный набор свойств.
16. На основе одного класса – родителя можно создавать дочерние классы. Класс потомок наследует свойства и методы класса родителя, при этом может иметь набор собственный свойств и методов.
17. Переменные и функции, которые находятся внутри класса, имеют специальный атрибут «self» (собственный), благодаря которому значения этих атрибутов не могут быть испорчены (изменены) в инструкциях, находящихся вне класса.



5.7. Вопросы для самоконтроля к главе 5

1. Какая из инструкций, создающих класс, не имеет ошибки:

- «class My_class:»;
- «Class my_class:»;
- «clacc my_class:»;
- «klass My_class:».

2.Какая из функций является конструктором класса и не имеет ошибки:

- «def __init__(self, x, y):»;
- «def _init_(self, x, y):»;
- «def init(self, x, y):»;
- «def (__init__, self, x, y):».

3.Какая из функций является методом класса и не имеет ошибки:

- «def my_metod(self, x, y):»;
- «def __init__(my_metod, x, y):»;
- «def my_metod(x, y, self):»;
- «def my_metod(x, y):».

4.Какая из инструкций создает объект на основе класса «Car» и не имеет ошибки:

- «my_car = Car()»;
- «my_car = Car:»;
- «my_car = car()»;
- «m10 пту_car = car(my_car)».

Глава 6. Модули, пакеты и библиотеки в Python



В соответствии с модульным подходом к программированию большая задача может быть разбита на несколько более мелких задач, каждую из которых решает отдельный модуль. При этом важно составить такую композицию модулей, которая позволила бы свести к минимуму обмен данными между ними в ходе работы программы. А вот набор классов и функций, имеющих множество связей между своими элементами, было бы логично расположить в одном модуле. Есть и еще одно полезное замечание: модуль должен иметь удобный интерфейс, который предоставит доступ к его функциям и классам со стороны других модулей.

Одним из преимуществ языка Python является наличие большой библиотеки модулей и пакетов, входящих в стандартную поставку. Кроме стандартной библиотеки для Python разработано более 200 тыс. сторонних модулей. Дело в том, что Python относится к языкам с открытым исходным кодом, а это значит, что сообщество программистов имеет возможность обмениваться межу собой разработанными программами. Такие программы упаковываются в библиотеки, которыми могут свободно пользоваться программисты по всему миру. Из материалов данной главы вы узнаете:

- что такое модули, пакеты и библиотеки;
- как создать простой программный модуль;
- как организовать импорт одного программного модуля в другой модуль;
- что за функция main() и для чего она нужна;
- как импортировать стандартные модули Python и модули собственной разработки;
- как использовать библиотеки сторонних модулей.

6.1. Общие понятия о модулях, пакетах и библиотеках

Программа на Python может содержать тысячи или даже десятки тысяч строк кода, которые невозможно хранить в одном файле. В таких условиях программу разбивают на части, которые хранятся в разных файлах. С целью получения доступа из кода из одних файлов, к коду других файлов в Python реализован механизм модулей.

Модуль (от анг. module) - это отдельный файл, который содержит программный код и имеет расширение «.py». Модуль может содержать инструкции, переменные, классы и функции. Каждый модуль хранится в отдельном файле, который имеет имя и расширение, например, «may_prog.py». Здесь «may_prog» - это имя файла (оно же имя модуля), «.py» - расширение имени файла.

Каждый программный модуль может импортировать другой модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python (обычные модули), но и на других языках, например, на C или C++ (модули расширения). Итак, модульный принцип построения программ позволяет:

- разбить большую программу на функционально и логически связанные фрагменты (модули);
- модуль может импортировать другие модули;
- модуль может быть импортирован и многократно использован другими модулями.

Пример структуры проекта, состоящего из набора модулей, приведен на рис. 6.1.

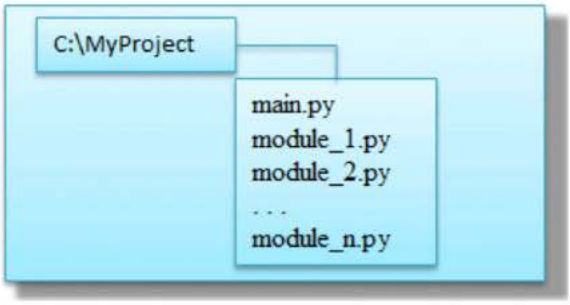


Рис. 6.1. Структура проекта из набора модулей

Двигаемся дальше. Если создавать крупный проект из большого количества модулей, то может возникнуть желание собрать несколько близких по тематике модулей в одну группу. А может возникнуть потребность вынести часть модулей из проекта, чтобы их можно было использовать в других проектах. И тут нам на помощь приходят пакеты, которые служат для объединения модулей в группы.

Пакет (от анг. packages) - это набор взаимосвязанных модулей (при этом стоит уточнить, что сам пакет тоже является модулем), предназначенных для решения задач определенного класса некоторой предметной области. Это такой способ структуризации модулей. Пакет представляет собой папку, в которой содержатся модули и другие пакеты, и файл с именем __init__.py, отвечающий за инициализацию пакета.

> ***Примечание.***
> В Python, начиная с версии 3.3, было введено неявное определения пакета. Это позволяет создать пакет без каких либо специальных файлов с именем __init__.py. Конечно, файл __init__.py все еще можно встретить во многих примерах и библиотеках, но его наличие теперь для инициализации пакета не обязательно.

Фактически пакет – это папка (каталог) с модулями, с другими пакетами и с необязательным файлом с именем __init__.py. При этом:

- именем пакета является название данного каталога (папки).
- в Python, начиная с версии 3.3, любая папка (даже без файла с именем __init__.py), считается пакетом;
- пакет может быть импортирован (так же как и модуль).
- пакет может быть многократно использован (так же как и модуль).

Пример структуры проекта, состоящего из набора модулей и пакета, приведен на рис. 6.2.

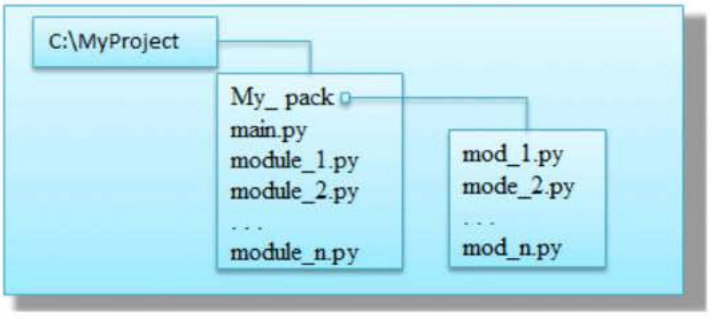


Рис. 6.2. Структура проекта из набора модулей и пакета

Теперь перейдем к библиотекам Python. Если представить обычную библиотеку, то это помещение, где храниться множество книг, доступных для большого количества читателей. Любой пользователь библиотеки может воспользоваться любой книгой и подчерпнуть из нее нужную информацию.

Если представить библиотеки Python в реальном мире, то в них находились бы полки с модулями, которые можно взять и использовать в своем коде. Именно поэтому, библиотеки Python считаются источником различного и полезного функционала. Ведь можно не тратить время на написание собственного кода, когда подобный код уже написан, проверен, отлажен и лежит в библиотеке для общего пользования. В таких библиотеках вместо книг доступны модули, которые можно просто встроить в свои программы. Библиотеки Python можно разделить на две группы: стандартные библиотеки и сторонние библиотеки.

Стандартная библиотека Python. Стандартная библиотека Python является очень обширной и включает в себя два типа модулей:

- **Встроенные модули**. Входят в состав интерпретатора и написаны на языке «С», что позволяет им обеспечивать эффективный доступ к функциональности на уровне операционной системы, например, к системе ввода-вывода данных. Многие из встроенных модулей являются зависимыми от платформы. Например, модуль «winreg», предназначенный для работы с реестром Windows, будет работать только на компьютерах под управлением операционной системы Windows.
- **Модули расширения**. Написаны на Python. Представляют собой стандартные способы решения наиболее часто встречающихся задач программирования. Сюда входят модули для работы со многими сетевыми протоколами и форматами интернета, регулярными выражениями, текстовыми кодировками, с различными форматами мультимедийных файлов, криптографическими протоколами и т.п.

Некоторые модули из стандартной библиотеки Python приведены в табл.6.1.

Таблица 6.1.

Модули стандартной библиотеки Python

Библиотека	Описание
sys	Обеспечивает доступ к некоторым переменным и функциям, взаимодействующим с интерпретатором Python, например - доступ к аргументам командной строки, списку встроенных модулей Python, текущим исключениям, информации об операционной системе

os	Предоставляет множество функций для работы с операционной системой. Например, получение версии и другой информации об операционной системе, работа с переменными окружения и файловой системой
os.path	Является вложенным в модуль os(по сути os является пакетом), и реализует некоторые полезные функции для работы с файлами - доступ к характеристикам файла и манипуляции с путями
time, datetime	Данные модули предоставляют классы для обработки времени и даты разными способами
random	Предоставляет функции для генерации случайных чисел, букв, случайного выбора элементов последовательности
collections	Предоставляет специализированные типы данных - счетчики, очереди, упорядоченные словари
json	Позволяет кодировать и декодировать данные в формате JSON
unittest	Предоставляет богатый набор инструментов для написания и запуска автоматизированных тестов

Библиотека сторонних модулей Python. Это модули и пакеты, которые не входят в дистрибутив Python и могут быть установлены самостоятельно (при необходимости) из каталога пакетов PyPI с помощью пакетного менеджера (например, pip).

PyPI (Python Package Index) - главный каталог библиотек на Python, который содержит более 200 000 готовых к использованию пакетов программных модулей. Список некоторых популярных библиотек из каталога PyPI приведен в табл.6.2.

Таблица 6.2.

Список некоторых популярных библиотек из каталога PyPI

Библиотека	Описание
TensorFlow	Библиотека для создания нейронных сетей и их обучения
PyTorch	Библиотека с рядом различных алгоритмов машинного обучения
Keras	Библиотека для быстрого создание прототипа нейронной сети
Pillow	Библиотека для работы с изображениями.
pandas	Библиотека для анализа данных
MatPlotLib	Библиотека для визуализации данных
OpenCV	Библиотека методов компьютерного зрения для работы с изображениями
numpy	Пакет для работы с многомерными массивами. Его можно рассматривать как свободную альтернативу MATLAB.
Django	Библиотека для создания веб-приложений и сайтов, работающих с базами данных
Flask	Фреймворк для создания веб-приложений на Python

Еще в Python используется такое понятие, как приложение.

Приложение – это набор программных модулей и других данных, которые в совокупности выполняют набор запрограммированных действий. Приложение может быть простым, состоящим из одного модуля, а может иметь достаточно большое количество модулей и файлов с другими данными (изображения, текстовые файлы, базы данных, модели нейронных сетей и т.д.).

6.2. Простые программные модули

В предыдущих главах было приведено множество примеров программных модулей. Но, чтобы не возвращаться к ним, создадим простейший программный модуль, в котором есть функции пользователя (листинг 6.1).

Листинг 6.1. Простейший программный модуль с функцией пользователя

```
# Листинг 6.1
def my_fun_1(x, y):  # Определяем функцию пользователя
    z = x + y / 2 + x ** 3 - y ** 2
    print("Это работает функция - my_fyn_1")
    print('z =', z)
    print('-'*40)
    return

def my_fun_2(par_1, par_2):  # Определяем функцию пользователя
    print("Это работает функция - my_fyn_2")
    z = par_1 + par_2
    print('z =', z)
    return

# Это точка входа в программу (начало)
print("   --- Начало работы модуля - Listing_6_1 ---")
a = 2
b = 3
my_fun_1(a, b)  # Вызов функции my_fun_1

arg_1 = 'Привет '
arg_2 = 'Python!'
my_fun_2(arg_1, arg_2) # Вызов функции my_fun_2

print('  --- Конец работы модуля - Listing_6_1 ---')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.1. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис6.3).

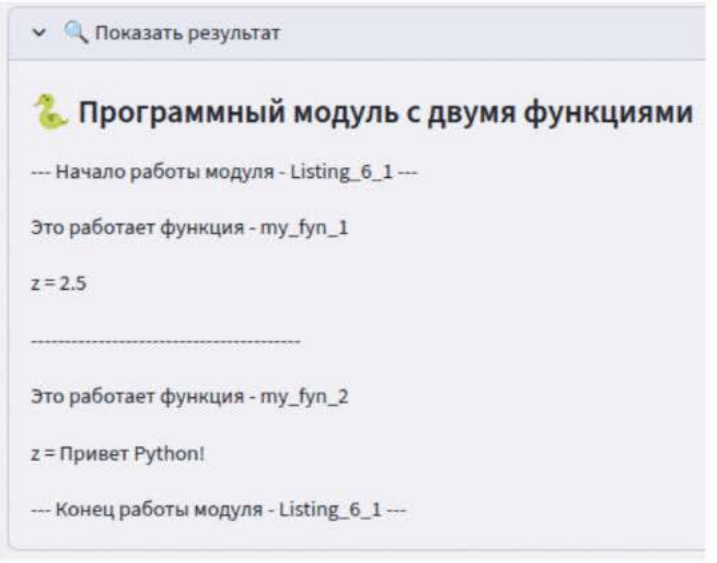


Рис. 6.3. Простейший программный модуль с функцией пользователя

В инструкциях этой программы выполняются простейшие операции:

- определено две функции пользователя «my_fun_1» и «my_fun_2»;
- выполнено обращение к этим функциям;
- выведены результаты работы инструкций этих функций.

Мы здесь мы не просто так явно обозначили точку входа в программу (начало) и точку выхода (конец). Это нам пригодится чуть позже, когда будем изучать совместную работу нескольких модулей. Теперь проверим, можно ли воспользоваться функциями этого модуля в другом программном модуле.

6.3. Импорт модулей в Python

В начале главы было сказано, что в Python одни программные модули могут импортировать другие программные модули и получать доступ к их классам, функциям и объектам. Такая процедура выполняется с помощью инструкции «import». Эта инструкция может иметь следующие варианты синтаксиса:

- import <имя_модуля>;
- import <имя_модуля> as <новое_имя_модуля>;
- from <имя_модуля> import <фрагмент_модуля>, <фрагмент_модуля>, … .

В первом случае будет импортирован код всего модуля (функции, классы, и прочие инструкции), обращаться к этому модулю можно будет только по указанному имени.

Во втором случае так же будет импортирован код всего модуля, но ему будет присвоено другое имя, и обращаться к этому модулю можно будет уже по новому имени. Обычно это делается в тех случаях, когда имя импортируемого модуля достаточно длинное и его нужно

сократить. Например, при импорте модуля из библиотеки «pyspellcheker», это длинное имя можно заменить на «psc», и тогда буде проще писать программный код при обращении к функциям данного модуля.

В третьем случае будет импортирован только один или несколько фрагментов из указанного модуля, то есть только те функции и классы, имена которых перечислены после инструкции «import».

Примечание.
Обратите внимание, что файлы, в которых хранится программный код модулей, имеют расширение «.py», но когда выполняется импорт этого модуля, то расширение не указывается.

Теперь создадим еще один программный модуль и в нем выполним импорт модуля из файла с именем Listing_6_1.py, его код приведен в листинге 6.2 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.2. Импорт модуля из файла с именем Listing_6_1.py

```
# Листинг 6.2
import Listing_6_1
```

В данном модуле был выполнен только импорт модуля из файла с именем Listing_6_1.py и нет ни одной инструкции для выполнения действий. Однако если активировать выполнение данного модуля, то он будет и импортирован, и выполнен.

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.2. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.4).

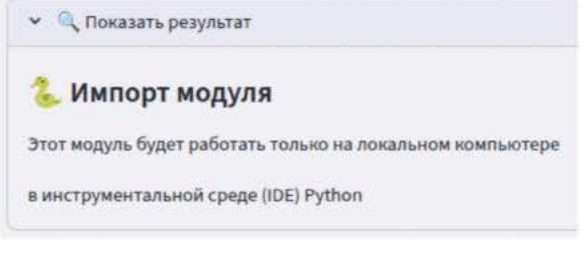


Рис. 6.4. Импорт модуля из файла с именем Listing_6_1.py

При запуске данного модуля в инструментальной среде Pytnon будет получен следующий результат:

```
  --- Начало работы модуля - Listing_6_1 ---
Это работает функция - my_fyn_1
z = 2.5
----------------------------------------
Это работает функция - my_fyn_2
```

```
z = Привет Python!
 --- Конец работы модуля - Listing_6_1 ---
```

В чем же здесь дело, почему отработали все инструкции модуля Listing_6_1? Ведь была дана команда только импорта этого модуля? Попробуем в этом разобраться. Дело в том, что интерпретатор Python (по умолчанию) начинает выполнение программного кода модуля с первой строки. Первой строкой в текущем модуле стоит инструкция «import». Эта инструкция на свое место вставила код из другого модуля - Listing_6_1. В свою очередь в модуле Listing_6_1 первой выполняемой строкой является инструкция:

```
print("  --- Начало работы модуля - Listing_6_1 ---")
```

С этой инструкции и началась работа модуля, который был импортирован. Таким образом, после импорта загруженный модуль сразу начал свою работу, однако в большинстве случаев этого делать не нужно. Как правило, требуется просто импортировать с десяток «спящих» модулей, и запускать тот или иной модуль по мере необходимости.

Если предполагается, что элементы текущего модуля (функции и классы) буду использоваться другими модулями (импортированы), то такие элементы должны быть определены в начальных строках модуля, и отделяться от базовой части модуля инструкцией «if» со следующим синтаксисом:

```
<инструкции функций и классов>)
if __name__ == '__main__':
   <инструкции текущего модуля>)
```

Если в модуле присутствует такая инструкция «if», то все строки, которые находятся в блоке «if», не будут выполняться при импорте такого модуля.

Рассмотрим работу такого подхода к импорту модулей на простом примере. Используем код предыдущего модуля, добавив в него строку с инструкцией «if» - листинг 6.3 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.3. Использование инструкции if __name__ == '__main__':

```
# Листинг 6.3
def my_fun_0():  # Определяем точку входа в программу
    if __name__ == "__main__":
        print('<- Модуль был запущен самостоятельно->')
        print('Переменная __name__ имеет значение ->', __name__)
        print('=' * 45)
    else:
        print('<- Модуль был импортирован другим модулем->')
        print('Переменная __name__ имеет значение ->', __name__)
        print('=' * 45)

def my_fun_1(x, y):  # Определяем функцию пользователя
    z = x + y / 2 + x ** 3 - y ** 2
    print("Это работает функция - my_fyn_1")
    print('z =', z)
    print('-'*40)
    return
```

```
def my_fun_2(par_1, par_2):  # Определяем функцию пользователя
    print("Это работает функция - my_fyn_2")
    z = par_1 + par_2
    print('z =', z)
    return

if __name__ == "__main__":
    print("   --- Начало работы модуля - Listing_6_3 ---")
    my_fun_0()  # Вызов функции my_fun_0

    a, b = 2, 3
    my_fun_1(a, b)  # Вызов функции my_fun_1

    arg_1, arg_2 = 'Привет ', 'Python!'
    my_fun_2(arg_1, arg_2)  # Вызов функции my_fun_2

    print('  --- Конец работы модуля - Listing_6_3 ---')
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.3. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.5).

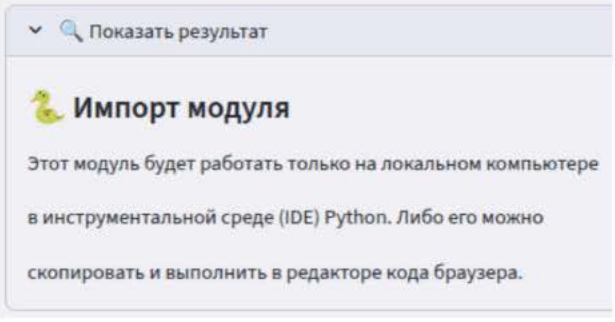


Рис. 6.5. Использование инструкции if __name__ == '__main__':

При запуске данного модуля в инструментальной среде Pytnon будет получен следующий результат:

```
  --- Начало работы модуля - Listing_6_3 ---
<- Модуль был запущен самостоятельно->
Переменная __name__ имеет значение -> __main__
==========================================
Это работает функция - my_fyn_1
z = 2.5
------------------------------------------
Это работает функция - my_fyn_2
```

```
z = Привет Python!
 --- Конец работы модуля - Listing_6_3 ---
```

Сначала разберемся, что такое специальная переменная «__name__» и объект-строка со значением «__main__», которые используются в инструкции «if». Это особые объекты Python, которые используются для выполнения строго определенных действий.

Значение переменной «__name__» интерпретатор Python воспринимает как имя среды, в которой выполняется код верхнего уровня. Код верхнего уровня - это первый пользовательский модуль Python, который начинает выполняться интерпретатором. Верхним уровнем его называют потому, что он является «главным» и импортирует все другие модули, необходимые программе. Иногда в литературе кроме термина «код верхнего уровня» можно встретить термин «точка входа в приложение».

Значение переменной __name__ позволяет интерпретатору Python определить, вызван ли модуль для исполнения, или он импортирован в другой модуль.

> ***Примечание.***
> ВАЖНО! При вызове модуля на исполнение он становится модулем верхнего уровня (точкой входа в приложение), и в его области видимости переменная __name__ ВСЕГДА принимает значение '__main__' (главный модуль приложения). При импорте модуля, он является подгружаемым модулем, и в его области видимости аналогичная переменная __name__ принимает значение имени импортируемого модуля (загружаемый модуль в главный модуль приложения).

Из приведенной выше важной информации следует, что если модуль вызван на исполнение, то переменная __name__, которая находится в области главного модуля, получит значение '__main__'. А если модуль был импортирован, то переменная __name__, которая находится в области видимости подгружаемого модуля, получит значение имени импортируемого модуля. На первый взгляд эти рассуждения кажутся запутанными, но приведенные примеры позволят распутать этот клубок.

Вернемся к приведенному выше примеру и результатам его работы. В этой программе создано три функции:

11. my_fun_0() – определяет и выводит точку входа в приложение;
12. my_fun_1() – принимает два значения, делает вычисления и выводит результат;
13. my_fun_2() – принимает два символьные строки, суммирует их и выводит результат.

После этого следует инструкция «if»;

```
if __name__ == '__main__':
```

Эта инструкция выполняет следующие действия:

- если специальная переменная __name__ будет иметь значение '__main__', то далее будут выполняться инструкции, находящиеся в блоке if;
- если специальная переменная __name__ будет иметь любое другое значение, то программа просто завершит работу.

Из результатов работы функции my_fun_0() видно, что __name__ имеет значение '__main__'. Это говорит о том, что данный модуль является главным модулем приложения, соответственно далее были выполнены все инструкции, которые находятся в блоке «if».

Теперь попробуем выполнить импорт этого модуля в другой модуль. Напишем программный код всего из одной строки - листинг 6.4 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.4. Импорт модуля

```
# Листинг 6.4
import Listing_6_3 as new_name
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.4. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.6).

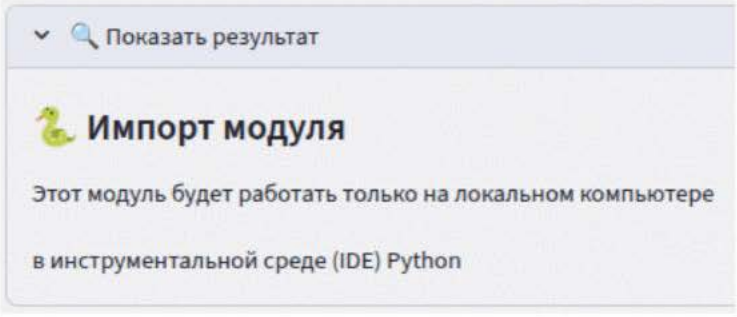


Рис. 6.6. Импорт модуля

Здесь мы просто импортировали модуль с именем Listing_6_3 и присвоили ему более короткое имя. Если мы теперь выполним этот модуль в инструментальной среде Pytnon , то **он не вернет нам никакого результата**, и при этом не получим ошибки. Что в этом случае произошло? На место этой строки был вставлен код из модуля Listing_6_3, и на самом деле этот код был запущен на исполнение. Первой исполняемой строкой в том коде является инструкция:

```
if __name__ == '__main__':
```

Поскольку этот мjдуль не является главным, а был импортирован, то переменная __name__ в области видимости переменных этого модуля, получила значение имени загруженного модуля – 'Listing_6_3'. Это строка не равна строке '__main'__, в результате чего программа просто завершила свою работу.

Все это произошло, как бы за «кулисами». Чтобы увидеть все закулисные действия напишем еще один программный модуль - листинг 6.5 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.5. Импорт модуля

```python
# Листинг 6.5
import Listing_6_3 as new_name

new_name.my_fun_0()  # Вызов функции my_fun_0
new_name.my_fun_1(2, 3)
new_name.my_fun_2("Был выполнен ", "импорт модуля!")

print('Переменная __name__ В модуле верхнего уровня ->', __name__)
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.5. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.7).

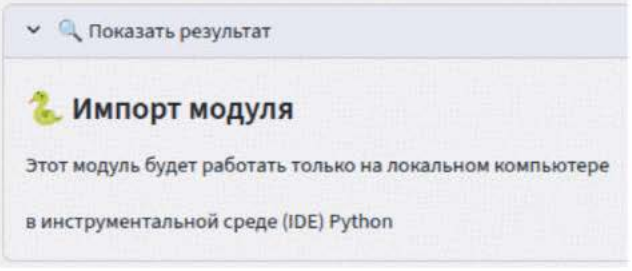


Рис. 6.7. Импорт модуля

При запуске данного модуля в инструментальной среде Pytnon будет получен следующий результат:

```
<- Модуль был импортирован другим модулем->
Переменная __name__ имеет значение -> Listing_6_3
================================================
Это работает функция - my_fyn_1
z = 2.5
----------------------------------------
Это работает функция - my_fyn_2
z = Был выполнен импорт модуля!
Переменная __name__ В модуле верхнего уровня -> __main__
```

Здесь импортировали модуль с именем Listing_6_3 и присвоили ему более короткое имя «new_name». После этого обратились ко всем трем функциям загруженного модуля. Функция my_fun_0 вернула сообщение, что импортируемый модуль имеет имя Listing_6_3, и он не работал самостоятельно, а был загружен другим модулем. Затем корректно отработали функции my_fun_1 и my_fun_2. В завершении в инструкциях главного модуля было выведено значение переменной __name__ , оно оказалась равным __main__.

Обратите внимание, что специальная переменная __name__ этих двух модулях имеет разные значения:

12. в главном модуле (точка входа в приложение) __name__ == __main__;
13. в модуле, который был импортирован, __name__ == Listing_6_3.

Эту особенность специальной переменной нужно использовать в тех случаях, когда приложение состоит из множества модулей. В этом случае выделяется один главный модуль (с которого начинается работы программы), а остальные модули импортируются по мере необходимости. При этом все импортируемые модули должны завершаться инструкцией:

```
if __name__ == '__main__':
        <имя функции или имя класса>
```

В этом случае такой модуль может быть импортирован любым другим модулем, а в блоке после инструкции «if» указать имя той функции или класса, который нужно вызывать в процессе отладки работы этого программного модуля. Пример такого модуля приведен в листинге 6.6 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.6. Использование инструкции if __name__ == '__main__':

```
# Листинг 6.6
def my_fun_1(x, y):
    return x + y

if __name__ == '__main__':
    print('rez = ', my_fun_1(1, 2))
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.6. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.8).

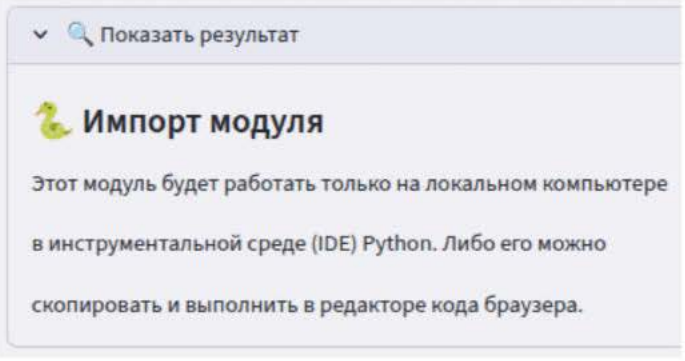


Рис. 6.8. Использование инструкции if __name__ == '__main__':

При запуске данного модуля в инструментальной среде Pytnon или в редакторе браузера будет получен следующий результат:

```
rez =  3
```

Такой модуль может быть запущен и как главный модуль в процессе его разработки (отладки), так и импортирован в любой другой модуль.

6.4. Функция main() и модуль main.py в Python

В английском языке слово main означает «главный». Во многих языках программирования ключевое слово «main» используется для обозначения точки входа в приложение. Например, все программы, написанные на языке «C», содержат основную функцию, которая должна иметь имя main(). Можно создать функцию с таким именем (или модуль) и в Python, хотя это, в отличие языка программирования «C», не является обязательным. Разберемся где и как ее можно использовать функцию main() в Python.

Если в приложение много программных модулей, или в одном модуле много функций или классов, то хорошим тоном считается выделить главный модуль, или главную функцию с именем «main». Пример программы с использованием функции main() приведен в листинге 6.7 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.7. Функция main() и модуль main.py

```
# Листинг 6.7
def my_fun_1():
    print("Работает функция my_fun_1()")
    my_fun_2()

def my_fun_2():
    print("Работает функция my_fun_2()")

def main():
    print('Запущена функция main() - точка входа в программу')
    my_fun_1()

if __name__ == "__main__":
    main()
```

Примечание.

В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.7. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.9).

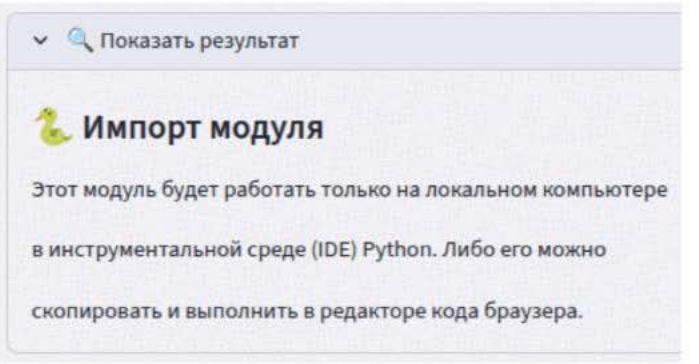


Рис. 6.9. Функция main() и модуль main.py

При запуске данного модуля в инструментальной среде Pytnon или в редакторе браузера будет получен следующий результат:

```
Запущена функция main() - точка входа в программу
Работает функция my_fun_1()
Работает функция my_fun_2()
```

В этой программе была определена точка входа в программу (функция с именем main). Она обратилась к функции my_fun_1, а та, в свою очередь, вызвала функцию my_fun_1.

По аналогии, если в программе десятки модулей, то один из них, главный, можно назвать main.py, в этом случае всем будет понятно, что этот модуль является точкой входа в приложение.

6.5. Использование библиотек сторонних модулей

В процессе разработки программного обеспечения на Python часто возникает необходимость воспользоваться пакетом (библиотекой), который в текущий момент отсутствует на вашем компьютере. В этом разделе будет рассказано о том, откуда можно взять нужную библиотеку или пакет для использования в собственных разработках. В частности:

17. где взять отсутствующий пакет;
18. как установить pip — менеджер пакетов в Python;
19. как использовать pip;
20. как установить пакет;
21. как удалить пакет;
22. как обновить пакет;
23. как получить список установленных пакетов;
24. как выполнить поиск пакета в репозитории.

Репозиторий пакетов программных средств PyPI. Необходимость в установке дополнительных пакетов возникнет достаточно часто, поскольку решение практических задач обычно выходит за рамками базового функционала, который предоставляет Python.

Это, например, создание веб-приложений, обработка изображений, распознавание объектов, нейронные сети и другие элементы искусственного интеллекта, геолокация и т. п. В таком случае, необходимо узнать, какой пакет содержит функционал, который вам необходим, найти его, скачать, разместить в нужном каталоге и начать использовать. Все указанные действия можно выполнить и вручную, однако этот процесс поддается автоматизации. К тому же скачивать пакеты с неизвестных сайтов может быть весьма опасно.

В рамках Python все эти задачи автоматизированы и решены. Существует так называемый Python Package Index (PyPI) — репозиторий, открытый для всех разработчиков на Python, в котором вы можете найти пакеты для решения практически любых задач. При этом у вас отпадает необходимость в разработке и отладке сложного программного кода — вы можете воспользоваться уже готовыми и проверенными решениями огромного сообщества программистов на Python. Вам нужно просто подключить нужный пакет или библиотеку к своему проекту и активировать уже реализованный в них функционал. В этом и заключается преимущества Python перед другими языками программирования, когда небольшим количеством программного кода можно реализовать решение достаточно сложных практических задач. В открытый для всех репозиторий можно выкладывать и свои пакеты для общего доступа. Для скачивания и установки нужных модулей в ваш проект используется специальная утилита, которая называется «pip». Сама аббревиатура, которая на русском языке звучит как «пип», фактически раскрывается как «установщик пакетов» или «предпочитаемый установщик программ». Это утилита командной строки, которая позволяет устанавливать, переустанавливать и деинсталлировать PyPI пакеты простой командой «pip».

Менеджер пакетов в Python - pip. Менеджер пакетов «pip» - это консольная утилита (без графического интерфейса). После того, как вы ее скачаете и установите на свой компьютер, она пропишется в путях PATH операционной системы и будет доступна для использования. Эту утилиту можно запускать как самостоятельно — например, через терминал в Windows или Linux, а также в терминальном окне PyCharm командой:

```
> pip <аргументы>
```

pip можно запустить и через интерпретатор Python:

```
> python -m pip <аргументы>
```

Ключ -m означает, что мы хотим запустить модуль (в нашем случае pip).

При развертывании современной версии Python (начиная с Python 2.7.9 и более поздних версий), «pip» устанавливается автоматически. У разработчиков программ на Python наиболее популярной средой программирования является IDE PyCharm, поэтому использование менеджера пакетов «pip» будет показано с использованием именно этой IDE. Как ее установить на свой компьютер будет показано в следующей главе.

В PyCharm проверить наличие модуля «pip» достаточно просто — для этого нужно войти в настройки проекта через меню File | Settings | Project Interpreter. Модуль «pip» должен присутствовать в списке загруженных пакетов и библиотек (рис. 6.10).

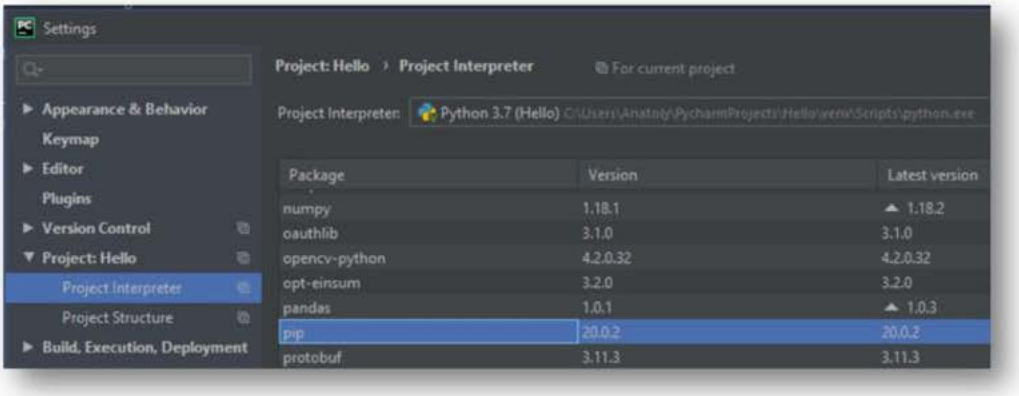


Рис. 6103. Проверка наличия в проекте модуля pip

В случае отсутствия в списке этого модуля последнюю его версию можно загрузить, нажав на значок «+» в правой части окна и выбрав модуль «pip» из списка (рис. 6.11).

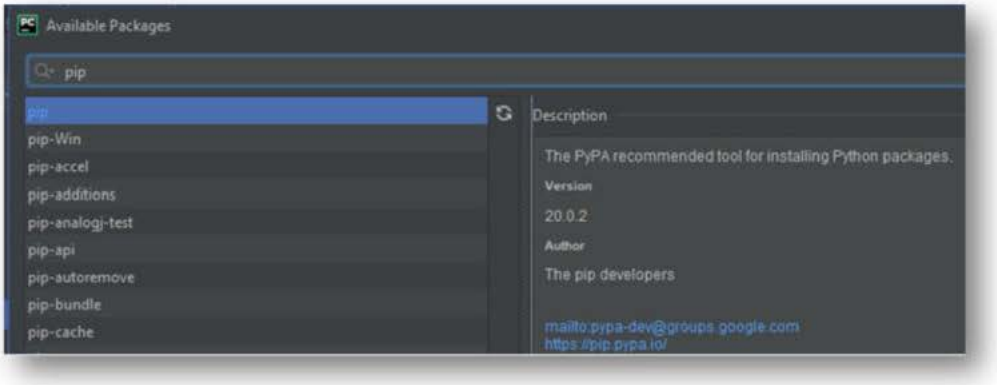


Рис. 6.11. Загрузка модуля «pip»

Использование менеджера пакетов pip. Здесь мы рассмотрим основные варианты использования pip: установку пакетов, удаление и обновление пакетов.

Менеджер пакетов «pip» позволяет установить самую последнюю версию пакета, конкретную версию или воспользоваться логическим выражением, через которое можно определить, что вам нужна версия не ниже указанной. Также есть поддержка установки пакетов из репозитория. Рассмотрим, как использовать эти варианты (здесь Name — это имя пакета).

- Установка последней версии пакета:

```
pip install Name
```

- Установка определенной версии:

```
pip install Name==3.2
```

- Установка пакета с версией не ниже 3.1:

```
pip install Name>=3.1
```

- Для того чтобы удалить пакет, воспользуйтесь командой:

```
pip uninstall Name
```

- Для обновления пакета используйте ключ --upgrade:

```
pip install --upgrade Name
```

- Для вывода списка всех установленных пакетов служит команда:

```
pip list
```

- Если вы хотите получить более подробную информацию о конкретном пакете, то используйте аргумент show:

```
pip show Name
```

- Если вы не знаете точного названия пакета или хотите посмотреть на пакеты, содержащие конкретное слово, то вы можете это сделать, используя аргумент search:

```
pip search "test" .
```

Если вы запускаете «pip» в терминале Windows, то терминальное окно автоматически закроется после того, как эта утилита завершит свою работу. При этом вы просто не успеете увидеть результаты ее работы. Чтобы терминальное окно не закрывалось автоматически, команду «pip» нужно запускать в нем с ключом «/k». Например, запуск процедуры установки пакета tensorflow должен выглядеть так, как показано на рис. 6.12.

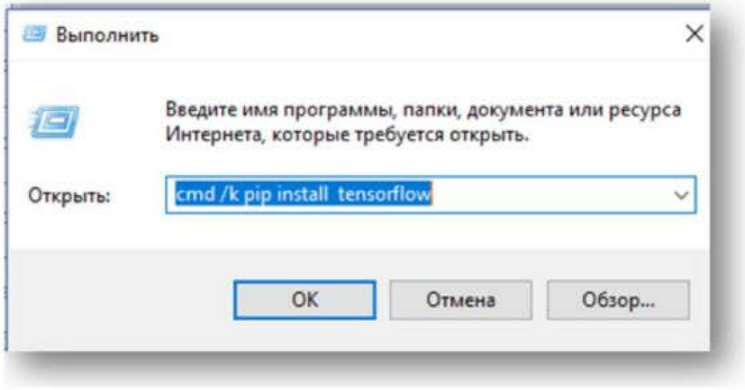


Рис. 6.12. Выполнение команды модуля «pip» в терминальном окне Windows

Если же пакет «pip» запускается из терминального окна PyCharm, то в использовании дополнительных ключей нет необходимости, так как терминальное окно после завершения работы программ не закрывается. Пример выполнения той же команды в терминальном окне PyCharm показан на рис. 6.13.

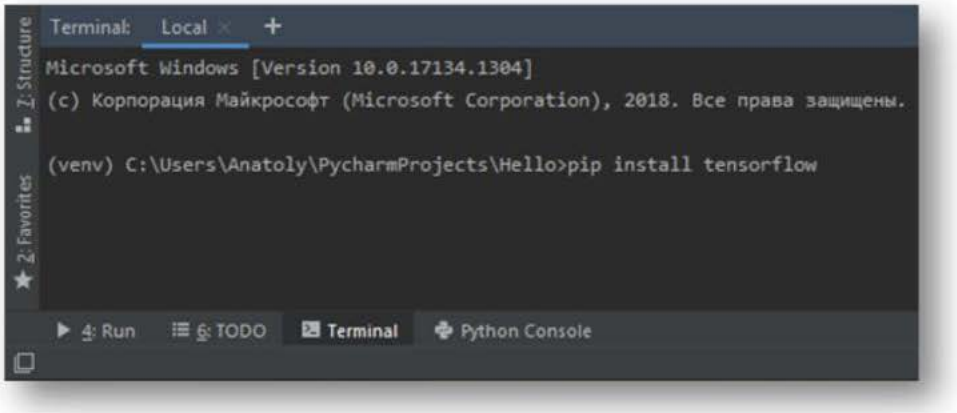


Рис. 6.13. Выполнение команды модуля «pip» в терминальном окне PyCharm

6.6. Примеры использования инструкции импорта модулей

Итак, мы знаем, что в Python можно импортировать три типа модулей:

18. стандартные модули, встроенные в Python;
19. собственные (пользовательские) модули;
20. модули сторонних разработчиков из открытых библиотек.

Рассмотрим эти три варианта на простых примерах.

Пример программы, в которой показан импорт стандартного модуля и собственного модуля, приведен в листинге 6.8 (этот модуль будет работать только в инструментальной среде Python).

Листинг 6.8. Пример использования инструкции импорта модулей

```
# Листинг 6.8
import math  # импорт модуля стандартной библиотеки Python
from my_modules import ma_class_car  # импорт собственного модуля

# Использование модуля из стандартной библиотеки
x = math.sqrt(9)
print('x =', x)
print('-'*30)

# Использование собственного модуля
# Создаем объект - Автомобиль на основе класса
# на основе класса Car из модуля ma_class_car
MyCar_1 = ma_class_car.Car()

# Выводим свойства автомобиля
```

```
print('Это автомобиль ->', MyCar_1.brand)
print('Как он будет сигналить ->', MyCar_1.signal)
print('-'*30)

# Используем методы класса Car
MyCar_1.drive()  # Начинаем движение
MyCar_1.left()   # Поворачиваем налево
MyCar_1.beep()   # Подаем звуковой сигнал
MyCar_1.righ()   # Поворачиваем направо
MyCar_1.brake()  # Тормозим
```

Примечание.
В интерактивном самоучителе для запуска данного примера нужно в строке меню «Глава 4» выбрать опцию «Листинги главы 6», и в выпадающем списке боковой панели выбрать раздел «Листинг 6.8. Результаты работы модуля можно увидеть, открыв контейнер «Показать результат». Можно также скопировать листинг в «Редактор кода» и работать с ним во встроенном редакторе.

Если выполнить данный программный код, будет получен следующий результат (рис. 6.14).

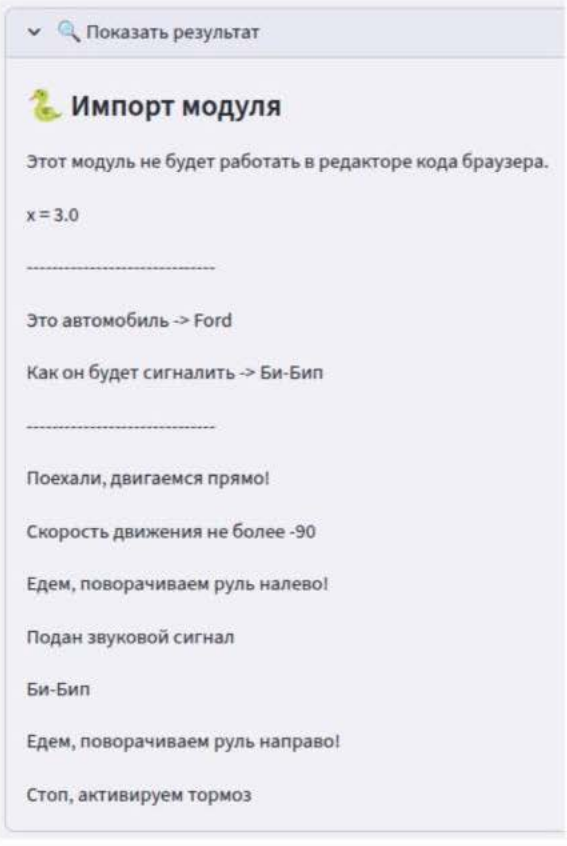


Рис. 6.14. Пример использования инструкции импорта модулей

При запуске данного модуля в инструментальной среде Pytnon будет получен аналогичный результат:

```
x = 3.0
------------------------------
Это автомобиль -> Ford
Как он будет сигналить -> Би-Бип
------------------------------
Поехали, двигаемся прямо!
Скорость движения не более - 90
Едем, поворачиваем руль налево!
Подан звуковой сигнал
Би-Бип
Едем, поворачиваем руль направо!
Стоп, активируем тормоз
```

В этой программе был выполнен импорт стандартной библиотеки – math (математические функции), и импорт собственного программного модуля, который находится в пакете (в папке) с именем my_modules. В этой папке находится модуль с именем my_class_car.py, содержащий класс – Car (автомобиль).

На первом шаге мы воспользовались функцией извлечения квадратного корня из модуля math (передали в функцию sqrt() этого модуля число – 9, в ответ получили число – 3).

На втором шаге мы воспользовались свойствами и методам класса – «Car», который был нами ранее создан и сохранен в модуле с именем my_class_car.py. Сначала мы на основе класса «Car» создали свой объект MyCar_1 (автомобиль). Затем вывели на экран два свойства этого автомобиля, и обратились к некоторым методам базового класса (активировали движение автомобиля, подали звуковой сигнал, выполнили повороты и остановились). Как видно из данного примера в Python достаточно просто в одном модуле использовать функционал другого модуля. Ниже для наглядности показан код модуля my_class_car.py, который мы импортировали:

```
#  Модуль my_class_car.py
class Car:
    '''Автор программы Иванов А.В.'''
    class_obj = "Автомобили"  # Объекты класса
    max_speed = 90  # Максимально допустимая скорость на дорогах

    def __init__(self, brand='Ford', weight=900, power=150,
colour='Красный'):
        self.brand = brand      # Марка, модель автомобиля
        self.weight = weight    # Вес автомобиля
        self.power = power      # Мощность двигателя
        self.colour = colour    # Мощность двигателя
        self.signal = 'Би-Бип'  # Имитация звука сигнала

    def drive(self):  # Метод двигаться прямо
        print("Поехали, двигаемся прямо!")
        print("Скорость движения не более -", Car.max_speed)
```

```
    def righ(self):  # Метод повернуть на право
        print("Едем, поворачиваем руль направо!")

    def left(self):  # Метод повернуть на лево
        print("Едем, поворачиваем руль налево!")

    def brake(self):  # Метод тормозить
        print("Стоп, активируем тормоз")

    def beep(self):  # Метод подать звуковой сигналь
        print("Подан звуковой сигнал")
        print(self.signal)

if __name__ == "__main__":
    Car()
```

Как видно из этого листинга, программный код в последних строках имеет инструкцию:

```
if __name__ == "__main__":
   Car()
```

Эта инструкция, как мы уже знаем, обеспечивает возможность импорта данного модуля в другие программные средства (без запуска на выполнение в момент импорта).

Что касается импорта библиотек сторонних разработчиков, то теперь вы сможете делать это самостоятельно. Однако для этого нужно установить на свой компьютер интерпретатор Python и одну из программных оболочек (IDE) для редактирования программного кода. Именно этим вопросам и посвящена заключительная глава.

6.7.Краткие итоги главы 6

- В Python большие приложения можно разбивать на модули. При этом модули можно объединять, как в пакеты, так и в библиотеки.
- Стандартная библиотека Python имеет большой ассортимент модулей, которые можно напрямую использовать в инструкциях. Кроме того, для пользователей Python существует библиотека сторонних модулей, которыми можно пользоваться совершенно бесплатно.
- Функционал любого модуля Python можно расширить, как путем импорта собственных модулей, так и модулей из сторонних библиотек. Для этих целей используется инструкция «import».
- Любой импортируемый модуль сразу запускается на исполнение (в момент импорта). Для того чтобы этого не происходило, импортируемый модуль должен завершаться инструкцией:

  if __name__ == '__main__':

- Для того чтобы использовать функционал сторонних модулей, их сначала нужно установить на свой компьютер с использованием менеджера пакетов «pip».

6.8. Вопросы для самоконтроля к главе 6



1. Какая из инструкций, импортирует только часть функций и классов модуля:
   - from new_module import fun_1;
   - import new_module;
   - import new_module as mod_1;
   - import mod_1, mod_2.
2. Какая из инструкций не имеет ошибки:
   1. «if __name__ == '__main__':»;
   2. «if ‘__name__’ == __main__:»;
   3. «if __main__ == '__name__':»;
   4. «if ‘__main__’ == __name__':».

Глава 7. Инструментальные средства для разработки приложений на Python



Для того чтобы разрабатывать и запускать программы на Python нужно скачать соответствующий дистрибутив и установить его на свой компьютер. В дистрибутив входит и простой редактор программного кода (**IDLE** или Python Shell). Писать программы с использованием IDLE конечно можно, если речь идёт о простых модулях, однако с ростом размеров проектов программирование превращается в мучение.

Для ускорения процесса написания программного кода удобно использовать специализированную инструментальную среду **IDE** (Integrated development environment - интегрированная среда разработки). Данная среда разработки имеет полный комплект средств, необходимых для эффективного программирования на Python. Обычно IDE имеет текстовый редактор, компилятор или интерпретатор, отладчик и другое программное обеспечение. Использование IDE позволяет увеличить скорость разработки программ (при условии предварительного обучения работе с данной инструментальной средой).

Одной из лучших полнофункциональных IDE, предназначенных именно для Python, является PyCharm. Существует как бесплатная версия - open-source (Community), так и платный вариант IDE (Professional). PyCharm доступен на Windows, Mac OS X и Linux. Инструментальная среда PyCharm «из коробки» поддерживает разработку на Python напрямую — откройте новый файл и начинайте писать код. Вы можете запускать и отлаживать код прямо из PyCharm. Кроме того, в IDE есть поддержка проектов и системы управления версиями.

Из материалов данной главы вы узнаете:

- откуда можно скачать и как установить интерпретатор Python;
- откуда можно скачать и как установить IDE PyCharm.

7.1. Интерпретатор Python

Язык программирования Python является весьма мощным инструментальным средством для разработки различных систем. Однако наибольшую ценность представляет даже не столько сам этот язык программирования, сколько набор подключаемых библиотек, на уровне которых уже реализованы все необходимые процедуры и функции. Разработчику достаточно написать несколько десятков строк программного кода, чтобы подключить требуемые библиотеки, создать набор необходимых объектов, передать им исходные данные и отобразить итоговые результаты.

Для установки интерпретатора Python на компьютер, прежде всего надо загрузить его дистрибутив. Язык постоянно совершенствуется и на официальном сайте достаточно много версий, причем для разных платформ. Скачать последнюю версию дистрибутив Python можно с официального сайта, перейдя по ссылке: **https://www.python.org/downloads/** (рис. 7.1).

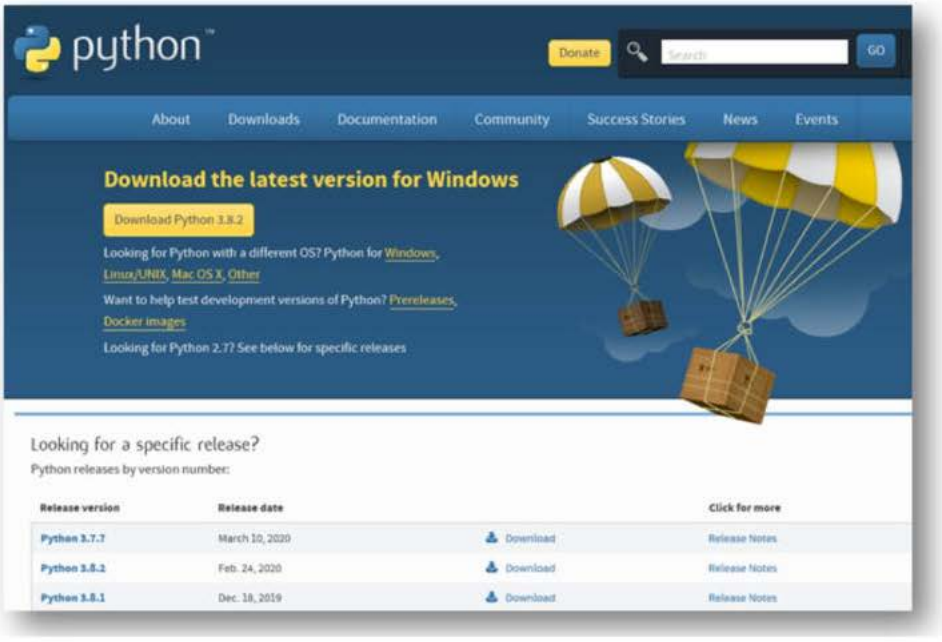


Рис. 7.1. Сайт для скачивания дистрибутива языка программирования Python

Установка Python в Windows. Для операционной системы Windows дистрибутив Python распространяется либо в виде исполняемого файла (с расширением exe), либо в виде архивного файла (с расширением zip). На момент подготовки этой книги была доступна версия Python 3.8.3. Порядок установки Python в Windows следующий: запустите скачанный установочный файл и выберите способ установки (рис. 7.2).

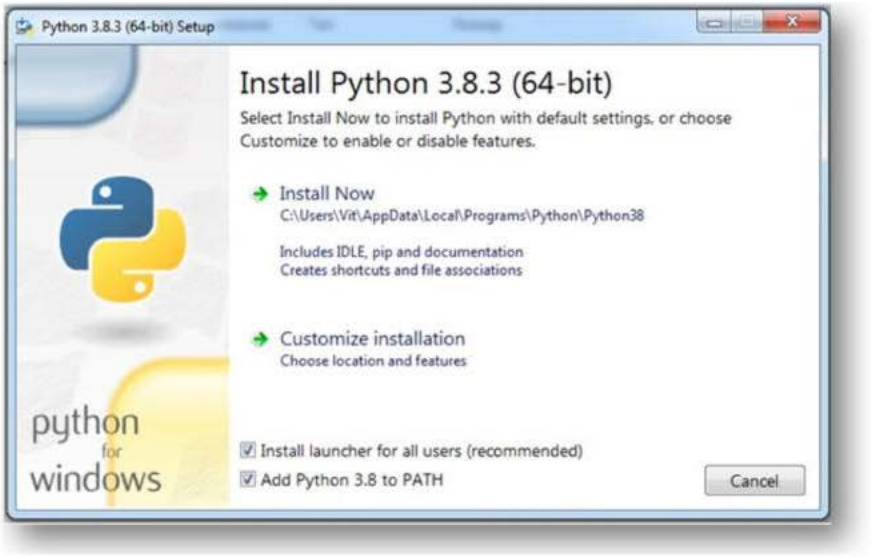


Рис. 7.2. Выбор способа установки Python

В открывшемся окне предлагаются два варианта: **Install Now** и **Customize installation**:

- при выборе **Install Now** Python установится в папку по указанному в окне пути. Помимо самого интерпретатора будут инсталлированы IDLE (интегрированная среда разработки), «pip» (пакетный менеджер) и документация, а также созданы соответствующие ярлыки и установлены связи (ассоциации) файлов, имеющих расширение «py», с интерпретатором Python;
- **Customize installation** — это вариант настраиваемой установки. Опция **Add Python 3.8 to PATH** нужна для того, чтобы появилась возможность запускать интерпретатор без указания полного пути до исполняемого файла при работе в командной строке.

Отметьте необходимые опции установки, как показано на рис. 7.3 (доступно при выборе варианта **Customize installation**).

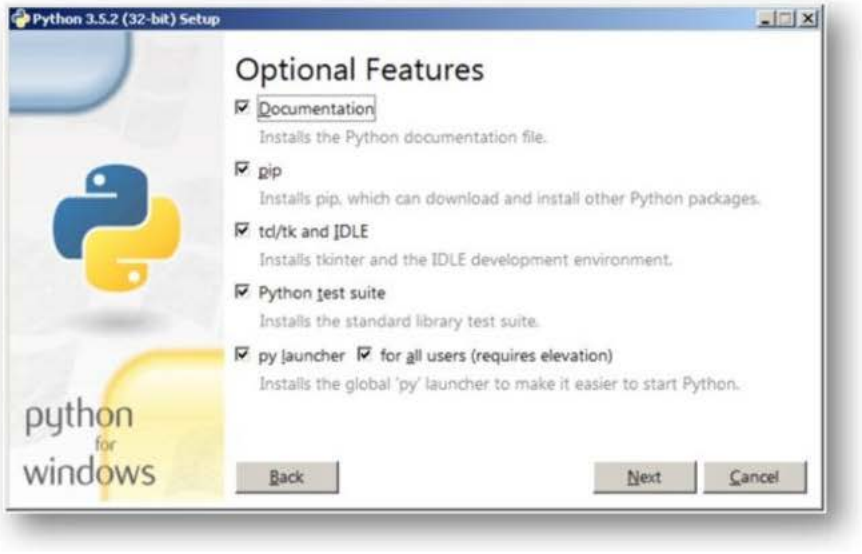


Рис. 7.3. Выбор опций установки Python

На этом шаге нам предлагается отметить дополнения, устанавливаемые вместе с интерпретатором Python. Рекомендуется выбрать как минимум следующие опции:

- **Documentation** — установка документации;
- **pip** — установка пакетного менеджера «pip»;
- **tcl/tk and IDLE** — установка интегрированной среды разработки (IDLE) и библиотеки для построения графического интерфейса (tkinter).

На следующем шаге в разделе **Advanced Options** (дополнительные опции) выберите место установки, как показано на рис. 7.4 (доступно при выборе варианта **Customize installation**).



Рис. 7.4. Выбор места установки Python

Помимо указания пути, этот раздел позволяет внести дополнительные изменения в процесс установки с помощью опций:

- **Install for all users** — установить для всех пользователей. Если не выбрать эту опцию, то будет предложен вариант инсталляции в папку пользователя, устанавливающего интерпретатор;
- **Associate files with Python** — связать файлы, имеющие расширение «.py», с Python. При выборе этой опции будут внесены изменения в Windows, позволяющие Python запускать скрипты по двойному щелчку мыши;
- **Create shortcuts for installed applications** — создать ярлыки для запуска приложений;
- **Add Python to environment variables** — добавить пути до интерпретатора Python в переменную PATH;
- **Precomple standard library** — провести перекомпиляцию стандартной библиотеки.

Последние два пункта связаны с загрузкой компонентов для отладки, их мы устанавливать не будем.

После успешной установки Python вас ждет следующее сообщение (рис. 7.5).



Рис. 7.5. Финальное сообщение после установки Python

Установка Python в Linux. Чаще всего интерпретатор Python уже входит в состав дистрибутива Linux. Это можно проверить, набрав в окне терминала команду:

```
> python
```

или

```
> python3
```

В первом случае, вы запустите Python 2, во втором — Python 3. В будущем, скорее всего, во все дистрибутивы Linux, включающие Python, будет входить только третья его версия. Если у вас при попытке запустить Python выдается сообщение о том, что он не установлен или установлен, но не тот, что вы хотите, то у вас есть возможность взять его из репозитория.

Для установки Python из репозитория Ubuntu воспользуйтесь командой:

```
> sudo apt-get install python3
```

Проверка интерпретатора Python. Для начала протестируем интерпретатор в командном режиме. Если вы работаете в Windows, то нажмите комбинацию клавиш <Win>+<R> и в открывшемся окне введите: python. В Linux откройте окно терминала и в нем введите: python3 (или python).

В результате Python запустится в командном режиме. Выглядеть это будет примерно так, как показано на рис. 7.6 (иллюстрация приведена для Windows, в Linux результат будет аналогичным).

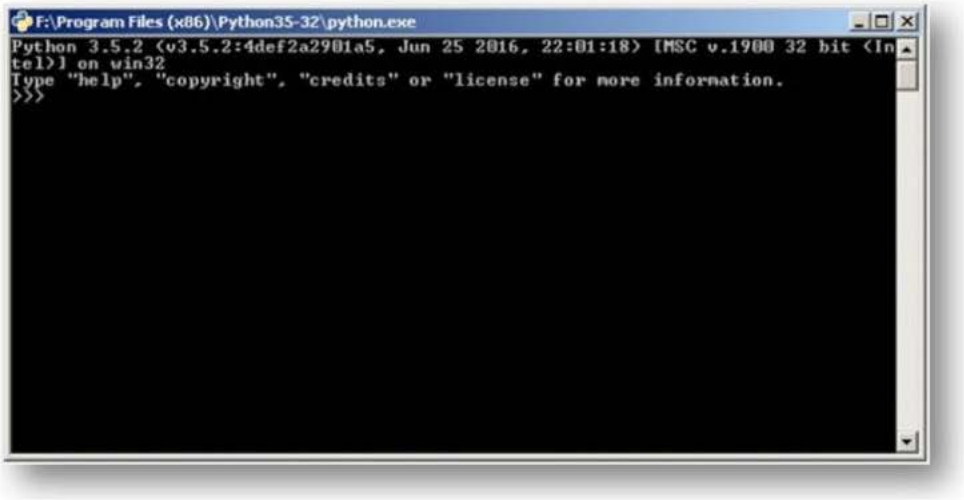


Рис. 7.6. Результат запуска интерпретатора Python в окне терминала

В этом окне введите программный код следующего содержания:

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

В результате вы увидите следующий ответ (рис. 7.7).

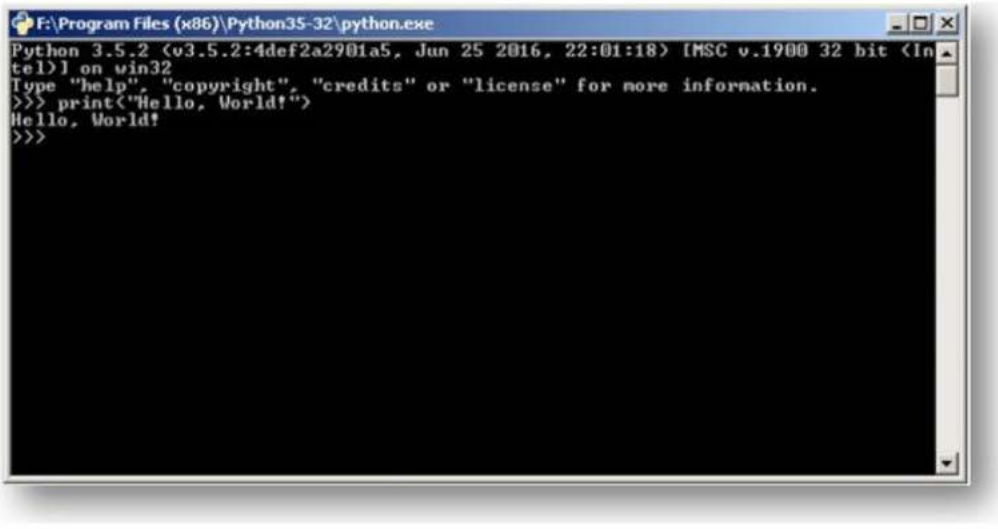


Рис. 7.7. Результат работы программы на Python в окне терминала

Получение такого результата означает, что установка интерпретатора Python прошла без ошибок.

7.2. Интерактивная среда разработки программного кода PyCharm

В процессе разработки программных модулей удобнее работать в интерактивной среде разработки (IDE), а не в текстовом редакторе. Для Python одним из лучших вариантов считается IDE PyCharm от компании JetBrains. Для скачивания его дистрибутива перейдите по ссылке: **https://www.jetbrains.com/pycharm/download/** (рис. 7.8).

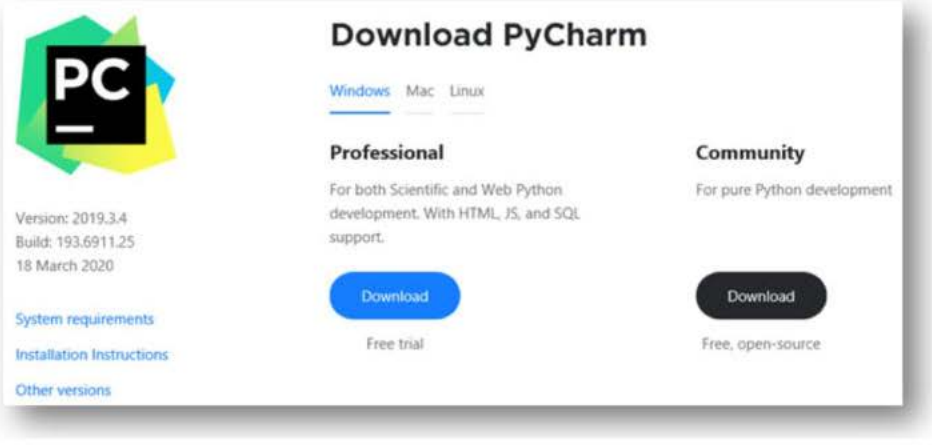


Рис. 7.8. Главное окно сайта для скачивания дистрибутива PyCharm

Эта среда разработки доступна для Windows, Linux и macOS. Существуют два вида лицензии PyCharm: **Professional** и **Community**. Мы будем использовать версию **Community**, поскольку она бесплатная и ее функционала более чем достаточно для наших задач. На момент подготовки этой книги была доступна версия PyCharm 2020.1.2.

Установка PyCharm в Windows. Запустите на выполнение скачанный дистрибутив PyCharm (рис. 7.9).



Рис. 7.9. Начальная заставка при инсталляции PyCharm

Выберите путь установки программы (рис. 7.10).

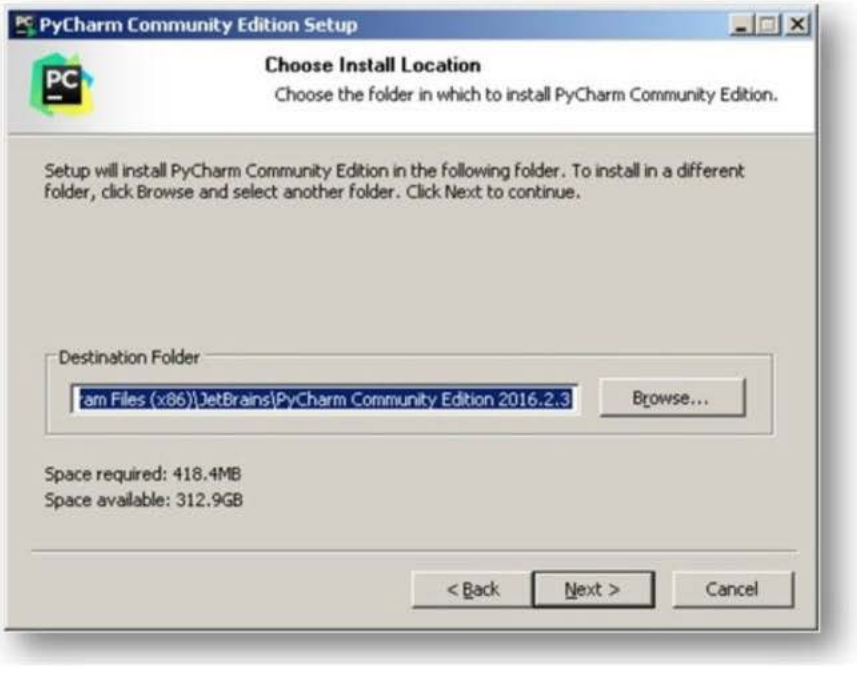


Рис. 7.10. Выбор пути установки PyCharm

Укажите ярлыки, которые нужно создать на рабочем столе (запуск 32- или 64-разрядной версии PyCharm), и отметьте флажком опцию **.py** в области **Create associations,** если требуется ассоциировать с PyCharm файлы с расширением «.py» (рис. 7.11).

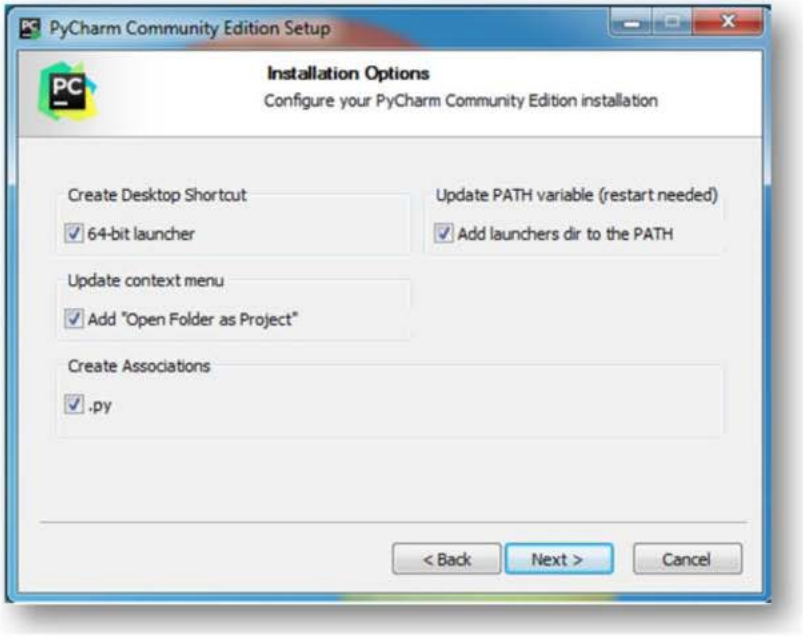


Рис. 7.11. Выбор разрядности устанавливаемой среды разработки PyCharm

Выберите имя для папки в меню **Пуск** (рис. 7.12).

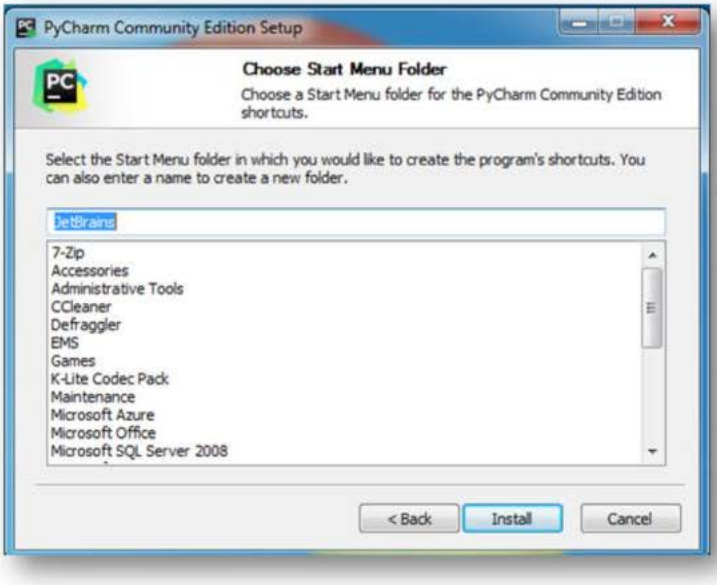


Рис. 7.12. Выбор имени папки для PyCharm в меню Пуск

Далее PyCharm будет установлен на ваш компьютер (рис. 7.13).

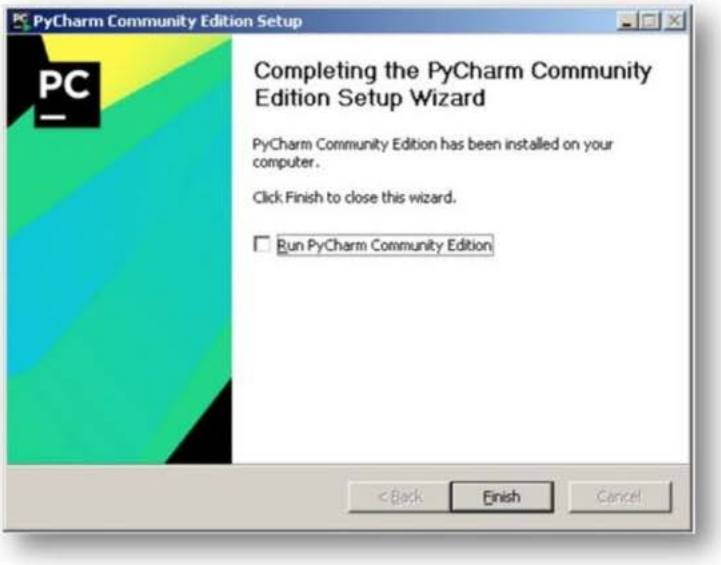


Рис. 7.13. Финальное окно установки пакета PyCharm

Установка PyCharm в Linux. Скачайте с сайта программы ее дистрибутив на свой компьютер. Распакуйте архивный файл, для чего можно воспользоваться командой:

```
> tar xvf имя_архива.tar.gz
```

Результат работы этой команды представлен на рис. 7.14.

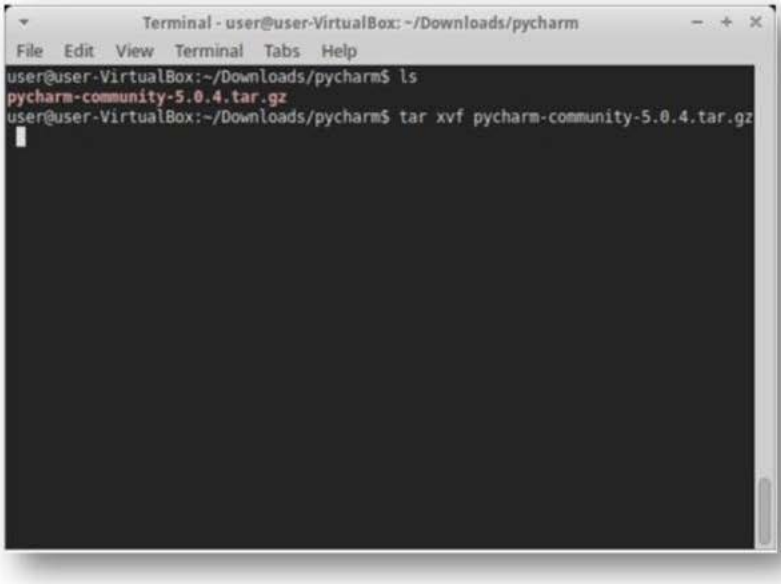


Рис. 7.14. Результат работы команды распаковки архива PyCharm

Перейдите в каталог, который был создан после распаковки дистрибутива, найдите в нем подкаталог bin и зайдите в него. Запустите установку PyCharm командой:

```
> ./pycharm.sh
```

Результат работы этой команды представлен на рис. 7.15.

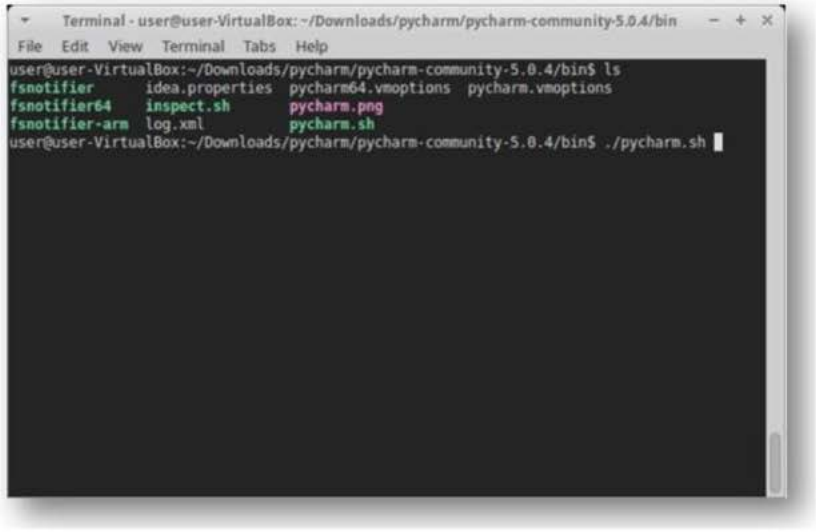


Рис. 7.15. Результаты работы команды инсталляции PyCharm

Проверка PyCharm. Запустите PyCharm и выберите вариант **Create New Project** в открывшемся окне (рис. 7.16).



Рис. 7.16. Создание нового проекта в среде разработки PyCharm

Укажите путь до создаваемого проекта Python и интерпретатор, который будет использоваться для его запуска и отладки (рис. 7.17).

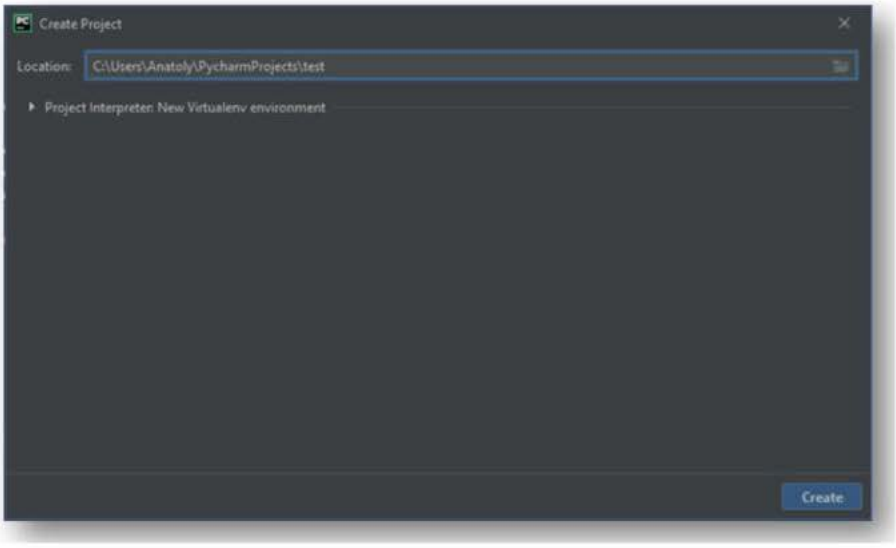


Рис. 7.17. Указание пути до проекта в среде разработки PyCharm

Добавьте в проект файл, в котором будет храниться программный код Python (рис. 7.18).

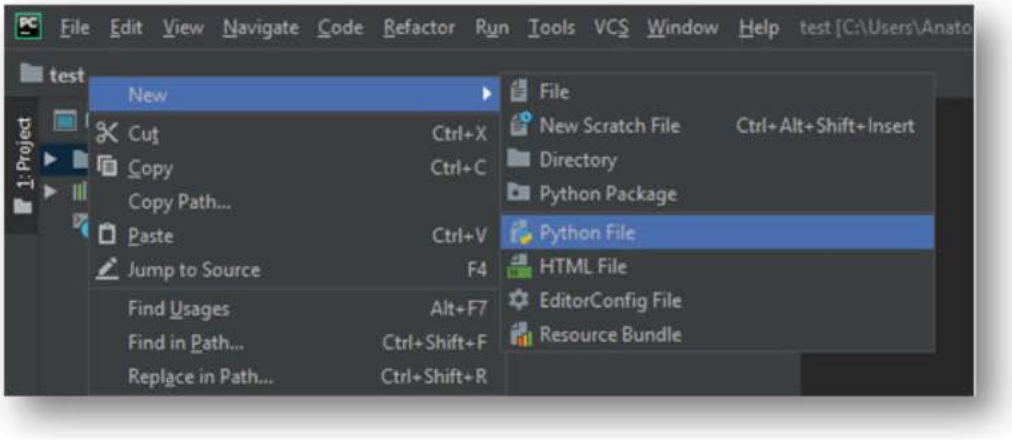


Рис. 7.18. Добавление в проект файла для программного кода на Python

Введите одну строчку кода программы (рис. 7.19).

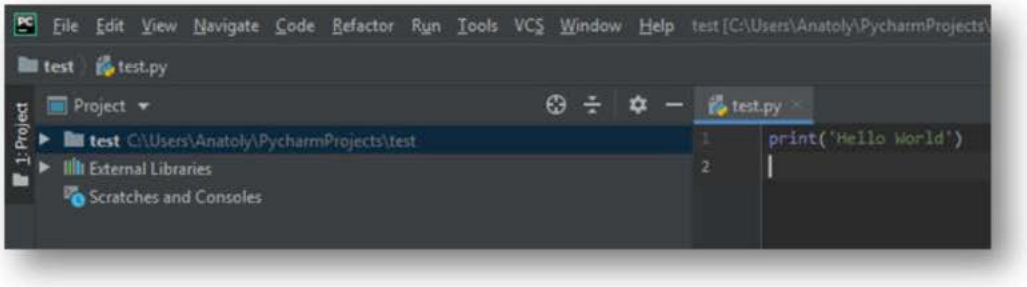


Рис. 7.19. Одна строка программного кода на Python в среде разработки PyCharm

Запустите программу командой **Run** (рис. 7.20).

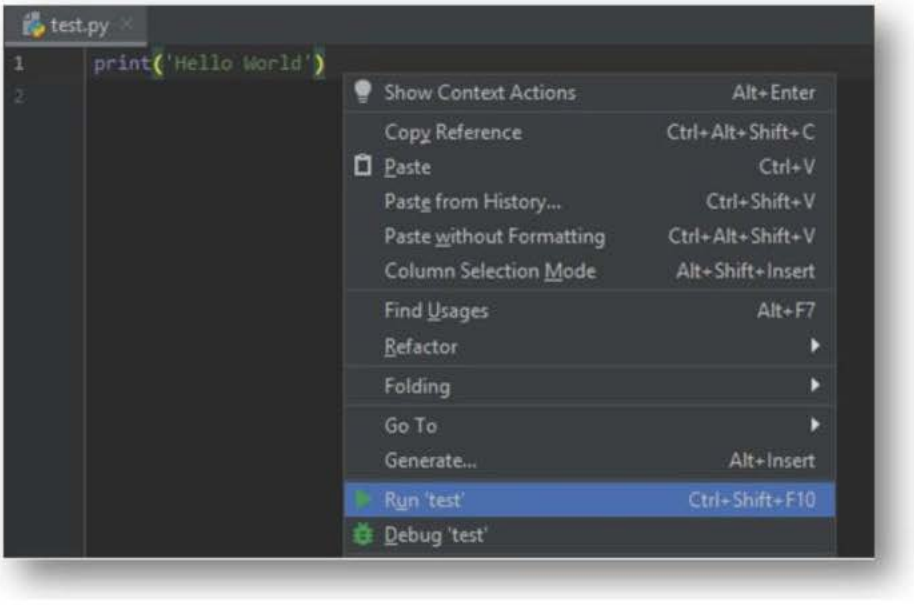


Рис. 7.20. Запуск программного кода на Python в среде разработки PyCharm

В результате в нижней части экрана должно открыться окно с выводом результатов работы программы (рис. 7.21).

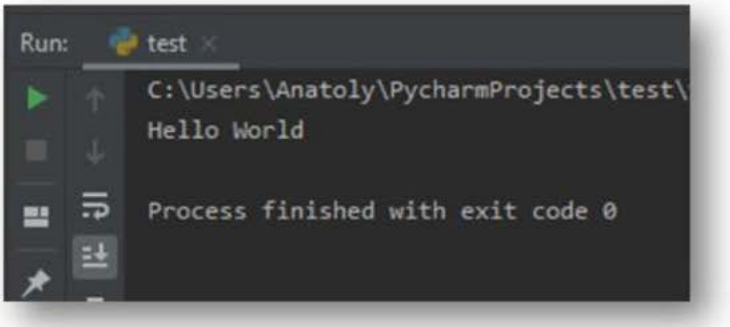


Рис.7. 21. Вывод результатов работы программы на Python в среде разработки PyCharm

7.3. Интерактивная среда разработки программного кода GigaIDE

GigaIDE — это российский продукт, созданный Сбербанком для разработки на различных языках. Она представляет собой сборку из всех бесплатных IDE от компании JetBrains с дополнительными плагинами и встроенным AI-ассистентом GigaCode. Скачать этот инструментарий можно с официального сайта по следующей ссылке:

https://gitverse.ru/features/gigaide/desktop/download/

Это инструментальное средство может работать в разных операционных системах (рис.7.22).

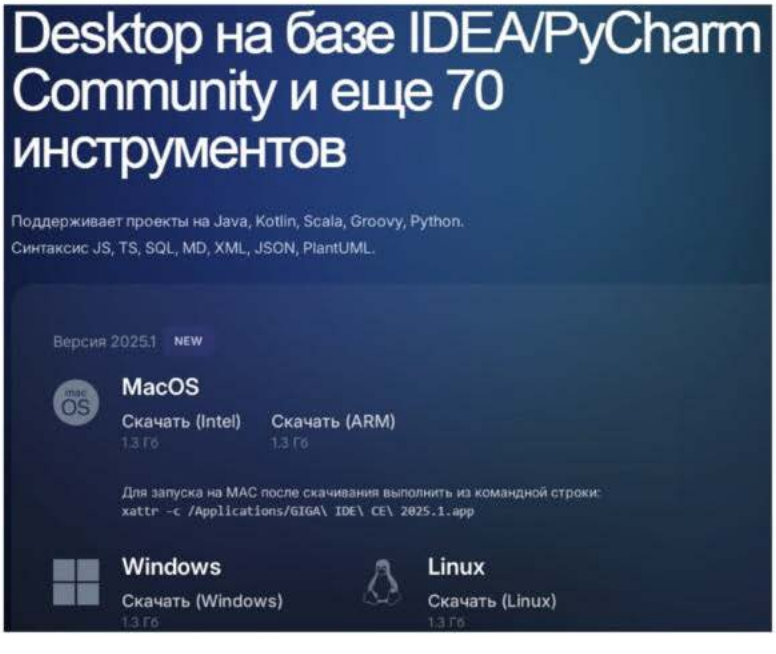


Рис.7. 22. Варианты скачивания GigaIDE для различных ОС

GIGA IDE обеспечивает программистам стабильность и привычный пользовательский интерфейс. Можно не беспокоиться, что вас заблокируют, как сделала компания JetBrains со своей инструментальной оболочкой PyCharm.

Среда совместима с различными плагинами и поддерживает разработку на многих языках программирования, включая Python. Явное преимущество данной среды разработки это встроенный интеллектуальный ассистент GIGA CODE, работающий с элементами искусственного интеллекта. По мере набора кода разработчиком, ассистент анализирует контекст, предлагает наиболее вероятные продолжения кода до конца строки, а также полные конструкции функций, циклов и других элементов. Помощник поддерживает более 30 языков программирования — и его создатели обещают, что языков станет еще больше.

Послесловие

В данной интерактивной электронной книге были рассмотрены только базовые элементы, которые используются при программировании на Python. Здесь сознательно максимально упрощены все примеры. Основная цель книги – познакомить читателей с типами данных и с базовыми инструкциями, с возможностями создания функций и классов из часто используемых фрагментов программ. Показать, как можно разбить приложение на модули и организовать импорт одного модуля из другого, как можно работать с файлами для ввода и вывода информации. Если после знакомства с приведенными материалами, вы почувствовали, что Python прекрасно работают, что программировать на нем не так уж и сложно, то основная цель данного издания достигнута. Значит можно устанавливать на свой компьютер необходимый инструментарий и приступать к практической деятельности.

Следующим шагом в освоении программирования на Python является изучение инструментария для создания привлекательного пользовательского интерфейса. Ведь программы пишутся не столько для себя, сколько для пользователей, которых может быть тысячи, десятки тысяч и более. Для этого в составе Python имеется встроенная библиотека Tkinter, а так же множество библиотек сторонних разработчиков: PyQT, PySide, WxPython, Streamlit и др. Родной для Python является библиотека Tkinter, которая входит в состав дистрибутива, ее не нужно искать и выполнять действия по дополнительной установке. Следующим изданием из серии «Академия Python», будет интерактивное электронное пособие по использованию библиотеки Tkinter для создания пользовательского интерфейса.

В завершении остается пожелать удачного применения в своей практической деятельности тех навыков и знаний, которые вы приобрели при прочтении данного материала.

Анатолий Постолит
д.т.н., профессор

Anatoly Postolit

Список источников и литературы

1. Дмитрий Мусин. Самоучитель Python. Изд. СИ, 2017. -154 с.
2. Бэрри П. Изучаем программирование на Python. O'Reilly, -М., :Изд. «Э», 2017. -624 с.
3. Копырин, А. С. Программирование на Python : учебное пособие / А. С. Копырин, Т. Л. Салова. – М, : ФЛИНТА, 2021. - 48 с. - URL: **https://znanium.com/catalog/product/1851993**.
4. Мэтиз Э. Изучаем Python. Программирование игр, визуализация данных, вэи-приложения: пер. с анг. -СПб.: Питер, 2017. -496 с.
5. Самоучитель Python. **https://pythonworld.ru/samouchitel-python**.
6. Топ языков программирования в 2025 году: рейтинг IEEE и влияние на него языковых моделей. **https://habr.com/ru/companies/selectel/articles/951348/**.
7. Федоров, Д. Ю. Программирование на языке высокого уровня Python : учеб. пособие. - М.: Издательство Юрайт, 2019.
8. Postolit.A. django_world_book. **https://github.com/APostolit/django_world_book.**
9. Python для начинающих. **https://okpython.net/.**
10. Python. Загрузите последнюю версию для Windows. **https://www.python.org/downloads/** .
11. Python. Урок 1. Установка. **https://devpractice.ru/python-lesson-1-install/.**
12. Python. Урок 16. Установка пакетов в Python. **https://devpractice.ru/python-lesson-16-install-packages/**.
13. Python's documentation, tutorials, and guides**. https://www.python.org/doc/**.
14. TIOBE Index for September 2022. **https://www.tiobe.com/tiobe-index/**.