МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

В.Ю. Полищук, Ю.А. Иванова, Е.С. Попова

ПРОГРАММИРОВАНИЕ НА PYTHON

**Методические указания
по выполнению лабораторных работ**

*Рекомендовано в качестве учебно-методического пособия
Редакционно-издательским советом
Томского политехнического университета*

Томский политехнический университет
2023

УДК 621.397:681.3.01(075.8)
ББК 32.973.2я73
А79

Полищук В.Ю., Иванова Ю.А., Попова Е.С.

А Программирование на Python: учебно-методическое пособие / В.Ю. Полищук, Ю.А. Иванова, Е.С. Попова; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2023. – 74 с.

Учебно-методическое пособие знакомит читателя с устройством ЭВМ, описывает краткую историю развития языков программирования, объясняет отличия в работе компилятора и интерпретатора, содержит подробное описание процесса установки сред разработки и синтаксиса языка Python с демонстрацией примеров кода. В конце каждого раздела приводятся контрольные вопросы для самопроверки, практические навыки закрепляются выполнением заданий к лабораторным работам.

Предназначено для бакалавров всех направлений, обучающихся в рамках модуля дополнительной специализации «Разработка методов вычислительного интеллекта на Python».

УДК 621.397:681.3.01(075.8)
ББК 32.973.2я73

Рецензенты

Начальник отдела развития
геоинформационного ПО ТомскНИПИНефть
А.А. Напрюшкин

Доцент кафедры
Автоматизации обработки информации ТУСУР
кандидат технических наук
Т.О. Перемитина

Оглавление

Введение.....5
I. Теоретический раздел 7
1. Общие сведения 7
1.1. Особенности устройства компьютеров.....7
1.2. Как компьютер исполняет программу 8
1.3. Основные особенности языка программирования Python 9
1.4. Контрольные вопросы для самопроверки.....10
2. Создание программ на языке программирования Python 11
2.1. Установка средств создания программ на языке Python 11
2.2. Введение в типы данных Python 16
2.3. Модель динамической типизации 19
2.4. Разделяемые ссылки.....20
2.5. Ввод и вывод данных 22
2.6. Пример простой программы на Python 26
2.7. Контрольные вопросы для самопроверки.....26
3. Условный оператор.....28
3.1. Логический тип данных.....28
3.2. Условные конструкции.....28
3.3. Контрольные вопросы для самопроверки.....32
4. Циклы в Python.....33
4.1. Цикл while 33
4.2. Цикл for 35
4.3. Итерируемые объекты в Python 36
4.4. Контрольные вопросы для самопроверки.....46
5. Работа с файлами. Создание функций. 47
5.1. Работа с текстовыми файлами 47
5.2. Работа с файлами *.xlsx 50
5.3. Создание собственных функций в Python.....51
5.4. Контрольные вопросы для самопроверки.....55
6. Объектно-ориентированное программирование в Python 56
6.1. Классы 56
6.2. Экземпляр класса.....56
6.3. Конструктор класса 57
6.4. Наследование 58

6.5. Инкапсуляция 59
6.6. Контрольные вопросы для самопроверки 61
7. Реализация простой базы данных средствами языка Python 62
7.1. История реляционных баз данных 62
7.2. Реляционная модель данных 62
7.3. Первичный и внешний ключи 63
7.4. Типы связей в реляционных базах данных 64
7.5. Нормализация отношений 65
7.6. Нормальные формы 65
7.7. Создание базы данных с помощью SQLite 65
7.8. Работа с SQLite и Pandas DataFrame 68
7.9. Контрольные вопросы для самопроверки 71
8. Визуализация модели линейной регрессии 72
8.1. Линейная регрессия 72
8.2. Библиотека matplotlib 74
8.3. Контрольные вопросы для самопроверки 74
II. Практический раздел 75
1. Лабораторная работа №1 75
2. Лабораторная работа №2 77
3. Лабораторная работа №3 79
4. Лабораторная работа №4 82
III. Раздел для самостоятельной работы 85
1. Самостоятельная работа №1 85
2. Самостоятельная работа №2 85
3. Самостоятельная работа №3 86
4. Самостоятельная работа №4 87
5. Самостоятельная работа №5 87
6. Самостоятельная работа №6 88
7. Самостоятельная работа №7 89
Заключение 90
Список литературы 91
Приложение А. Дополнительные самостоятельные работы 93

Введение

Навыки создания программ являются важной частью современного процесса обучения в высших учебных заведениях. Данное учебное пособие призвано помочь обучающимся приобрести понимание теоретических и практических вопросов создания программ на языке высокого уровня Python.

Язык программирования Python, Питон или Пайтон, называть можно и так, и так, был создан голландцем Гвидо ван Россумом.

В 80-е Гвидо ван Россум занимался проектированием языка ABC который можно назвать прототипом языка Python. Так как ABC разрабатывался как язык программирования для пользователей, которые не имели опыта в программировании, в результате ожидалось, что получится простой в освоении язык с простым синтаксисом, однако проект закрылся и по мнению Гвидо, из-за того, что язык медленно распространялся и не получал оперативной обратной связи от пользователей.

Следующим шагом к созданию языка стало участие Гвидо в проекте Амеба — проекте разработки операционной системы для крупных организаций. Так в 1989 году системе Амеба не хватало своего языка сценариев, по этой причине Гвидо ван Россум собирался написать язык программирования на основе ранних наработок языка ABC. Вскоре Гвидо показал коллегам прототип будущего Python.

Главная особенность первого прототипа заключалась в том, что будущий Python предлагал широкие возможности расширяемости, помимо стандартных возможностей, каждый программист мог самостоятельно добавить в систему нужные типы объектов. Прототип языка понравился коллегам Гвидо, которые начали использовать язык для внутренних проектов и помогли доработать код. Первый опубликованный дистрибутив языка Python версии 0.9.0 появился 20 февраля 1991 года, дата ориентировочная.

Следует упомянуть, что свое имя Пайтон получил от названия телесериала "Летающий цирк Монти-Пайтона", а не пресмыкающегося.

Получить дополнительные сведения о Python и скачать его дистрибутив можно с помощью официального сайта: https://www.python.org/

Учебно-методическое пособие содержит описание основных особенностей языка Python для обучающихся начального уровня, а именно: краткую историю создания языка программирования Python, как выполняется код программы на языке Python, вопросы установки средств создания приложений на языке Python, какие типы данных предоставляет для использования язык Python, варианты применения условных инструкций и работы с циклами, примеры работы с файлами, вопросы создания функций и применения объектно-ориентированного программирования. В материалах

пособия раскрываются темы создания базы данных, средствами языка Python, и вопросы применения линейной регрессии, как наиболее простого для освоения студентами, метода машинного обучения и представления результата линейной аппроксимации в виде графика с помощью библиотеки «matplotlib».

При ознакомлении с материалами учебного пособия важно, чтобы обучающиеся осваивали материал эффективно, поэтому в конце каждой главы приведены контрольные вопросы для самопроверки, которые позволят обучающимся студентам выявить пробелы в усвоенном материале.

Для формирования у студентов практических навыков программирования на языке Python, учебно-методическое пособие содержит задания, по изложенным темам, для выполнения четырех лабораторных работ, семи самостоятельных работ и дополнительные задания для выполнения пяти усложненных самостоятельных работ.

I. Теоретический раздел

1. Общие сведения

1.1. Особенности устройства компьютеров

Инструментом программиста является электронно-вычислительная машина (ЭВМ) – компьютер, общая схема устройства компьютера представлена на рис. 1.1. Центральный процессор (ЦП) производит все вычисления в ЭВМ. Место хранения файлов с программами – постоянная память, а при выполнении, данные выгружаются в оперативную память. Для ввода информации в компьютер используются устройства ввода, такие как клавиатура, а для вывода информации – устройства вывода, например, монитор.

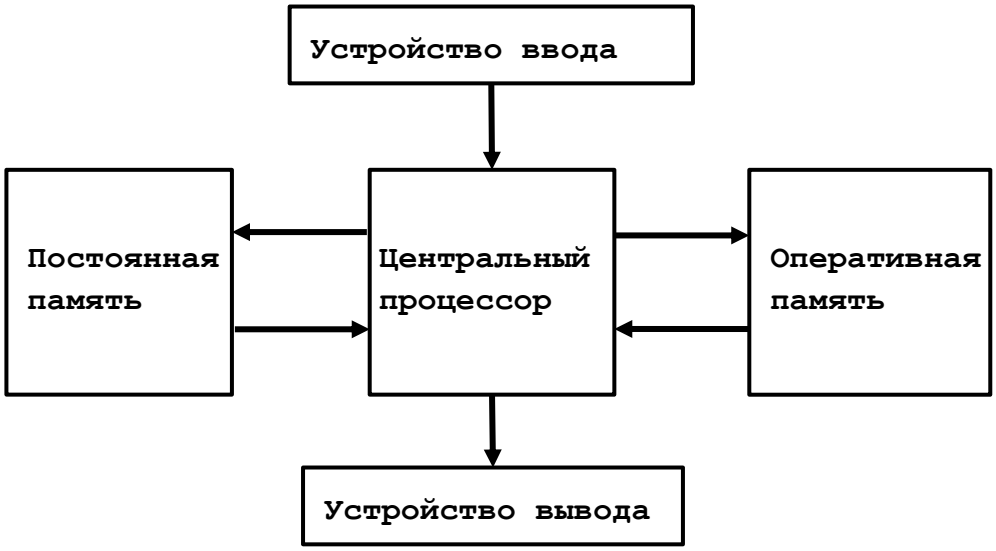


Рис. 1.1 Устройство ЭВМ.

Компьютер хранит и обрабатывает данные в виде двоичного кода, где информация представлена с помощью нулей и единиц, такой код называют машинным.

Для человека трудно писать программы в двоичном коде, который может выглядеть, например, так: 0011 0011 1011, поэтому со временем появились специальные средства – трансляторы, позволяющие переводить программы с понятного человеку языка программирования, на машинный язык. Чем ближе язык программирования к человеческому, тем он более высокоуровневый. К высокоуровневым языкам программирования относятся Python, Java, C#.

Первые программы создавались программистами в двоичном коде. Затем появился низкоуровневый язык ассемблер, уже не машинный язык, но

еще далекий от понятного человеку. Изучение ассемблера равносильно изучению архитектуры процессора, под управлением которого будет запущена программа. На языке ассемблера до сих пор пишут программы, например для программирования микроконтроллеров. Этот этап характеризуется функциональным подходом в программировании.

Следующий этап характеризуется появлением парадигмы объектно-ориентированного программирования (ООП), которая упрощает разработку масштабных промышленных систем. Большой вклад в развитие данной парадигмы внес ученый Бьерн Страуструп, создавший новый язык программирования C++. Данный язык по своей сути является расширением языка C, и предоставляет возможности для применения объектно-ориентированного подхода.

В период массового распространения персональных компьютеров и активного развития сети Интернет, потребовались новые технологии и языки программирования, так появился язык Java, примерно в одно время с Java появляется и Python.

1.2. Как компьютер исполняет программу

Для перевода программного кода с одного языка программирования на другой, например, с языка высокого уровня на машинный язык требуется специальная программа – транслятор, выделяют два основных способа трансляции: компиляция и интерпретация.

При компиляции весь исходный программный код сразу переводится в машинный, который сохраняется в отдельном исполняемом файле. Выполнение исполняемого файла обеспечивается операционной системой (ОС). После того как получен исполняемый файл, для его чтения транслятор уже не нужен [1].

При интерпретации выполнение кода происходит последовательно, условно, строка за строкой. По сути, операционная система взаимодействует с интерпретатором, а не с файлом, содержащим программный код. Интерпретатор же, прочитав очередную часть исходного кода, переводит его в машинный и передает его операционной системе, которая исполняет этот код и ждет следующей части кода от интерпретатора, и Python именно такой язык программирования – интерпретируемый [1].

Выполнение скомпилированной программы происходит быстрее, чем интерпретируемой, однако на современных ЭВМ снижение скорости выполнения при интерпретации часто не заметно.

Трансляция в Python

Python перед непосредственной интерпретацией переводит исходный программный код в промежуточный *байт-код*, что обеспечивает переносимость программ на другие операционные системы. Из-за этого этапа может наблюдаться некоторое замедление программ, написанных на языке Python по сравнению с такими языками программирования, как C++. Однако, в некоторых прикладных областях, таких как обработка данных, программный код на языке Python может обладать достаточно высокой производительностью. Это объясняется тем, что для решения подобных задач внутри интерпретатора Python используется скомпилированный программный код на языке C. Важно отметить, что при работе с языком Python скорость разработки новых программ обычно превышает скорость разработки программ на C-подобных языках [1].

1.3. Основные особенности языка программирования Python

Python является интерпретируемым языком программирования, это значит, что интерпретатор преобразует исходный код частями, последовательно, строку за строкой, в отличие от компилируемых языков, например, C или Java, для которых исходный код программы преобразуется в машинный код целиком.

Другой отличительной чертой языка Python является краткий и понятный синтаксис, который характеризуется ограниченным использованием вспомогательных синтаксических элементов, таких как скобки и точки с запятыми. Вместо них для выделения блоков кода используются отступы, что упрощает зрительное восприятие программ [1].

Python является одним из наиболее широко используемых языков программирования в разных областях, например, в системном программировании, разработке программ с графическим интерфейсом, разработка динамических веб-сайтов, создания программ для анализа данных и др.

На основании лицензии подобной General Public License (GNU) интерпретаторы Python распространяется свободно. General Public License (GNU) является универсальной общественной лицензией на свободное программное обеспечение, созданная в рамках проекта GNU в 1988 г., по которой автор передаёт программное обеспечение в общественную собственность.

Сильные стороны Python [1]:

- объектно-ориентированный;
- мощный с точки зрения функциональных возможностей:

- В Python реализована динамическая типизация, язык сам отслеживает типы объектов.
- Python автоматически следит за выделением памяти под объекты и её освобождением.
- Основным преимуществом языка является его модульность, позволяющая разрабатывать и быстро подключать отдельные библиотеки.
- Объектно-ориентированных подход позволяет создавать крупные системы и программные комплексы на Python.
- Множество встроенных типов объектов, таких как списки, словари, строки и др.
- Для выполнения узких задач в состав Python также входит большая коллекция пакетов библиотек, с помощью которых можно выполнить большое количество разнообразных задач.
- Лёгок в освоении.

1.4. Контрольные вопросы для самопроверки

1. Что такое трансляция в программировании?
2. Чем отличается процесс компиляции от интерпретации?
3. С точки зрения трансляции каким языком является Python?
4. Назовите сильные стороны языка программирования Python.
5. Для каких целей можно применять язык Python?

2. Создание программ на языке программирования Python

2.1. Установка средств создания программ на языке Python

Интерактивный режим

Для работы в интерактивном режиме необходимо установить среду IDLE, которая устанавливается вместе с интерпретатором *Python*.

Чтобы установить *Python* необходимо:

1) Сначала необходимо зайти на официальный сайт Python: https://www.python.org/

2) Затем скачиваем установочный файл последний версии

3) После скачивания файла выполняем установку *Python* в ОС *Windows*.

4) Теперь, когда *Python* установлен, открываем командную строку *Windows*. Одним из способов, например, нажав сочетание клавиш *Win + "X"*, во всплывшем списке меню нажимаем "Выполнить", появится окно, в появившемся окне вводим "*cmd*" и жмем "*Ok*". Также во всплывающем меню можно выбрать «Windows PowerShell». Результат, появившаяся командная строка *Windows*. Проверить установлен ли *Python* в системе можно командной:

```
python --version
```

Если *Python* установлен, мы увидим номер установленной версии.

Теперь можно открыть среду IDLE, в таком режиме программирования интерпретатор выполняет команды построчно. После ввода команды и нажатия клавиши *Enter*, интерпретатор выполняет ее и результат выводится на экран.

Это удобно, для ознакомления с особенностями языка или тестирования какой-нибудь небольшой части кода. В среде IDLE (в *Windows*), чтобы заново не вводить строку используют сочетания клавиш Alt+N и Alt+P, а в консоли можно прокручивать историю команд, используя стрелки вверх и вниз на клавиатуре.

Создание скриптов

При написании программы на языке Python ее исходный код сохраняется в файл с расширением **.py*. Создавать файлы с расширением **.py* можно не только в среде IDLE, но и например, в *PyCharm*.

Чтобы установить *PyCharm* нужно:

1) Сначала необходимо зайти на официальный сайт *PyCharm*: https://www.jetbrains.com/pycharm/download

2) Затем скачиваем установочный файл версии “*Community*”

3) После скачивания файла выполняем его установку.

4) После установки запускаем *PyCharm*, указываем путь к интерпретатору *Python*, который установили на первом шаге.

Jupyter notebook

Воспользоваться *Jupyter*-ноутбук можно после установки пакета программ *Anaconda*, для этого необходимо скачать дистрибутив с официального сайта: https://anaconda.org/. После установки появится возможность запустить *Jupyter*-ноутбук.

Jupyter-ноутбук – это среда разработки, где сразу можно видеть результат выполнения кода. Отличительной особенностью написания кода в блокноте Jupyter, от традиционных сред разработки, является то, что код разбивается на ячейки, каждая из которых может выполняться по отдельности. Файлы, создаваемые и редактируемые в блокноте Jupyter имеют расширение *.ipynb.

Jupyter-ноутбук представляет собой не статическую страницу, а интерактивную среду, которая позволяет писать и выполнять код на *Python* и других языках.

Альтернативой Jupyter-ноутбук является Google Colab (Google Colaboratory). В Google Colab, можно перейти по следующей ссылке: https://colab.research.google.com/notebooks/welcome.

На рис.2.1 представлен пример кода в *Jupyter*-ноутбук. Если *Jupyter*-ноутбук запущен в ОС Windows, то чтобы выполнить код нужно нажать на стрелку в соответствующей ячейке или выделить ячейку и нажать *ctrl+enter*. Для выполнения кода последовательно во всех ячейках можно воспользоваться сочетанием клавиш *ctrl+F9*.

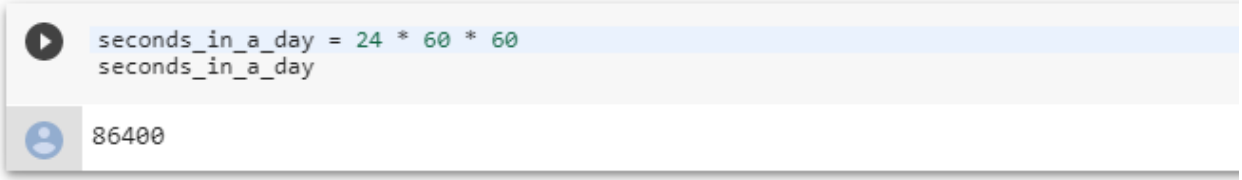


Рис. 2.1. Пример кода на языке *Python* в блокноте *Jupyter*

В программе представленной на рис. 2.1 производится подсчет количества секунд в сутках. Переменной *seconds_in_a_day* присваивается значение, равное 24(часа)*60(минут)*60(секунд). Вторая строка предназначена для вывода значения, хранящегося в переменной *seconds_in_a_day*.

Для создания нового файла вашей программы нужно перейти в меню *File→New Python 3 notebook* (рис. 2.2).

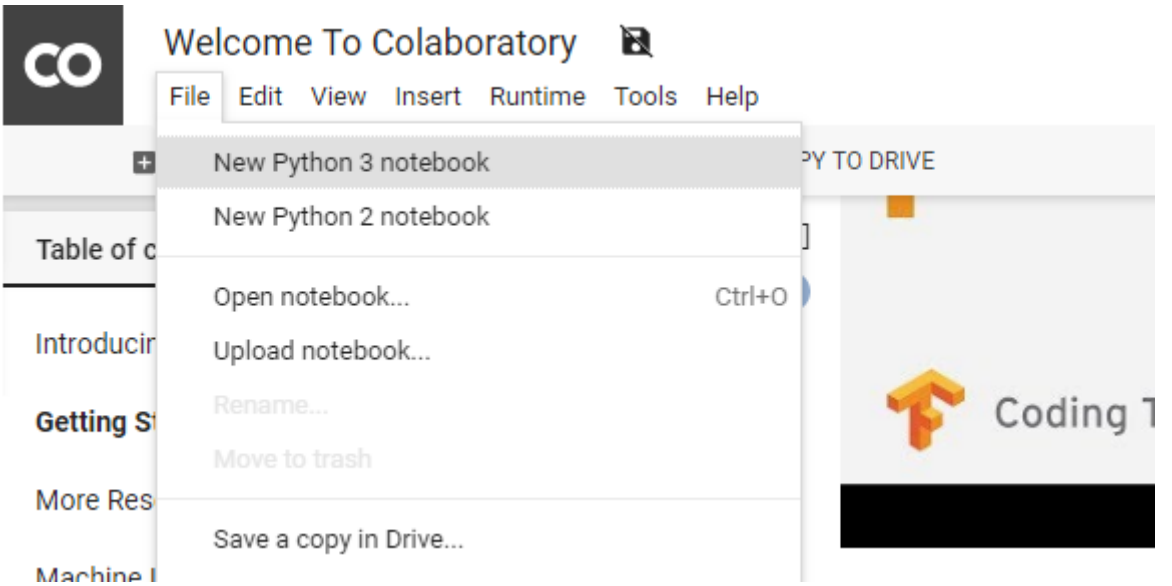


Рис. 2.2. Создание нового программного файла *.*ipynb*

В новой вкладке откроется файл *Untitled2.ipynb* (рис. 2.3), с которым вы будете работать.

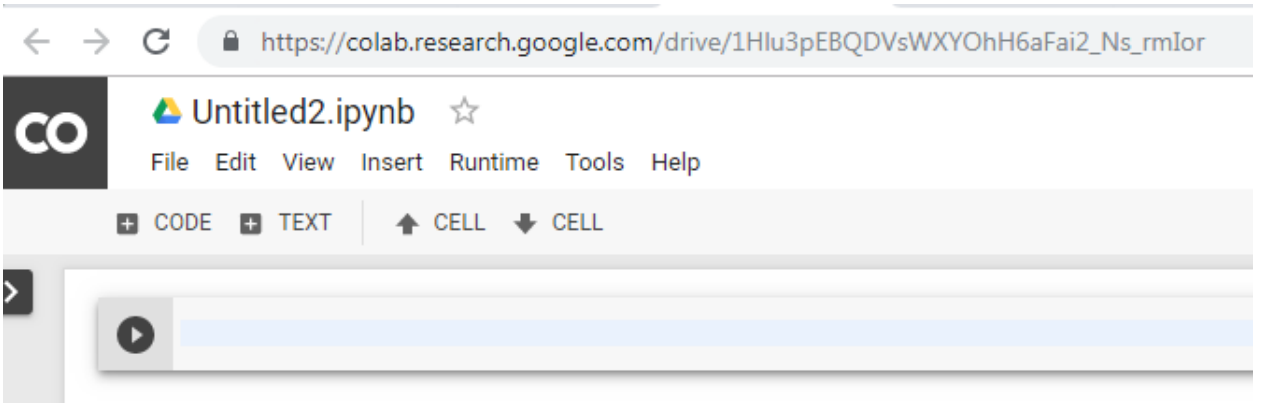


Рис. 2.3. Пример нового программного файла

Для переименования файла нужно выбрав в меню *File→Rename*. Открытие ранее созданных проектов осуществляется выбором *File → Open Notebook*. Появится диалоговое окно (рис. 2.4), отображающее недавно открытые файлы.

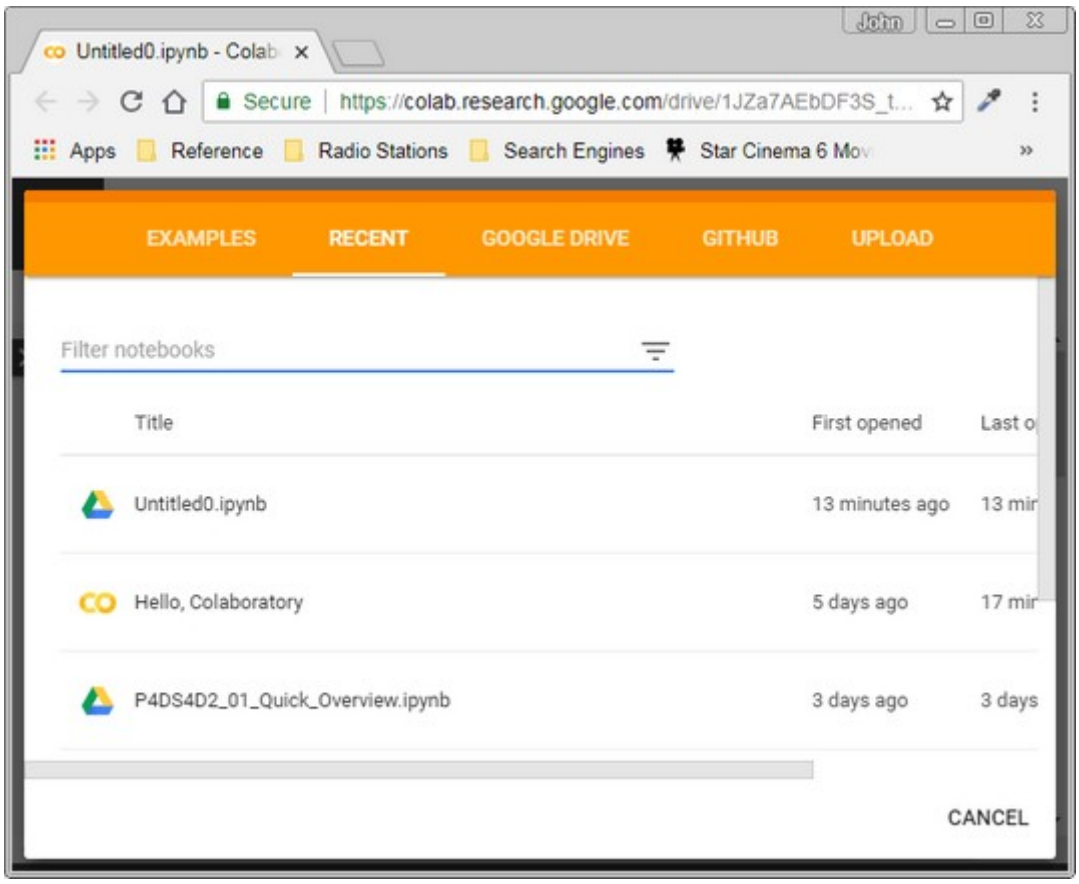


Рис. 2.4. Пример нового программного файла

Для того чтобы сохранить файл с новым именем на жесткий диск необходимо в меню выбрать *File → download *.ipynb*. После выполнения лабораторной работы нужно будет сохранять файлы на диске.

Для добавления новой ячейки выберите *Insert → Code cell* (рис. 2.5).

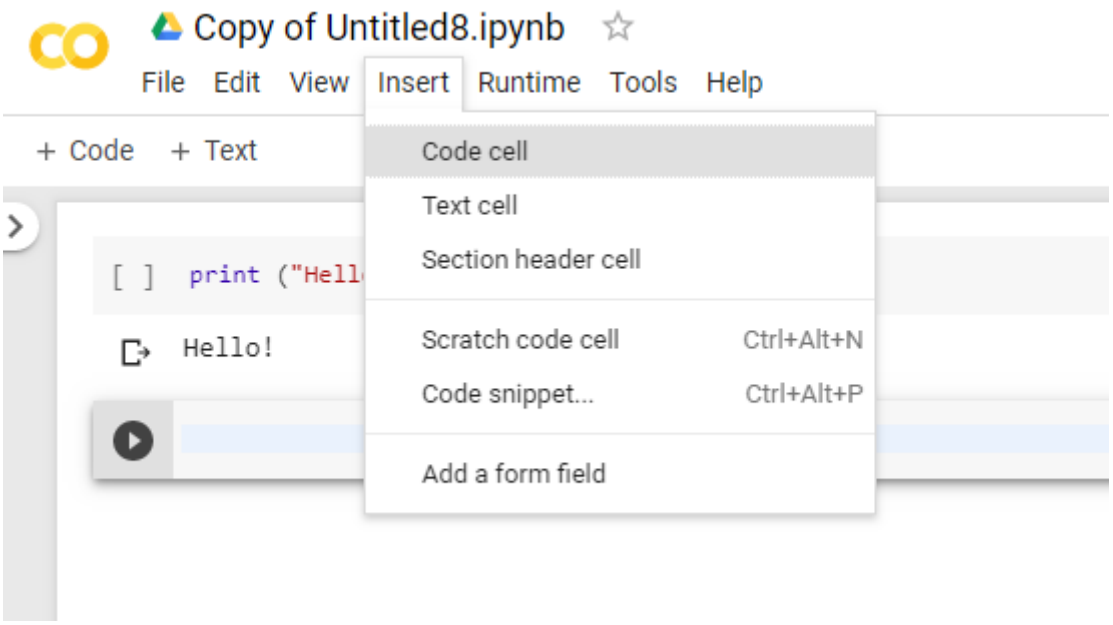


Рис. 2.5. Добавление новой ячейки кода

При наведении курсора мыши на ячейку кода, справа вверху появляется всплывающее меню (рис. 2.6). Нажатие на значок «*сообщение*» позволяет добавить комментарий к ячейке. Нажатие на корзину – удалить ячейку. Нажатие на «*шестерёнку*» даст возможность редактирования настроек данной ячейки (рис 2.6).

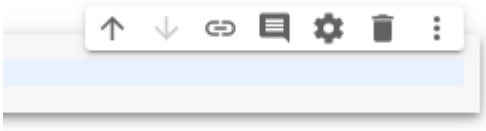


Рис. 2.6. Всплывающее окно ячейки кода

Для вывода автоматической подсказки используйте следующие сочетания клавиш:

«*Ctrl-пробел*» – чтобы открыть подсказку с кодом.

«*Ctrl-shift-пробел*» – чтобы открыть подсказки параметров.

Для получения дополнительных сведений воспользуйтесь документацией: https://docs.python.org/3/tutorial/.

Установка сторонних библиотек Python

Установку сторонних библиотек языка Python можно осуществить с помощью командой строки, как ее открыть в системе *Windows* указано вначале раздела. Перед установкой библиотек необходимо удостовериться, что установщик пакетов – pip, который устанавливается совместно с интерпретатором Python, имеет актуальную версию, сделать это можно введя следующую команду в командную строку:

```
python -m pip install --upgrade pip
```

Если окажется необходимым, то *pip* обновит себя сам. Затем уже можно вводить команду установки:

```
python -m pip install «ИМЯ БИБЛИОТЕКИ»
```

Так, например, если мы хотим установить библиотеку Pandas, то команда будет иметь следующий вид:

```
python -m pip install pandas
```

Следует отметить, что устанавливать пакеты библиотек можно не только через командную строку, а также с помощью среды разработки, например,

PyCharm. Если вы работаете в *Jupyter*-ноутбук, то можно установить библиотеки просто введя в ячейку те же команды для установки, но без «`python -m`», а если проводить установку через командную строку, то также можно использовать команду:

```
conda install matplotlib
```

2.2. Введение в типы данных Python

Данные хранятся на компьютере в виде последовательности битов. Бит – минимальная ячейка памяти, в которой может храниться только одно из двух значений: «0» или «1». Восемь соседних бит равны одному байту (рис. 2.7). Последовательные наборы байтов содержат все данные, хранимые в ЭВМ. Данные бывают разного размера и типа: целые, вещественные числа, текстовые символы и др.

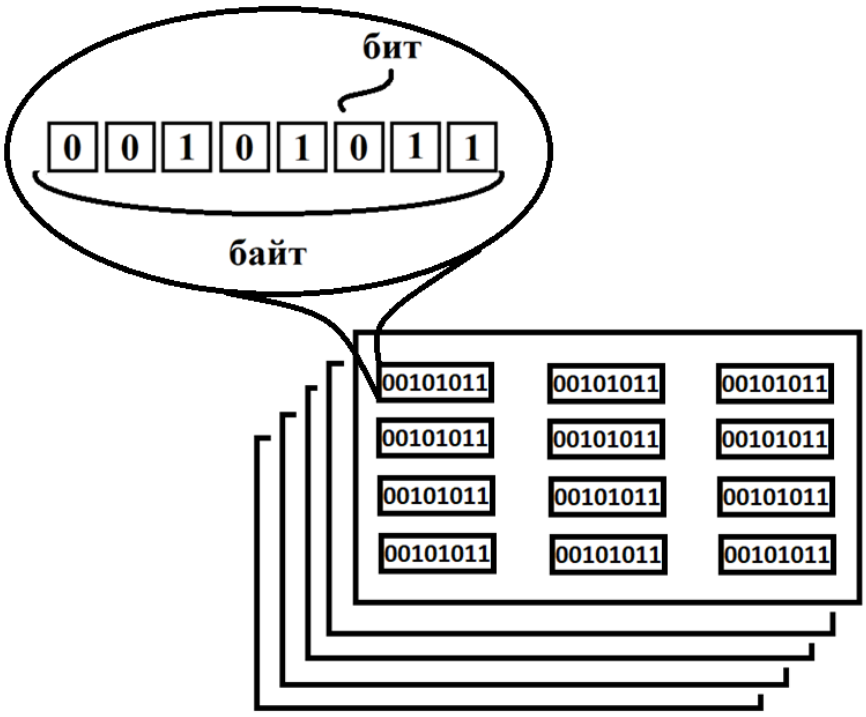


Рис. 2.7. Визуальное представление памяти компьютера.

С помощью операционной системы программа на Python получает доступ к определенной области памяти компьютера. В памяти хранится код самой программы, а также данные, созданные и/или загруженные во время ее выполнения. Операционная система гарантирует, что программа не допускается к чтению или записи данных в другие области памяти без соответствующего разрешения [2].

При хранении различных типов данных используется разное количество байт. Например, в 64-битной ЭВМ для хранения целого числа используется 64 бита (8 байт).

Python упаковывает каждое значение, например, целые числа, числа с плавающей точкой, строки и даже крупные структуры данных, функции и программы, в памяти как объекты. Объектом является фрагмент данных, в котором содержится, как минимум его уникальный номер (id – представляет собой адрес первого байта объекта в памяти), тип объекта (целый, вещественный и др.), счётчик ссылок на данные объект (рис. 2.8).

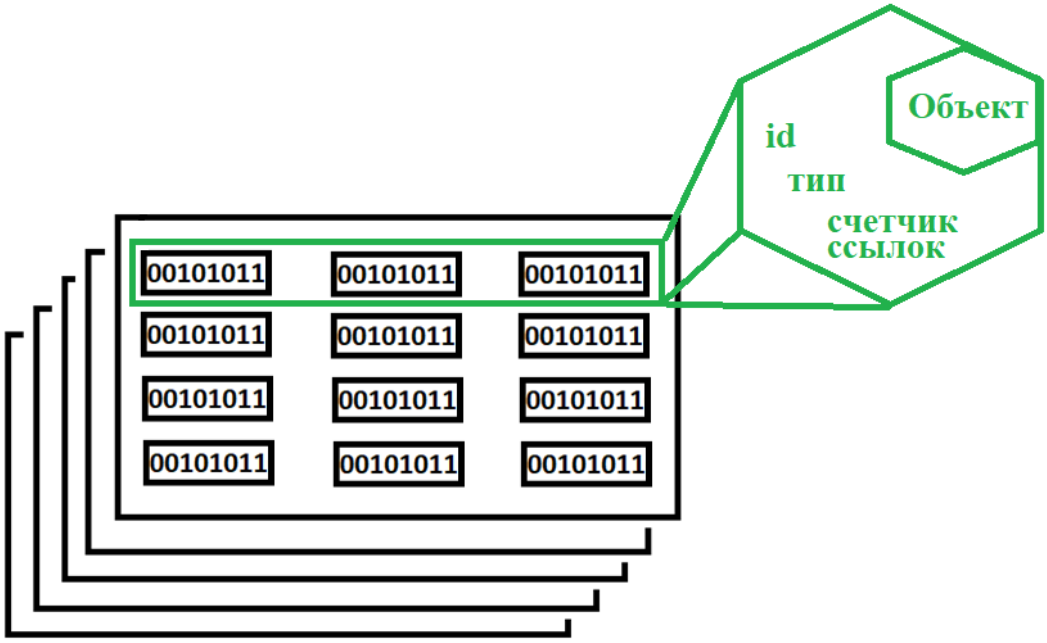


Рис. 2.8. Иллюстрация представления объекта в памяти

В таблице 2.1 [2] представлены базовые типы данных в Python. В первом столбце содержится название типа. Во втором столбце содержится имя этого типа в Python. Третий столбец указывает, можно ли изменить значение переменной после ее создания. В столбце «Примеры» представлены примеры значений, соответствующие типу.

Тип также определяет, можно ли изменить значение, тогда это будет либо изменяемое значение, либо неизменяемое. Неизменяемый объект как будто находится в закрытом ящике с прозрачными стенками увидеть значение вы можете, но не в силах его изменить. По той же аналогии изменяемый объект похож на коробку с крышкой: вы можете не только увидеть хранящееся там значение, но и изменить его, не изменяя его тип.

Таблица 2.1. Базовые типы данных в Python.

Название	**Тип**	**Является изменяемым**	**Пример**
Булево значение	bool	Нет	True, False
Целое число	int	Нет	3, 13
Число с плавающей точкой	float	Нет	5.45, 7.3e35
Комплексное число	complex	Нет	3 + 7j
Текстовая строка	str	Нет	‘слово’, “строка", ‘’’текст’’’
Список	list	Да	[‘один', ‘два', ‘три']
Кортеж	tuple	Нет	(1,2,3)
Множество	set	Да	set([4,5,6])
Словарь	dict	Да	{‘фрукт’: ‘апельсин', ‘возраст’: 10, ‘имя’: ‘Петя'}

Python является строго типизированным языком, а это означает, что тип объекта не изменяется, даже если его значение изменяемо.

В программе на языке Python, как и в большинстве других языков, связь между данными и переменными устанавливается с помощью знака «=» или оператора присваивания. Такая операция называется присваивание:

```
x = 9
```

Например, выражение «*x присвоить* 9» означает, что на объект, представляющий собой число 9, находящийся в определенной области памяти, теперь ссылается переменная «*x*», и обращаться к этому объекту следует по имени «*x*».

Когда интерпретатор Python перейдет к обработке данной строки, он автоматически создаст переменную «*x*», в которой запишет адрес целочисленного объекта «9». Все последующие операции присваивания, как

`x = 125` просто изменяют адрес в переменной «*x*» на адрес целочисленного объекта «125».

При этом переменные (имена) не имеют никакой информации о типе или ограничениях, связанных с ним. Понятие типа присуще только объектам, но не именам. Переменные универсальны по своей природе – они всегда являются всего лишь ссылками на конкретные объекты в конкретные моменты времени.

Переменные создаются при выполнении операции присваивания, могут ссылаться на объекты любых типов и им должны быть присвоены некоторые значения, прежде чем к ним можно будет обратиться.

2.3. Модель динамической типизации

Модель динамической типизации существенно отличается от моделей типов в традиционных языках программирования. Динамическая типизация обычно проще поддается пониманию начинающих, особенно если они четко осознают разницу между именами и объектами. Например, если ввести такую инструкцию *«x равно* 9*»*, концептуально интерпретатор Python выполнит эту инструкцию в три этапа. Сначала создается объект, представляющий число 9. Затем создается переменная *«x»*, если она еще отсутствует. И наконец в переменную *«x»* записывается ссылка на вновь созданный объект, представляющий число 9 [1]. Иллюстрация результата выполнения этих этапов представлена на рис. 2.9.

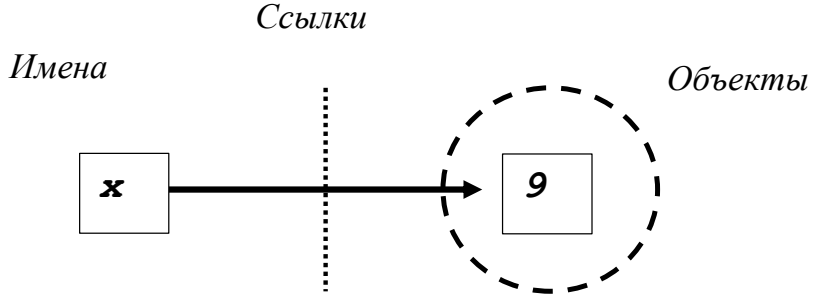


Рис. 2.9. Иллюстрация операции присваивания.

Как показано на схеме, переменные и объекты хранятся в разных частях памяти и связаны между собой ссылкой, которая на рисунке показана в виде стрелки. Переменные всегда ссылаются на объекты и никогда на другие переменные, но крупные объекты могут ссылаться на другие объекты, например, объект списка содержит ссылки на объекты, которые включены в список.

В конкретных терминах:

• Переменные – это записи в системной таблице, где предусмотрено место для хранения ссылок на объекты.

• Объекты – это области памяти размера, достаточного для представления значений этих объектов.

• Ссылки – это указатели на объекты.

2.4. Разделяемые ссылки

Введем в действие еще одну переменную и посмотрим, что происходит с именами и объектами в этом случае [1]:

```
x = 9
z = x
```

В результате выполнения приведенных выше инструкций получается схема взаимоотношений, отраженная на рис. 2.10. Вторая инструкция вынуждает интерпретатор создать переменную «*z*» и использовать для инициализации переменную «*x*», при этом она замещается объектом, на который ссылается 9, и «*z*» превращается в ссылку на этот объект. В результате переменные «*x*» и «*z*» ссылаются на один и тот же объект, то есть указывают на одну и ту же область в памяти. В языке Python это называется *разделяемая ссылка* – несколько имен ссылаются на один и тот же объект.

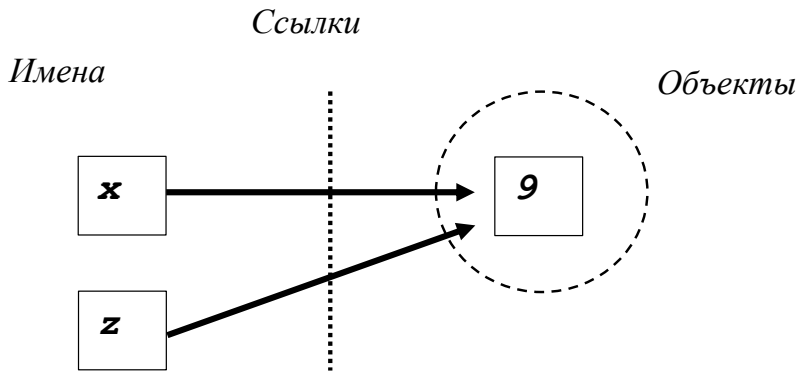


Рис. 2.10. Разделяемая ссылка.

Переменная «*z*» превращается в ссылку на объект 9. Переменная является указателем на область памяти объекта, созданного в результате выполнения операции присваивания. Добавим еще одну инструкцию:

```
x = 9
```

```
z = x
x = "value"
print(z) # z по прежнему равно 9

результат:
9
```

В результате выполнения этой инструкции создается новый объект, представляющий строку «*value*», а ссылка на него записывается в переменную «*x*» (рис. 2.11). Однако эти действия не оказывают влияния на переменную «*z*» – она по-прежнему ссылается на первый объект – целое число 9. В результате схема взаимоотношений приобретает вид, представленный на рис. 2.11.

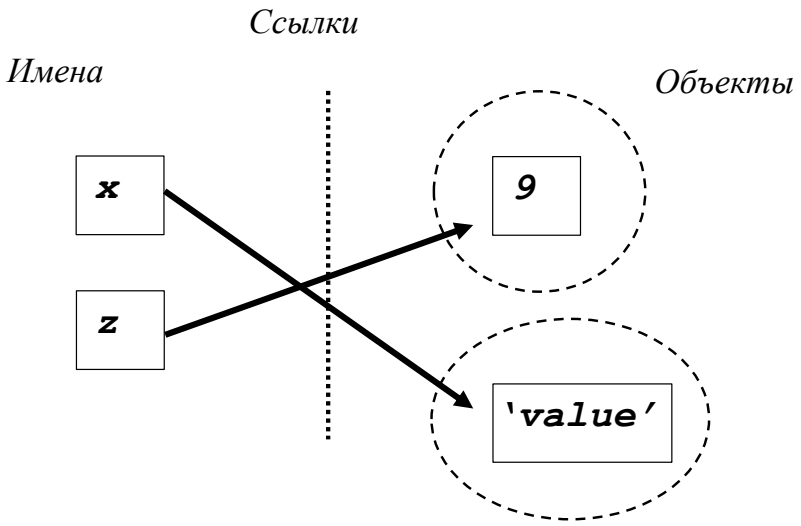


Рис. 2.11. Схема взаимоотношений имен и объектов.

Например, рассмотрим следующие три инструкции:

```
x = 3
z = x
x = x + 2
```

В этой последовательности происходят те же самые события: интерпретатор Python создает переменную «*x*» и записывает в нее ссылку на объект 3. После этого он создает переменную «*z*» и записывает в нее ту же ссылку, что хранится в переменной «*x*». Наконец, последняя инструкция создает совершенно новый объект, в данном случае – целое число 5, которое является результатом выполнения операции сложения. Это не приводит к изменению переменной «*z*». В действительности нет никакого способа

перезаписать значение объекта 3, целые числа относятся к категории неизменяемых, и потому эти объекты невозможно изменить.

Переменные в языке Python, в отличие от других языков программирования, всегда являются указателями на объекты, которые доступны для изменения, так запись нового значения в переменную не приводит к изменению первоначального объекта, но приводит к тому, что переменная начинает ссылаться на совершенно другой объект. В результате инструкция присваивания может воздействовать только на одну переменную. Однако, когда появляются изменяемые объекты и операции, их изменяющие, картина несколько меняется [1].

2.5. Ввод и вывод данных

Ввод и вывод данных являются важными аспектами программирования. Без возможности ввода данных программы выполняли бы одну и ту же последовательность действий, за исключением случаев, когда значения генерируются случайным образом внутри программы. Вывод данных позволяет визуализировать, использовать и передавать результаты работы программы.

Для вывода данных в Python используется функция `print()`, а функция `input()` позволяет пользователю осуществлять ввод необходимых данных.

Вывод данных

В Python за вывод данных отвечает функция `print()`, которая выводит содержимое, указанное внутри ее скобок, на экран. В скобках могут быть любые типы данных. Кроме того, количество данных может быть различным [3]:

```
print(3) # выведет 3
print(5.6) # выведет 5.6
print("Hi") # выведет Hi
print("x:", 2) # выведет Hi

a = 1
b = 2
c = 3
print(a, b, c) # выведет 1 2 3
```

Если функции `print()` передать выражение, то сначала оно выполнится, после чего `print()` уже выведет результат:

```
    print("Hello" + " " + "world!") # выведет Hello
world!
```

```
print(5 + 2/2) # выведет 6
```

В `print()` предусмотрены дополнительные параметры. Например, через параметр `sep` можно указать отличный от пробела разделитель строк:

```
a = 1
b = 2
c = 3
print(a, b, c, sep="-") # выведет 1-2-3
```

Параметр `end` позволяет указывать, что делать, после вывода строки. По умолчанию происходит переход на новую строку. Однако, это действие можно отменить, указав любой другой символ или строку:

```
print(40, end=">") # выведет 40>
```

Следующее, что стоит упомянуть о функции `print()`- это использование форматирования строк. На самом деле это не имеет никакого отношения к `print()`, а применяется к строкам. Но обычно используется именно в сочетании с этой функцией.

Форматирование может выполняться в так называемом старом стиле или с помощью строкового метода format. Старый стиль также называют Си-стилем, так как он схож с тем, как происходит вывод на экран в языке C. Рассмотрим пример [3]:

```
a = 1
b = 2
c = 3
print("a = %s, b = %d, c = %.2f" % (a,b,c))
# выведет a = 1, b = 2, c = 3.00
```

Здесь вместо трех комбинаций символов `%s`, `%d`, `%f` подставляются значения переменных `a`, `b`, `c`. Буквы `s`, `d`, `f` обозначают типы данных – строку, целое число, вещественное число. Если бы требовалось подставить три строки, то во всех случаях использовалось бы сочетание `%s`. Мы можем указать, сколько требуется знаков после запятой, записав перед буквой `f` точку с желаемым числом знаков в дробной части:

Теперь посмотрим на метод `format()`:

```
c = 3
print("a = {0}, b = {1}, c = {2}".format("a",1, c))
# выведет a = a, b = 1, c = 3
```

В строке в фигурных скобках указаны номера данных, которые будут сюда подставлены. Далее к строке применяется метод `format()`. В его скобках указываются сами данные, можно использовать переменные. На нулевое место подставится первый аргумент метода `format()`, на место с номером 1 – второй и т. д. Следует отметить, что возможности метода `format()` существенно шире, но нам пока будет достаточно этого.

Начиная с версии 3.6 в языке *Python*, появился еще один способ форматирования строк – f-строки. Этот способ является более кратким, удобным к восприятию и менее подверженным ошибкам, чем другие способы форматирования, при этом и быстрее всех остальных.

Чтобы воспользоваться f-строкам достаточно сначала поставить букву f, затем открыть кавычки, где указать необходимые для вывода данные, здесь следует отметить, что имена переменных, выражения и другие инструкции следует указывать в фигурных скобках.

Значением являющимся результатом работы f-строк является строка, поэтому взаимодействие с ним происходит как со строкой, оно может быть передано функции или присвоено переменной.

Рассмотрим несколько примеров применения f-строк.

```
name = "Ivan"
age = 10
print(f"Hello, {name}. You are {age}.")
# выведет Hello, Ivan. You are 10.
```

Следует отметить, что для применения f-строк можно использовать и заглавную букву F.

Передавать в f-строки можно как переменные, так и все допустимые выражения Python.

```
print(f" x = {2 * 3 - 1}")
# выведет x = 5
```

Вызов функций также доступен в f-строках.

```
print(f"{len('value')}")
# выведет 5
```

В приведенном коде, в f-строки передана функция определения длины последовательности, в данном случае последовательностью является слово `'value'`.

Ввод данных

За ввод данных в программу с клавиатуры в Python отвечает функция `input()`. При вызове данной функции, программа останавливает свое выполнение и ожидает, когда пользователь введет текст и нажмет Enter. После того как текст был введен, функция передает его программе [3].

Функция `input()` передает введенные данные в программу, их можно присвоить переменной [3]:

```
x = input() # введем 3
print(f"{x}")
# выведет 3
```

Чтобы не вводить в замешательство пользователя рекомендуется, для функции `input()` использовать параметр-приглашение. Это приглашение выводится на экран при вызове `input()`:

```
x = input("Введите число: ") # введем 2
print(f"x = {x}")

результат работы программы:
Введите число:
2
x = 2
```

Следует отметить, что в результате работы функции `input()` в программу передается объект типа строка. Неважно, что именно пользователь ввел, функция `input()` все равно вернет объект строкового типа. Если необходимо, чтобы переданные пользователем данные имели другой тип данных, то следует использовать функции преобразования типов.

```
x = input("Введите число: ") # введем 2
print(f"x_int = {int(x)}")
print(f"x_float = {float(x):.3f}")


результат работы программы:
Введите число:
2
x_int = 2
x_float = 2.000
```

В данном случае с помощью функций `int()` и `float()` строковое значение переменной «*x*» преобразуются в целое число и вещественное

число, соответственно. Следует обратить внимание, что для вещественного числа можно указать количество знаков после запятой.

2.6. Пример простой программы на Python

Реализуем первую программу, которая должна предложить пользователю ввести два любых числа, а затем произвести математические операции над ними, такие как сложение, умножение и вычитание.

Сначала объявим переменные:

```
a = int(input('Введите первое число:'))
b = int(input('Введите второе число:'))
```

В приведенных строках кода, происходит следующее: сначала программа прерывает свой поток действий с помощью функции «*input»,* выводя на экран предложение пользователю ввести число, чтобы пользователь ввел число, затем полученное значение преобразуется в целое число с помощью функции «*int*», так как функция «*input*» возвращает данные текстового типа. И последнее действие – это присваивание переменной полученного значения в результате описанной работы функций.

Теперь в соответствии с заданием нужно произвести математические операции:

```
print(f"Сумма: {a + b}")
print(f"Произведение: {a * b}")
print(f"Разность: {a - b}")
```

В приведенных строках кода, происходит следующее, для вывода результата используется функция «*print*», математические операции выполняются с помощью математических операторов сложение (+), вычитание (-) и умножение (*). Следует отметить, что математические операции выполняются внутри f-строк, которые обеспечивают необходимое форматирование результата вычислений.

2.7. Контрольные вопросы для самопроверки

1. Какие вы знаете способы открытия командной строки в операционной системе Windows?

2. Назовите команду, которую нужно ввести в командную строку, чтобы обновить установщик пакетов pip?

3. Сколько бит в одном байте?

4. Чему будет равно значение переменной «x» и почему, после выполнения следующих инструкций:

```
y = 10
x = y
y = x + 11
```

3. Условный оператор

3.1. Логический тип данных

Логический тип данных (bool) – это простой тип данных, являющийся производным от типа «int». Переменные данного типа могут принимать два значения: «**истина**» – True и «**ложь**» – False.

Для работы с логическим типом данных предусмотрены следующие функции [4]:

- **and** – логическое «И» для двух переменных. Возвращает True, если обе переменные принимают значение True (истина), иначе возвращает False;
- **or** – логическое «ИЛИ» для двух переменных. Возвращает False, если обе переменные принимают значение False (ложь), иначе возвращает True;
- **xor** – логическое «исключающее ИЛИ» для двух переменных. Возвращает False, если обе переменные содержат одинаковые значения True или False, иначе возвращает True;
- **not** – логическое «НЕ» (отрицание) для одной переменной. Возвращает False, если переменная имеет значение True (истина), а True в противном случае.

Приведём таблицу истинности логических функций. Для удобства обозначим значения True = 1, False = 0:

x	y	not x	x or y	x and y	x xor y
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

3.2. Условные конструкции

Условные конструкции существуют для того, чтобы программист мог задавать несколько вариантов поведения программы при различных условиях [5].

Предположим, что нужно написать программу, которая вычисляет математическое выражение, содержащее деление. Выражение будет вычислено, если знаменатель не равен нулю, в противном случае – программа должна сообщить, что значение выражения не определено.

В такой программе будет проверяться условие на равенство нулю знаменателя: если оно истинное (True) – выполняется один набор инструкций, если ложно (False) – другой набор инструкций.

Синтаксически блок условия представлен на рис. 3.1 (а). Для лучшего понимания представим его в виде блок-схемы (рис. 3.1 (б)).

if <условие>:
    оператор_1
else:
    оператор_2

(а)

(б)

Рис. 3.1. Условный оператор «if»:
а) синтаксис; б) блок-схема.

Условные операторы в Python иногда называют операторами ветвления. Они состоят из заголовка и тела. Заголовок – это сама конструкция «`if (условие):`». Тело – операторы, записанные с отступом после двоеточия.

Результатом проверки условия чаще всего является логическое выражение (bool), но также условие считается истинным, если выражение в скобках не равно нулю или не является пустым (None).

В качестве условия может выступать результат операции сравнения:

>=	больше или равно
<	меньше
<=	меньше или равно
==	равно
!=	не равно

Вложенность операторов в языке Python определяется отступами. Если нужно, чтобы интерпретатор выполнил код при истинности условия, необходимо обязательно сделать отступ в виде четырех пробелов:

```
x = 1
y = 2
if (x < y):
    print('y больше x')
```

В повседневной жизни ситуации, в которых необходимо осуществить выбор между несколькими вариантами развития событий, возникают достаточно часто. К ним, например, относятся такие условия выбора, как прочитать лекцию или нет, сдан тест или нет и др. Подобные ситуации сопоставимы с ситуациями, когда программисту надо написать программу ход алгоритма работы которой может поменяться в результате возникновения тех или иных событий, например, окажется ли одно число больше другого или нет.

В приведенном примере (листинг. 3.2) сравниваются два числа `x` и `y`. Если `x < y` истинно, то выводится соответствующее сообщение. Эту конструкцию можно дополнить с помощью оператора `else`. Другими словами, программе нужно указать, что делать, если условие оказывается ложным. Необходимо запомнить, что в блоке `else` не задается никакое логическое выражение, этот блок выполняется, когда условие в блоке `if` ложно (False). Поэтому ситуация, при которой выполнятся обе ветви невозможна.

```
x = 1
y = 2
if (x<y):
    print('y больше x')
else:
    print('y меньше или равен x')
```

Рассмотрим еще одну ситуацию, когда необходимо перебрать несколько, больше двух, возможных вариантов. Например, программа получает значение баллов за сданный студентом тест, и в зависимости от полученного значения, программа выдаст соответствующую оценку по пятибалльной шкале.

В данном примере необходимо реализовать множественный выбор, для этого можно воспользоваться оператором `elif`, сокращение от `else if`. Данная конструкция позволяет реализовать программу, которая будет выбирать из нескольких альтернативных условий:

```
# x – количество баллов за тест
x = 70
if (x > 90):
    print('Оценка 5')
elif (x > 60):
    print('Оценка 4')
elif (x > 40):
    print('Оценка 3')
elif (x > 20):
```

```
        print('Оценка 2')
    else:
        print('Незачет')
```

Таким образом, оператор `elif` позволяет упростить код, сделать его более читаемым, при этом избежав написания нескольких последовательных блоков `if`.

Иногда, для сокращения записи используется так называемый тернарный оператор, благодаря ему условный оператор можно записать в одну строку. Часть кода, стоящая перед оператором if выполняется, если условие истинно, в случае ложного значения условия выполнится код, стоящий после оператора `else`.

```
print(f"{'y больше x ' if (x<y) else 'y меньше или равен x'}")
```

Логические условия можно комбинировать с помощью логических операторов: *not, and, or, xor*.

Например, необходимо, чтобы одновременно выполнились два условия `a > b` и `c > a`:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Оба условия выполнены")
```

Если необходимо, чтобы хотя бы одно из условий было выполнено, применяется оператор `or`:

```
if a > b or a > c:
  print("Хотя бы одно из условий истинно")
```

Можно задавать дополнительные условия внутри блока условия, такие конструкции называются вложенными.

```
    x = 41
if x > 10:
  print("Больше 10,")
  if x > 20:
    print("и больше 20!")
  else:
    print("но не больше 20.")
```

3.3. Контрольные вопросы для самопроверки

1. Опишите логический тип bool.
2. Опишите синтаксическую конструкцию if-elif-else.
3. Приведите блок-схему логического оператора.
4. Какие значения может принимать выражение, следующее за оператором `if`.
5. Придумайте свой пример, содержащий логическое условие и запишите его с помощью тернарного оператора.

4. Циклы в Python

Когда в программе необходимо выполнить некоторую последовательность действий несколько раз, применяются циклы. Циклы в языке Python можно реализовать с помощью конструкции: *while* и *for*.

4.1. Цикл while

Инструкция *while* позволяет реализовать повторение части кода, то есть организовать цикл [1]. На рис. 4.1 представлена блок-схема цикла, которую можно интерпретировать следующим образом: тело цикла − это набор инструкций, который необходимо выполнить многократно, пока <Условие повтора> истинно.

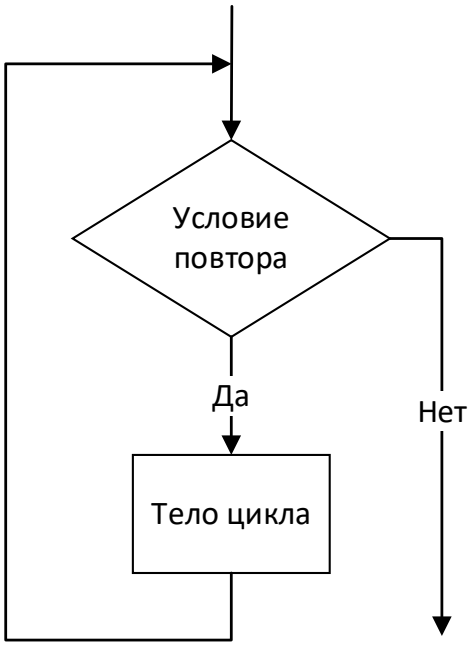


Рис. 4.1. Блок схема цикла.

Простая форма цикла *while* может быть реализована в программе следующим образом:

```
while <условие_повтора>: # пока условие истинно
    набор_инструкций # выполняется во время работы цикла
```

Интерпретатор продолжает проверять условное выражение в строке заголовка и выполнять вложенные инструкции в теле цикла, пока условное выражение не вернёт ложное значение:

```
i = 0
while i < 4: # цикл будет выполняться пока i меньше 4
    print(f"{i}")
    i+=1
```

Наиболее полная форма инструкции *while* может содержать блок с необязательной частью *else*, которая выполняется, когда <условие_повтора> становится ложным [1]:

```
    while <условие_повтора>:
        набор_инструкций_1
    else: # необязательная часть else
        набор_инструкций_2 # выполняется, если условие
ложно
```

Продемонстрируем пример работы блока else [1]:

```
i = 0
while i < 4: # цикл будет выполняться пока i меньше 4
    print(f"{i}")
    i+=1
else:
    print(f"{i} >= 4 ")
```

Теперь, настало время обратить внимание на две простые инструкции, которые могут использоваться только внутри циклов, это инструкции *break* и *continue*.

С помощью инструкции *break* можно выйти из цикла, и программа продолжит выполнение следующего за блоком цикла оператора.

С помощью инструкции *continue* можно перейти на следующую итерацию цикла, то есть «пропустить» выполнение операторов, следующих за *continue*.

С учетом инструкций *break* и *continue*, цикл *while* в общем виде выглядит, довольно объемно:

```
while <условие_повтора>:
    набор_инструкций_1
    if <условие_1>: break # выйти из цикла, минуя блок else
```

```
    if <условие_2>: continue # на следующую итерацию цикла
else:
    набор_инструкций_2 # выполняется, если выход из цикла
#производится без помощи инструкции break
```

Пример программы с применением операторов break и continue [1]:

```
while True:
    i = int(input())
    if i == 10: break # завершение цикла
    if i > 10:
        print(f"вы ввели число")
        continue
    print(f"{i} - это цифра")
```

Программа предлагает вводить числа. Если введена цифра (от 0 до 9), то программа сообщает, что введена цифра. Если введено число (больше 10), то программа сообщает, что введено число. Выход из цикла происходит при введении числа 10.

Будьте осторожны при использовании цикла `while`, так как с помощью него можно задавать бесконечный цикл.

```
while True:
    print(f"Нажми Ctrl-C чтобы остановить меня")
```

Цикл в данном примере может быть остановлен только при нажатии сочетания клавиш Ctrl-C, т.е. остановки интерпретатора.

4.2. Цикл for

С помощью инструкции *for*, как и с помощью инструкции *while,* можно организовать цикл. Следует отметить, что отличие инструкций заключается в особенностях их реализации, так циклы *for* в языке Python начинаются со строки заголовка, где указывается переменная цикла, а также объект или последовательность, обход которой будет выполнен. Вслед за заголовком следует блок инструкций, которые требуется выполнить [1].

```
for <имя_переменной in последовательность>: # связывает
                    # элементы объекта с переменной цикла
    набор_инструкций_1 # повторяющееся тело цикла,
                        # использует переменную цикла
else:
    набор_инструкций_2 # выполняется, если выход из цикла
                        #производится не инструкцией break
```

Когда интерпретатор выполняет цикл *for*, он поочередно, один за другим, присваивает элементы объекта последовательности переменной

цикла и выполняет тело цикла для каждого из них. Таким образом переменная цикла используется для обращения к текущему элементу последовательности в теле цикла.

Инструкция *for* также поддерживает необязательную часть *else,* которая работает точно так же, как и в циклах *while*. Инструкции *break* и *continue*, представленные в подразделе 4.1, в циклах *for* работают точно так же, как и в циклах *while*. Полная форма цикла *for* выглядит следующим образом:

```
for <имя_переменной in последовательность>: # связывает
                          #элементы объекта с переменной
                       цикла
    набор_инструкций_1 # повторяющееся тело цикла,
                         # использует переменную цикла
    if <условие_1>: break # выйти из цикла, минуя блок else
    if <условие_2>: continue # переход в начало цикла
else:
    набор_инструкций_2 # выполняется, если выход из цикла
                         #производится не инструкцией break
```

В приведенном ниже примере для цикла *for*, мы поочередно, слева направо, присваиваем переменной «*x*» один из четырех символов слова «word» и выводим каждый из них с помощью инструкции *print*.

```
for x in "word":
    print(x, end = " ")
```

Внутри инструкции *print,* или в теле цикла, переменная «*x*» ссылается на текущий элемент последовательности. В результате поочередно выводятся все элементы последовательности с разделителем «пробел».

4.3. Итерируемые объекты в Python

Итерируемый объект – это объект, который можно проитерировать, т.е. пройтись по элементам объекта в цикле. Ниже разберем некоторый типы данных Python, которые являются итерируемыми.

Списки

Список – это структура данных для хранения последовательности объектов различных типов. Список является изменяемым типом данных.

Списки создаются с помощью квадратных скобок `[]`, внутри которых через запятую перечисляются элементы списка. Ниже в примере определим список строк и список чисел [1]:

```
empty_list = [] # создание пустого списка
numbers = [1, 2, 3, 4, 5]
```

```
colors = ["красный", "желтый", "зеленый"]
```

Списки могут содержать данные разных типов, мы можем поместить в один и тот же список целые и дробные числа, строки и даже сами списки:

```
list_objects = [1, 9.8, "black", [1, 2, 3]]
```

Также для создания списка можно использовать функцию `list()`. Данная функция может принимать набор значений, на основе которых будет создан список. Ниже приведен пример создания списка из строки:

```
letters = list("красный")
print(letters) # ['к', 'р', 'а', 'с', 'н', 'ы', 'й']
```

Обращение к элементам списка

Для обращения к элементам списка необходимо использовать индексы, которые представляют номер элемента в списке. Индексы начинаются с нуля, то есть первый элемент будет иметь индекс 0, второй элемент – индекс 1 и так далее. Для обращения к элементам с конца списка можно использовать отрицательные индексы, начиная с -1. То есть у последнего элемента будет индекс -1, у предпоследнего -2 и так далее.

```
colors = ["красный", "желтый", "зеленый"]
print(colors [0])   # красный
print(colors [1])   # желтый
print(colors [-1])  # зеленый
```

Для изменения элемента списка достаточно присвоить ему новое значение:

```
colors = ["красный", "желтый", "зеленый"]
colors [0] = "серый"
print(colors) # ["серый", "желтый", "зеленый"]
```

Перебор элементов списка

Для перебора элементов списка можно использовать как цикл `for`, так и цикл `while`. Перебор с помощью цикла `for`:

```
colors = ["красный", "желтый", "зеленый"]
for person in colors:
    print(person)
```

Для перебора списка с помощью цикла `while`, воспользуемся функцией `len()`, которая возвращает длину списка. С помощью счетчика `i` будем выводить по элементу, пока значение счетчика не станет равно длине списка:

```
colors = ["красный", "желтый", "зеленый"]
i = 0
while i < len(colors):
    print(colors[i]) # Применяем индекс для получения
                          # элемента
    i += 1
```

Добавление и удаление элементов списка

Для добавления элемента применяются методы `append()`, `extend` и `insert`, а для удаления – методы `remove()`, `pop()` и `clear()` [2].

```
numbers = [1, 2, 3]
# добавляем в конец списка
numbers.append(4) # [1, 2, 3, 4]

# добавляем на позицию под индексом "1"
numbers.insert(1, 5) # [1, 5, 2, 3, 4]

# добавляем набор элементов [6,7]
numbers.extend([6,7]) # [1, 5, 2, 3, 4, 6, 7]

# получаем индекс элемента
ind = numbers.index(5)

# удаляем по этому индексу
removed_item = numbers.pop(ind) # [1, 2, 3, 4, 6, 7]

# удаляем последний элемент
last_item = numbers.pop() # [1, 2, 3, 4, 6]

# удаляем элемент "6"
numbers.remove(6) # [1, 2, 3, 4]

# удаляем все элементы
numbers.clear() # []
```

Копирование списков

В Python изменяемые объекты нельзя скопировать, присвоив одну переменной другой, так как в этом случае копируется ссылка на объект, а не он сам. В итоге при изменении объекта через одну переменную, изменения будут видны и в другой переменной:

```
    cities1 = ["Томск", "Омск", "Новосибирск"]
    cities2 = cities1
    # теперь cities1 и cities2 указывают на один и тот же
список

    cities2.append("Москва") # добавляем элемент во второй
список
```

```
    print(cities1) # ["Томск", "Омск", "Новосибирск",
"Москва"]
    print(cities1) # ["Томск", "Омск", "Новосибирск",
"Москва"]
```

Чтобы избежать этой ситуации, можно использовать встроенный метод `copy()`, в результате его использования будет происходить копирование элементов, но при этом переменные будут указывать на разные списки:

```
cities1 = ["Томск", "Омск", "Новосибирск"] # копируем
                          # элементы из cities1 в cities2
cities2 = cities1.copy() # теперь cities1 и cities2
                             #указывают на разные списки

cities2.append("Москва") # добавляем элемент во второй
                               #список
print(cities1) # ["Томск", "Омск", "Новосибирск"]
print(cities2) # ["Томск", "Омск", "Новосибирск", "Москва"]
```

Однако такой способ копирования является поверхностным и не подойдет если объект является составным, то есть включает другие изменяемые объекты, например, списки или словари, в этом случаи они не копируются, а копируются только ссылки на них:

```
    nums = [1, 2, 3]
    data = ['Томск', 'Омск', nums]
    print(data) # [ 'Томск', 'Омск', [1, 2, 3]]

    # копируем элементы из списка data в список
data_copy
    data_copy = data.copy()
    data[2].append(4) # добавляем элемент в список data

    print(data) # ['Томск', 'Омск', [1, 2, 3, 4]]
    print(data_copy) # ['Томск', 'Омск', [1, 2, 3, 4]]
```

Поэтому чтобы сделать полную копию составного объекта, следует воспользоваться функцией `deepcopy()` из модуля `copy`. Кроме этой функции там также есть функция `copy()`, выполняющая поверхностное копирование, аналогичное методу `copy()` используемому выше.

```
    import copy
    nums = [1, 2, 3]
    data = ['Томск', 'Омск', nums]
    data_copy = copy.deepcopy(data)
    data[2].append(4)
    print(data) # ['Томск', 'Омск', [1, 2, 3, 4]]
    print(data_copy) # ['Томск', 'Омск', [1, 2, 3]]
```

Кортежи

Кортеж представляет собой последовательность элементов, которая во многом похожа на список. Так же, как и список, кортеж может состоять из элементов разных типов, перечисленных через запятую. Основное отличие кортежа от списка – это то, что кортеж является **неизменяемым** типом. Это значит, что мы не можем добавлять, удалять или поменять значение элементов в кортеже.

Для создания кортежа используются круглые скобки, в которые помещаются его значения, разделённые запятыми [2]:

```
user = ("Иванов Е.С.", 35)
print(user) # ("Иванов Е.С.", 35)

# Определять кортеж можно и без использования скобок
user = "Иванов Е.С.", 35
print(user) # ("Иванов Е.С.", 35)

# Если кортеж состоит из одного элемента, то в конце
ставится запятая
user = ("Иванов Е.С.",)
```

Для создания кортежа из других типов данных, например, из списка, можно воспользоваться функцией `tuple()`:

```
data = ["Иванов Е.С.", 35, "Томск"]
user = tuple(data)
print(user) # ("Иванов Е.С.", 35, "Томск")
```

Обращение к элементам кортежа

Обращение к элементам в кортеже происходит также, как и в списке, по индексу. Индексация начинается с нуля при получении элементов из начала списка и с [-1] – из конца списка:

```
user = ("Иванов Е.С.", 35, "Томск")
print(user[0]) # Иванов Е.С.
print(user[1]) # 35
print(user[-1]) # Томск
```

Так как кортеж является неизменяемым типом данных, то мы не сможем изменить его элементы после создания, в отличие от работы со списками. То есть следующая запись работать не будет:

```
user[1] = "Сидоров А.А." # Будет выведено сообщение об
ошибке
```

Перебор кортежей

Для перебора кортежа можно использовать стандартные циклы `for` и `while`:

```
# С помощью цикла for
user = ("Иванов Е.С.", 35, "Томск")
for item in user:
    print(item)

# С помощью цикла while
i = 0
while i < len(user):
    print(tom[i])
    i += 1
```

Строки

Строка – это тип данных, предназначенный для работы с текстом. Чтобы создать строку в Python, можно использовать как одинарные, так и двойные кавычки. Для многострочных строк нужно использовать тройные кавычки:

```
example_1 = 'Привет, мир!'
example_2= "Привет, мир!"
example_3= """У лукоморья дуб зелёный,
златая цепь на дубе том."""
```

Строка состоит из последовательности символов. Узнать количество символов в строке можно при помощи функции `len`:

```
s = 'Привет, мир!'
print(len(s)) # Результат вывода -> 12
```

Обращение к элементам строки по индексу

Обратиться к отдельным элементам строки можно по индексу, который размещается внутри квадратных скобок:

```
string = "Привет, мир"
s1 = string[0]
print(s1) # Результат вывода -> П

s6 = string[5]
print(s6) # Результат вывода -> т

s11 = string[11]
# Ошибка -> IndexError: string index out of range
```

Учитывая, что индексация начинается с нуля, то первый символ строки будет иметь индекс 0. Если мы попытаемся обратиться к индексу, которого

нет в строке, то мы получим исключение `IndexError`. Например, в случае выше длина строки 11 символов, поэтому ее символы будут иметь индексы от 0 до 10. Чтобы получить доступ к символам, начиная с конца строки, можно использовать отрицательные индексы. Так, индекс -1 будет представлять последний символ, а -2 предпоследний символ и так далее.

При работе со строками нужно учитывать, что они являются неизменяемым типом, поэтому попытка изменить какой-то отдельный символ строки приведет к ошибке, как в примере ниже:

```
string = "Привет, мир"
string[6] = "!" # Будет выведено сообщение об ошибке
```

Нельзя изменить отдельный символ строки, можно только полностью переустановить значение строки, присвоив ей другое значение.

Получение подстроки

При необходимости мы можем получить из строки не только отдельные символы, но и подстроку. Для этого используется следующий синтаксис:

- `string[:end]` - извлекает последовательность символов начиная с 0- го индекса по индекс `end` не включая его;
- `string[start:end]` - извлекает последовательность символов начиная с индекса `start` по индекс `end` не включая его;
- `string[start:end:step]` - извлекает последовательность символов начиная с индекса `start` по индекс `end` не включая его с шагом `step`.

Ниже приведены различные варианты получения подстрок:

```
string = "Привет, мир"

# с 0 до 5 индекса
sub_string1 = string[:5]
print(sub_string1) # Привет

# со 8 до 10 индекса
sub_string2 = string[8:10]
print(sub_string2) # мир
```

Перебор строки

Так как строки в Python являются итерируемыми объектами, то мы можем перебрать все элементы строки с помощью цикла, например, `for`:

```
string = "Привет, мир"
for char in string:
```

```
print(char)
```

Объединение строк

Одной из самых распространенных операций со строками является их объединение или конкатенация. Для объединения строк применяется операция сложения:

```
name = "Иван"
surname = "Иванов"
fullname = name + " " + surname
print(fullname) # Иван Иванов
```

Однако если нам требуется сложить строку и число, то предварительно необходимо привести число к строке с помощью функции `str()`:

```
name = "Иван"
age = 35
info = "Имя: " + name + " Возраст: " + str(age)
print(info)  # Имя: Иван Возраст: 35
```

Функцию `str()` можно использовать для приведения к строке других типов данных, для этого нужно вызвать функцию `str()`, передав ей в качестве параметра объект, переводимый в строку.

Словари

Словари в Python – неупорядоченные коллекции произвольных элементов с доступом по ключу. Словарь хранит коллекцию элементов, где каждый элемент имеет уникальный ключ и связанное с ним значение. Определение словаря имеет следующий вид:

```
dictionary = { ключ1: значение1,
               ключ2: значение2,
               ...}
```

В фигурных скобках через запятую определяется последовательность элементов, где для каждого элемента указывается ключ и его значение:

```
user = {} # Определение пустого словаря
user = {"name": "Иванов Е.С.", "age": 35}
```

В словаре `user` в качестве ключей и значений используются строки. То есть элемент с ключом `"name"` имеет значение "Иванов Е.С.", а элемент с ключом `"age"` – значение 35. Однако, ключи необязательно должны быть строкового типа, они могу представлять разные типы как в примере ниже:

```
user = {1: "Иванов Е.С.", "age": 35}
```

Получение и изменение элементов словаря

Для обращения к элементам словаря необходимо после его названия указать в квадратных скобках ключ элемента. Ниже приведен пример получения и изменения элементов словаря:

```
user = {
    "name": "Иванов Е.С.",
    "age": 35}

# Получение элемента с ключом "name"
print(user["name"]) # Иванов Е.С.

# Установка значения элемента с ключом "name"
user["name"] = "Сидоров А.А."
print(user["name"]) # Сидоров А.А.
```

Если при установке значения в словаре не окажется элемента с таким ключом, то произойдёт его добавление:

```
user["city"] = "Tomsk"
prtint(user["city"])  # Tomsk
```

Необходимо учитывать, что при попытки получить значение по ключу, которого нет в словаре, Python сгенерирует ошибку:

```
# Будет выведено сообщение об ошибке KeyError
user_profession = user["profession"]
```

Для того чтобы избежать этой ситуации перед обращением к элементу, необходимо проверять наличие ключа в словаре с помощью выражения «ключ» in «словарь». Если ключ имеется в словаре, то данное выражение вернет `True`:

```
key = "profession"
if key in user:
    user_profession = user[key]
    print(user_profession)
else:
    print("Элемент с таким ключем отсутствует в
словаре")
```

Для получения элементов словаря можно использовать метод `get`, который имеет две формы:

- `get(key)`: возвращает из словаря элемент с ключом `key`. Если

элемента с таким ключом нет, то возвращает значение None;

- get(key, default): возвращает из словаря элемент с ключом key. Если элемента с таким ключом нет, то возвращает значение по умолчанию default.

```
user = { "name": "Иванов Е.С.", "age": 35}

user_name = users.get("name")
print(user_name) # Иванов Е.С.

user_age = users.get("age", "Значение не найдено")
print(user_age) # 35

user_city = users.get("city", "Значение не найдено")
print(user_city) # Значение не найдено
```

Перебор словаря

Для перебора словаря можно воспользоваться циклом for. При переборе элементов мы получаем ключ текущего элемента и по нему можем получить сам элемент:

```
users = {1: "Иванов Е.С.", 2: "Петров А.С."}
for key in user:
    print(f"Number: {key}  User: {users[key]} ")
```

Другой способ перебора элементов реализуется с помощью метода items(), который возвращает набор кортежей. Каждый кортеж содержит ключ и значение, которые мы можем получить в переменные key и value:

```
users = {1: "Иванов Е.С.", 2: "Петров А.С."}
for key, value in users.items():
    print(f"Number: {key}  User: {value} ")
```

Также существуют возможности перебора ключей и значений. Для перебора ключей используется метод словаря keys(), для перебора значений метод values():

```
for key in users.keys():
    print(key)

for value in users.values():
    print(value)
```

Удаление элементов словаря

Для удаление элементов словаря можно использовать метод pop(), он имеет два возможных варианта использования:

- pop(key): удаляет элемент по ключу key и возвращает удаленный

элемент. Если элемент с данным ключом отсутствует, то генерируется исключение KeyError

- `pop(key, default)`: удаляет элемент по ключу `key` и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то возвращается значение `default`

Ниже приведён пример использования метода `pop()` для удаления элементов словаря:

```
user = { "name": "Иванов Е.С.", "age": 35}
key = "name"
user_name = user.pop(key)
print(user_name) # Иванов Е.С.

user_city = users.pop("city", "Значение
отсутствует")
print(user) # Значение отсутствует
```

4.4. Контрольные вопросы для самопроверки

1. Что такое циклы?
2. Какие синтаксические конструкции используются для задания циклов? Приведите блок-схему цикла.
3. Чем отличаются конструкции `while` и `for`. Привести пример.
4. Назовите типы последовательностей языка Python и расскажите об особенностях каждой из них.
5. Опишите тип данных списки. Являются ли списки изменяемыми типами данных или нет. Что это значит?
6. Опишите тип данных кортежи. Являются ли кортежи изменяемыми типами данных или нет. Что это значит?
7. Опишите тип данных строки. Являются ли строки изменяемыми типами данных или нет. Что это значит?
8. Опишите тип данных словари. Являются ли словари изменяемыми типами данных? Что это значит?
9. Как производится копирование списков?
10. Как производится копирование словарей?
11. Как производится копирование строк?
12. Поясните, отличие функций copy() и deepcopy() ?

5. Работа с файлами. Создание функций.

5.1. Работа с текстовыми файлами

Для создания файла необходимо выполнить команду:

```
f = open ("file1.txt", "w +")
```

В приведенной выше строке кода объявляется переменная «f», которая указывает на данные считанные из файла – «file1.txt», с помощью функции *open()*. Функция *open()* принимает два аргумента: путь к файлу, который мы хотим открыть, и строку, представляющую виды разрешений или операций, которые мы хотим выполнить над файлом.

Возможные варианты второго аргумента [6]:

•«r» – Открыть текстовый файл для чтения. Поток расположен в начале файла.

•«r+» – Открыт для чтения и записи. Поток расположен в начале файла.

•«w» – Очистить файл (для перезаписи) или создать новый текстовый файл для записи. Поток располагается в начале файла.

•«w+» – Открыт для чтения и записи. Файл создается, если он не существует, в противном случае он затирается. Поток расположен в начале файла.

•«a» – Открыта для записи. Файл создается, если он не существует. Поток расположен в конце файла. Последующие записи к файлу всегда будет в конце текущего конца файла.

•«a+» – Файл открыт для чтения и письма, если файл не существует, то он создается. Поток располагается в конце файла.

После завершения работы с файлом его необходимо закрыть, чтобы освободить файл для работы с другими программами и/или потоками:

```
f.close()
```

В следующем примере, в цикле, в файл записываются строки «This is line 1,2…, *n*»:

```
for i in range(10):
    f.write(f"This is line {i+1} \n")
f.close()
```

В цикле *for* задается диапазон чисел [0, 10), т.е. не включая 10. Для записи данных в файл используется функция *write*(). В каждой строке в файл записывается строка: «This is a line №». После того как файл создан,

необходимо освободить память, закрыв файл, с помощью инструкции *f.close()*. После этого файл можно открыть в любом текстовом редакторе, например, в «Блокноте» (рис. 5.1).

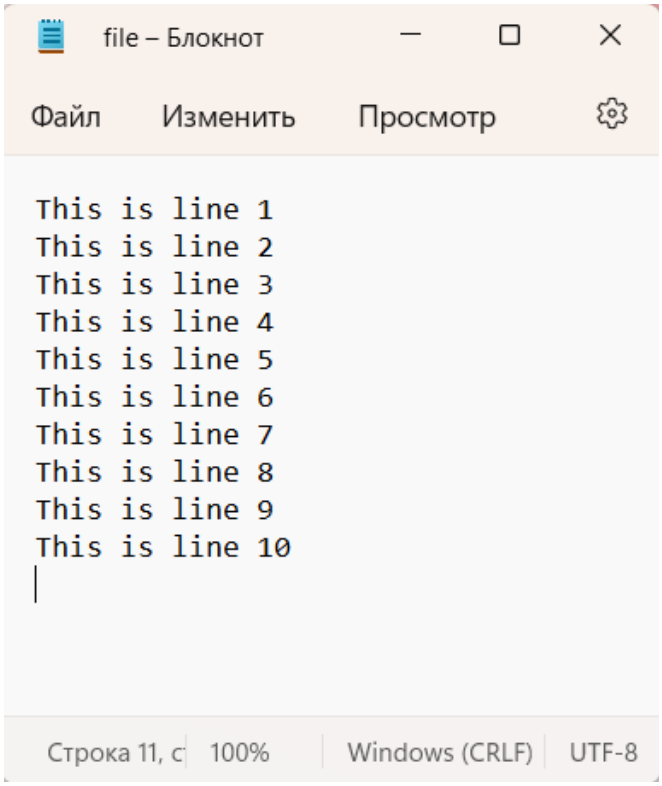


Рис. 5.1. Просмотр созданного текстового файла

Если вы попытаетесь записать в файл просто созданный список, то он запишется как строковые данные. Впоследствии это приведет к затруднению считывания данных в числовом формате. Поэтому лучше указывать тип данных при записи в файл. Уточним, как работает типизированный ввод данных:

```
    pupil = "Ben"
    old = 16
    grade = 9.2

    f.write (f"It's {pupil}, {old}. Level: {grade}")
    # f.write ("It's %s, %d. Level: %f" % (pupil, old,
grade))

    # Результат вывода:
    # It's Ben, 16. Level: 9.200000
```

В вышеприведенном коде создаются три переменные строкового, целого и вещественного типа. Для записи их в файл необходимо предварительно

указать, в каком порядке и какого типа переменные будут сохраняться `"It's %s, %d. Level: %f",` а затем перечислить сами переменные. Таким образом буквы s, d, f обозначают типы данных – строку, целое число и вещественное число, соответственно.

Для открытия файла в режиме чтения необходимо выполнить команду:

```
f = open ("file1.txt", "r")
```

При необходимости, с помощью функции `mode()`, можно определить, находится ли файл в режиме чтения. Если да, выполняем необходимые команды.

Чтобы прочитать данные из файла и сохранить их в переменной следует воспользоваться функцией `read()`.

```
f = open("file.txt", "r")
if f.mode == 'r':
    contents = f.read()
    print (contents)
f.close()
```

В приведенном коде, с помощью функции `read()` считывается информация из файла, на которую указывает переменная `contents`.

Для того, чтобы получить из строки список отдельных элементов, можно воспользоваться функцией split(“arg”), где arg – символ разбиения (рис. 5.2):

```
with open('file.txt','r') as f:
      c = f.read().split(' ')
      print(c)
```

```
['This', 'is', 'line', '1', '\nThis', 'is', 'line', '2', '\nThis', 'is', 'line
', '3', '\nThis', 'is', 'line', '4', '\nThis', 'is', 'line', '5', '\nThis', 'is
', 'line', '6', '\nThis', 'is', 'line', '7', '\nThis', 'is', 'line', '8', '\nTh
is', 'is', 'line', '9', '\nThis', 'is', 'line', '10', '\n']
```

Рис. 5.2. Пример работы функции split()

Следует отметить, что функции split(), может быть передан любой символьный знак, в качестве разделителя.

Второй вариант считывания файла – по строкам, для это применяется функция *readlines()*.

```
f = open("file.txt", "r")
fl = f.readlines()
for x in fl:
    print(x)
f.close()
```

5.2. Работа с файлами *.xlsx

Для работы с файлами Excel в Python можно использовать библиотеку XlsxWriter.

XlsxWriter – это модуль Python, который можно использовать для записи текста, чисел, формул и гиперссылок на несколько листов в файле Excel. Перед началом работы библиотеку необходимо установить с помощью команды ниже [7]:

```
pip install XlsxWriter
```

Далее разберем по шагам основные этапы работы с библиотекой. Первый шаг – импорт модуля:

```
import xlsxwriter
```

Следующим шагом является создание нового объекта рабочей книги с помощью конструктора `Workbook()`, принимающего один необязательный аргумент, который является именем создаваемого файла:

```
workbook = xlsxwriter.Workbook('example.xlsx')
```

Затем объект рабочей книги используется для добавления нового рабочего листа с помощью метода «add_worksheet()»:

```
worksheet = workbook.add_worksheet()
```

Далее зададим номер строки и ячейки, с которой начнется запись. В XlsxWriter строки и столбцы имеют нулевую индексацию, следовательно, первая ячейка рабочего листа A1– это (0, 0):

```
row = 0
col = 0
```

После этого необходимо записать данные в созданный файл Excel, для этого перебираем их в цикле и записываем построчно в таблицу с помощью метода `write(row, col, data)`:

```
data = (
    ['Иванов Е.С.', 35],
    ['Сидоров А.А.', 40],
    ['Петров А.С.', 29],)

for item in (data):
    worksheet.write(row, col, item[0])
    worksheet.write(row, col + 1, item[1])
    row += 1
```

В конце, закрываем файл Excel с помощью метода `close()`:

```
workbook.close()
```

Если запустить программу, мы получим таблицу Excel, которая выглядит, как показано на рис. 5.3.

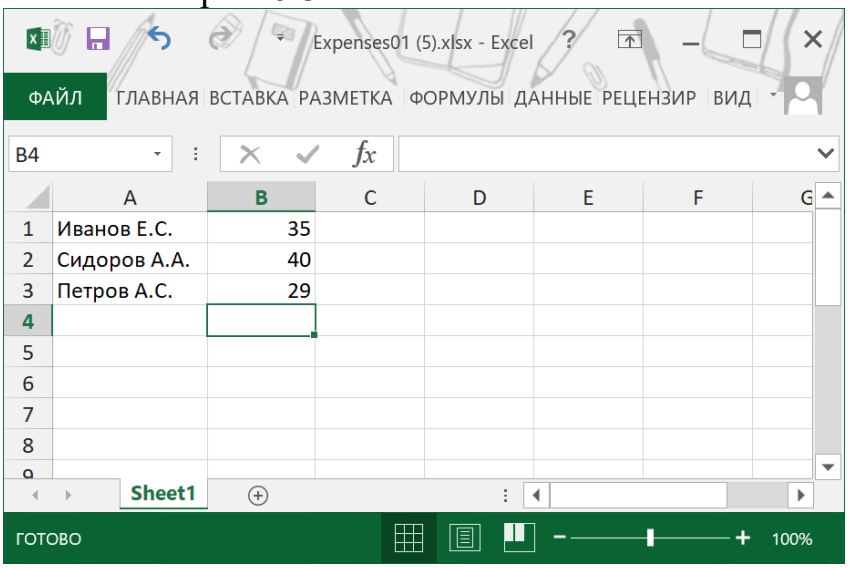


Рис. 5.3. Результат работы программы

5.3. Создание собственных функций в Python

Функции представляют собой блок кода, который выполняет определенную задачу и может быть вызван из других частей программы. Формальное определение функции представлено ниже [1]:

```
def имя_функции ([параметры]):
    инструкции
```

Определение функции начинается с выражения `def`, далее следует имя функции и передаваемые параметры в круглых скобках. Правила для записи имени функции такие: имя функции может содержать строчные буквы английского алфавита, цифры и знаки подчёркивания. Аргументы функции – это её параметры, которые становятся переменными в теле функции. Следует отметить, что параметры являются необязательными к заполнению, то есть можно определить функцию и без параметров.

После объявления функции идет блок инструкций, которые выполняет функция. Все инструкции функции должны иметь отступы от начала строки. Ниже приведен пример определения простой функции:

```
def print_hello():
    print("Привет")
```

Приведенная выше функция называется `print_hello`. Она не имеет параметров и содержит одну единственную инструкцию, которая выводит в консоль строку «Привет». Для вызова функции необходимо указать имя функции, после которого в скобках идет передача значений для всех ее параметров, если они есть, формально вызов функции выглядит следующим образом:

```
имя_функции ([параметры])
```

Для более ясного понимания определим функцию `print_hello()` еще раз и вызовем ее, как это может выглядеть в реальном коде:

```
# Определение функции print_hello
def print_hello():
    print("Привет")

# Вызов функции print _hello
print_hello()
```

В результате вызова функции в консоль будет выведено слово «Привет».

Через параметры в функцию можно передавать любые данные из основной программы. Самый простой пример – функция `print(name)`, которая с помощью параметра принимает значение, выводимое на консоль.

В качестве примера, изменим функцию `print_hello()` таким образом, чтобы она принимала одни параметр и вызовем измененную функцию:

```
def print_hello(name):
    print(f"Привет, {name}!")

print_hello("Андрей") # Результат вывода -> Привет, Андрей!
print _hello("Сергей") # Результат вывода -> Привет Сергей!
```

Теперь функция `print_hello` имеет параметр `name`, и при вызове функции мы можем передать этому параметру какое-либо значение. Внутри функции параметр может быть использован как обычная переменная, например, можно вывести его значение на консоль с помощью функции `print`.

Функция может возвращать результат. Для этого в функции используется оператор `return`, после которого указывается возвращаемое значение:

```
def имя_функции ([параметры]):
    инструкции
    return возвращаемое_значение
```

Таким образом тело функции содержит код, который работает с параметрами и внешними переменными, а затем возвращает результат с помощью оператора return. При возврате значения функция прекращает свою

работу, а интерпретатор продолжает работу программы, подставив на место вызова функции возвращённое значение.

Далее приведем пример простой функции, которая возвращает значение:

```
def get_sum(a, b):
    return a + b
```

Здесь оператор `return` возвращает результат сложения двух переменных – входных параметров `a` и `b` функции `get_sum`, это значение и будет возвращать функция `get_sum()`. Затем результат функции можно присвоить переменной или использовать как обычное значение:

```
def get_sum(a, b):
    return a + b

# получаем результат функции get_sum в переменную sum
sum = get_sum(1, 2)
print(sum) # Результат вывода -> 3

# можно напрямую передать результат функции get_sum
print(get_sum(1, 2)) # Результат вывода -> 3
```

Области видимости

Функции могут работать с параметрами и возвращать значения, при этом важно понимать, что значения параметров, которые определены внутри функции, доступны только внутри этой функции. Покажем это на примере реализации функции, которая проверяет состоит ли переданный ей список только из чётных значений:

```
def only_even(numbers):
    n = numbers
    for i, x in enumerate(n):
        if x % 2 != 0:
            return False, i
    return True

print(n)  # обращение к локальной переменной функции
```

Вывод:
NameError: name 'n' is not defined

В приведенном выше коде, функция принимает в качестве параметра переменную `numbers`, затем для того, чтобы продемонстрировать, что такое локальная область видимости, мы введем еще одну переменную, которая будет соответствовать переменной `numbers`, в реальной работе такой прием

делать необязательно. Теперь если мы за пределами функции `only_even()` обратимся к переменной `n` – попытаемся вывести ее значение с помощью функции `print()` на экран, то интерпретатор выведет сообщение, что такая переменная не определена. Это произошло, потому что параметр функции – переменная `n`, является локальной переменной, то есть она существует только во время выполнения функции и доступна только внутри неё.

Из приведенного примера следует, что параметры, определенные внутри функции, находятся в локальной области её видимости.

Существует и глобальная область видимости. Переменные, созданные вне функций, то есть в основном коде программы, находятся в глобальной области видимости. Это означает, что к ним можно получить доступ из любой части программы.

В следующем примере, в функции проверки пароля `check_password()` используется переменная `password`, которая определена за пределами функции – в глобальной области видимости, по отношению к функции:

```
def check_password(pwd):
    return pwd == password

password = "Python"
print(check_password("Python_123"))
```

Вывод: False

Если в функции необходимо менять значения переменных из глобальной области видимости путём операции присваивания, то нужно в теле функции сделать эти переменные глобальными с помощью ключевого слова *global*. Тогда в функции не будут создаваться локальные переменные с такими же именами, а значение поменяется в глобальной переменной:

```
def inc():
    global x
    x += 1
    print(f"Количество вызовов функции равно {x}.")


x = 0
inc()
inc()
inc()
print(f"Количество вызовов функции равно {x}.")

Количество вызовов функции равно 1.
Количество вызовов функции равно 2.
```

```
Количество вызовов функции равно 3.
Количество вызовов функции равно 3.
```

Важно понимать, что использование глобальных переменных будет приводить к усложнению интерпретации кода, так как изменить её значение можно в любой части программы. Поэтому использовать глобальные переменные рекомендуется только в случае, если это действительно необходимо.

Приведенный выше пример кода можно изменить таким образом, чтобы функция не использовала глобальных переменных, для этого достаточно передавать в функцию значение, полученное с предыдущего вызова функции.

```
def f(count):
    count += 1
    print(f'Количество вызовов функции равно {count}.')
    return count


count_f = 0
count_f = f(count_f)
count_f = f(count_f)
count_f = f(count_f)

Количество вызовов функции равно 1.
Количество вызовов функции равно 2.
Количество вызовов функции равно 3.
```

5.4. Контрольные вопросы для самопроверки

1. Расскажите, как можно создать обычный текстовый файл с помощью языка Python.

2. Расскажите, как с помощью библиотеки «xlsxwriter» создать файл и особенности работы с ним, средствами этой библиотеки.

3. Дайте определение функции.

4. В чём отличие определения функции от её вызова?

5. Как синтаксически задаются функции в Python?

10. Расскажите об особенностях создания и вызова собственных функций в языке Python.

11. Опишите работу с локальной и глобальной областью видимости параметров функции.

6. Объектно-ориентированное программирование в Python

6.1. Классы

Классы – это основные инструменты объектно-ориентированного программирования (ООП) в языке Python. ООП является более эффективным подходом к программированию, чем процедурное программирование.

Классы в языке Python создаются с помощью инструкции *class* [8]:

```
class ИмяКласса:
    код_тела_класса
```

После инструкции *class* должно следовать имя класса, которое должно начинаться, в соответствии со стандартом PEP-8, с большой буквы. Далее с новой строки начинается тело класса с отступом в четыре пробельных символа.

```
class Animals:
    # атрибуты класса
    claws = True
    legs_num = 4

    # методы класса
    def fight_for_food(self, tasty):
        if tasty:
            print(f"Добыча поймана")
    def animal_name(self, name):
        self.name = name
```

В приведенном выше коде создаётся класс *Animals* с атрибутами *claws, legs_num* и методами *fight_for_food(), animal_name().* Упрощая, атрибуты класса по сути являются его переменными, а методы класса его функциями.

6.2. Экземпляр класса

Класс можно сравнить с чертежом, на основе которого создаётся некоторый объект или экземпляр класса. Для создания экземпляра класса необходимо присвоить имя класса некоторой переменной, которая будет содержать ссылку на созданный объект [8].

Для примера создадим два экземпляра класса *Animals* из раздела 6.1:

```
a = Animals() # создаем a – экземпляр класса Animals
b = Animals() # создаем b – экземпляр класса Animals
a.fight_for_food(True) () # вызываем метод
                            fight_for_food()
a.animal_name("Fluffy")
```

```
b.animal_name("Spotty")

print(a.name)
print(b.name)

результат работы программы:
Добыча поймана
Fluffy
Spotty
```

Следует обратить внимание на результат работы программы, несмотря на то, что метод *animal_name()* в классе *Animals* один, а переданные параметры для каждого экземпляра класса были разные, переменная *name* у каждого экземпляра класса указывает только на присвоенное ей значение. Чтобы объяснить причину такого поведения, нужно разобраться с переменной self.

В разделе 6.1 представлен класс *Animals* с атрибутами *claws, legs_num* и методами *fight_for_food(), animal_name()*. В методах *fight_for_food()* и *animal_name()* используется переменная *self*, которая, определяет ссылку на объект или экземпляр класса. Переменная *self* нужна для связи с объектом класса, для которого вызываются методы класса и с помощью переменной *self* можно получить доступ к атрибутам объекта. Так создавая несколько объектов или экземпляров класса, каждый созданный экземпляр будет обладать своими методами и атрибутами, следовательно, данные у каждого экземпляра класса будут свои.

6.3. Конструктор класса

В объектно-ориентированном программировании конструктором класса называют метод, который автоматически вызывается при создании объектов. Его также можно назвать конструктором объектов класса. Имя такого метода обычно регламентируется синтаксисом конкретного языка программирования. В Python роль конструктора играет метод __init__() [8]:

```
class Person:
    def __init__(self, n, s):
        self.name = n
        self.surname = s

p1 = Person("Sam", "Baker")
print(p1.name, p1.surname)
```

```
результат работы программы:
Sam Baker
```

В Python наличие пар знаков подчеркивания спереди и сзади в имени метода означает, что его принадлежность к группе методов перегрузки операторов. Если подобные методы определены в классе, то объекты могут участвовать в таких операциях как сложение, вычитание, вызываться как функции и др.

При этом методы перегрузки операторов не надо вызывать по имени. Вызовом для них является сам факт участия объекта в определенной операции. В случае конструктора класса – это операция создания объекта. Так как объект создается в момент вызова класса по имени, то в этот момент вызывается метод __init__(), если он определен в классе.

Необходимость конструкторов связана с тем, что нередко объекты должны иметь собственные свойства сразу.

6.4. Наследование

Наследование – важная составляющая объектно-ориентированного программирования. Так или иначе мы уже сталкивались с ним, ведь объекты наследуют атрибуты своих классов. Однако обычно под наследованием в ООП понимается наличие классов и подклассов. Также их называют супер- или надклассами и классами, а также родительскими и дочерними классами. Суть наследования здесь схожа с наследованием объектами от классов. Дочерние классы наследуют атрибуты родительских, а также могут переопределять атрибуты и добавлять свои.

В качестве примера рассмотрим разработку класса столов и его двух подклассов – кухонных и письменных столов. Все столы, независимо от своего типа, имеют длину, ширину и высоту. Пусть для письменных столов важна площадь поверхности, а для кухонных – количество посадочных мест. Общее вынесем в класс, частное – в подклассы. Связь между родительским и дочерним классом устанавливается через дочерний: родительские классы перечисляются в скобках после его имени [8]:

```
class Table:
    def __init__(self, l, w, h):
        self.lenght = l
        self.width = w
        self.height = h
```

```
class KitchenTable(Table):
    def setPlaces(self, p):
        self.places = p

class DeskTable(Table):
    def square(self):
        return self.width * self.length
```

В данном случае классы *KitchenTable* и *DeskTable* не имеют своих собственных конструкторов, поэтому наследуют его от родительского класса.

6.5. Инкапсуляция

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними. В ряде других языков, например, в Java, под инкапсуляцией также понимают сокрытие свойств и методов, в результате чего они становятся приватными. Это значит, что доступ к ним ограничен либо пределами класса, либо модуля. Другими словами, в Python подобной инкапсуляции нет, хотя существует способ ее имитировать.

Для имитации сокрытия атрибутов в Python используется соглашение, соглашение – это не синтаксическое правило языка, при желании его можно нарушить, согласно которому, если поле или метод имеют два знака подчеркивания впереди имени, но не сзади, то этот атрибут предусмотрен исключительно для внутреннего пользования [8]:

```
class B:
    __count = 0
    def __init__(self):
        B.__count += 1
    def __del__(self):
        B.__count -= 1

a = B()
print(B.__count)
```

Попытка выполнить этот код приведет к возникновению исключения:

```
    File "test.py", line 9, in <module>
            print(B.__count)
    AttributeError: type object 'B' has no attribute
'__count'
```

То есть атрибут *__count* за пределами класса становится невидимым, хотя внутри класса он вполне себе видимый. Понятно, если мы не можем даже получить значение поля за пределами класса, то присвоить ему значение – тем более.

На самом деле сокрытие в Python не настоящее и доступ к счетчику мы получить все же можем. Но для этого надо написать *B._B__count,* таково соглашение:

```
print(B._B__count)
```

Если в классе есть атрибут с двумя первыми подчеркиваниями, то для доступа извне к имени атрибута добавляется имя класса с одним впереди стоящим подчеркиванием. В результате атрибут как он есть, в данном случае *__count*, оказывается замаскированным. Вне класса такого атрибута просто не существует. Для программиста же наличие двух подчеркиваний перед атрибутом должно сигнализировать, что трогать его вне класса не стоит вообще, даже через *_B__count*, разве что при крайней необходимости.

С помощью двойного подчеркивания обозначено, что поле *__count* защищено, осталось разобраться как получит доступ к его значению. Сделать это можно с помощью создания дополнительного метода, как показано в примере [8]:

```
class B:
    __count = 0
    def __init__(self):
        B.__count += 1
    def __del__(self):
        B.__count -= 1
    def qtyObject():
        return B.__count

a = B()
b = B()
print(B.qtyObject()) # будет выведено 2
```

В данном случае метод *qtyObject()* не принимает объект нет ключевого слова *self*, поэтому вызывать его надо через класс.

Следует отметить, что существует также и одиночное подчеркивание в начале имени атрибута, которое означает, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени, в отличии от двойного подчеркивания.

```
class A:
```

```
    def _private(self):
        print("Это приватный метод!")

a = A()
a._private()
```

6.6. Контрольные вопросы для самопроверки

1. Расскажите, что такое объектно-ориентированное программирование.
2. Что такое экземпляр класса и как его создать?
3. Как создать дочерний класс?
4. Что такое инкапсуляция?

7. Реализация простой базы данных средствами языка Python

7.1. История реляционных баз данных

Принципы реляционной модели были сформулированы в 1969—1970 годах Э. Ф. Коддом (E. F. Codd). Идеи Кодда были впервые публично изложены в статье «A Relational Model of Data for Large Shared Data Banks», ставшей классической.

В 1970-м его работа представляла лишь академический интерес. Однако в 1979 компания Oracle создала первую реляционную СУБД. А затем и другие компании стали использовать реляционные модели данных, в настоящее время это самый распространенный вид базы данных.

Долгое время на смену реляционным базам данных пророчили приход XML и объектно-ориентированных баз данных (ООБД). Однако эти технологии так и не стали массовыми, хотя продукты существуют и по сей день. XML вышел из моды из-за своей избыточности. ООБД, которые решают проблему несовместимости реляционных баз данных, основанных на множествах, и объектно-ориентированных программ, основанных на деревьях, тоже не стали слишком популярны.

7.2. Реляционная модель данных

Реляционная модель основывается на математических принципах, вытекающих непосредственно из теории множеств и логики предикатов. Эти принципы впервые были применены в области моделирования данных в конце 60-х гг. Е.Ф. Коддом, в то время работавшим в IBM, а впервые опубликованы — в 1970 г.

Реляционная модель определяет способ представления данных т.е. структуру данных, методы зашиты данных т.е. целостность данных, а также операции, выполняемые с данными т.е. манипулирование данными. Реляционная модель не единственный метод хранения и манипулирования данными. Существуют альтернативные варианты: иерархическая, сетевая, а также звездообразная модели данных. У каждой из них свои преимущества при решении задач определенного типа. Однако гибкость и эффективность реляционной модели делают ее наиболее популярным инструментом для разработки баз данных.

В общих чертах основные принципы реляционных систем баз данных можно сформулировать так:

- Все данные на концептуальном уровне представляются в виде упорядоченной организации, определенной в виде строк и столбцов и называемой *отношением.*

- Все значения являются *скалярами.* Это означает, что для любой строки и столбца любого отношения существует одно и только одно значение.
- Все операции выполняются над целым отношением, и результатом выполнения этих операций также является целое отношение. Этот принцип называется замыканием.

Принцип замыкания заключается в том, что и базовые таблицы, и результаты операций над ними на концептуальном уровне представляются как отношения. Он позволяет непосредственно использовать результаты одной операции в качестве исходных данных для выполнения другой. Этот принцип реализует в области разработки баз данных функциональность, аналогичную подпрограммам в процедурном программировании — возможность инкапсуляции сложных или часто повторяющихся операций для повторного использования.

Формулируя принципы реляционной модели, Кодд выбрал термин «отношение» (relation), потому что он однозначен в то время, как, например, термин «таблица» имеет множество дополнительных значений. Весьма распространено следующее заблуждение: реляционная модель названа так потому, что она определяет отношения между таблицами. На самом деле таблица, в которой все строки отличаются друг от друга, в математических терминах называется отношением (relation). Именно этому термину реляционные базы данных и обязаны своим названием, поскольку в их основе лежат отношения, т. е., таблицы с отличающимися друг от друга строками. В рамках реляционной модели данные представлены в виде отношения на концептуальном уровне; однако при этом совсем не дается никаких указаний, каким образом данные будут реализованы на физическом уровне.

7.3. Первичный и внешний ключи

Поскольку строки в реляционной таблице не упорядочены, нельзя выбрать строку по ее номеру в таблице. В таблице нет первой, последней или тринадцатой строки. В правильно построенной реляционной базе данных в каждой таблице есть столбец (или комбинация столбцов), для которого значения во всех строках различны. Этот столбец (или столбцы) называется первичным ключом (primary key) таблицы. Первичный ключ у каждой строки уникальный. В таблице с первичным ключом нет двух совершенно одинаковых строк [9].

Первичный ключ - уникально идентифицирует строку данных. Первичный ключ может быть простым или составным. Если первичный ключ состоит из нескольких атрибутов, он является составным.

Внешний ключ является ключом – ссылкой, связью между таблицами. Внешние ключи используются для проверки реляционной базы данных на целостность. Если атрибут, на который указывает внешний ключ, не будет присутствовать в таблице, на которую он указывает, это приведет к ошибке.

7.4. Типы связей в реляционных базах данных

Всего существует 3 типа связей [9]:

- Один к одному;
- Один ко многим;
- Многие ко многим.

Связь "Один к одному"

Один к одному - у каждой из двух сущностей есть лишь один спутник и больше никто.

***Например**:* В базе данных университета есть таблица с информацией о студентах (напр. паспортные данные) и таблицы профилей этих студентов на университетском сайте, где тоже есть несколько колонок, заполняемых по желанию). Если один студент может завести только один аккаунт - то мы имеем классический пример связи один к одному.

Связь "Один ко многим"

Один ко многим — это "иерархическая связь", т.е. по отношению одной сущности к другой есть множественность, а в обратную строну - нет. По сути является "расширением" связи типа "один к одному".

***Например**:* взаимоотношения командиров в армии — это серия таблиц, где "соседние" звания связаны как "один ко многим". Т.е. "у одного генерала под командованием несколько полковников". Или - одна большая группа учеников ходит в одну школу, другая в другую - тут "у одной школы много учеников".

Ученик не может ходить сразу в две школы (в обычной ситуации) - а значит, в обратную сторону "от ученика к школе" множественности нет (иначе имели бы связь "многие ко многим") — значит это "один ко многим".

Связь "Многие ко многим"

Например: Таблица предметов и таблица студентов университета. Рассуждаем: ясно что один студент может ходить на много предметов, при этом один предмет может слушаться многими студентами — значит, это "многие ко многим"

Проектирование БД: вводится дополнительная таблица, в каждый кортеж которой входят два ключа, каждый из этих ключей указывает на одну

из двух таблиц сущностей (между которыми таким образом и прокладывается связь "многие ко многим").

7.5. Нормализация отношений

Процесс преобразования отношений базы данных к виду, отвечающему нормальным формам, называется **нормализацией**. Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение, или увеличение физического объёма базы данных [10]. Конечной целью нормализации является уменьшение потенциальной противоречивости, хранимой в базе данных информации.

Устранение избыточности производится, как правило, за счёт декомпозиции отношений таким образом, чтобы в каждом отношении хранились только первичные факты, то есть факты, не выводимые из других хранимых фактов.

7.6. Нормальные формы

Нормальная форма – свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности, потенциально приводящей к логически ошибочным результатам выборки или изменения данных. Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение. Другими словами, нормальная форма — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).

Цель нормализации: исключить избыточное дублирование данных, которое является причиной аномалий, возникших при добавлении, редактировании и удалении кортежей (строк таблицы).

7.7. Создание базы данных с помощью SQLite

SQLite — это библиотека, написанная на языке C, которая предоставляет легкую дисковую базу данных, не требующую отдельного серверного процесса и позволяющую получить доступ к базе данных с использованием нестандартного варианта языка запросов SQL [11]. Некоторые приложения могут использовать SQLite для внутреннего хранения данных. Также возможно создать прототип приложения с использованием SQLite, а затем перенести код в большую базу данных, такую как PostgreSQL или Oracle.

Для использования СУБД sqlite3 нужно совершить импорт библиотеки «sqlite3» с помощью инструкции «import». Sqlite3 уже включена в состав Python, поэтому устанавливать ее дополнительно не надо.

На рис. 7.1 представлена модель сущность-связь (ER-model) базы данных.

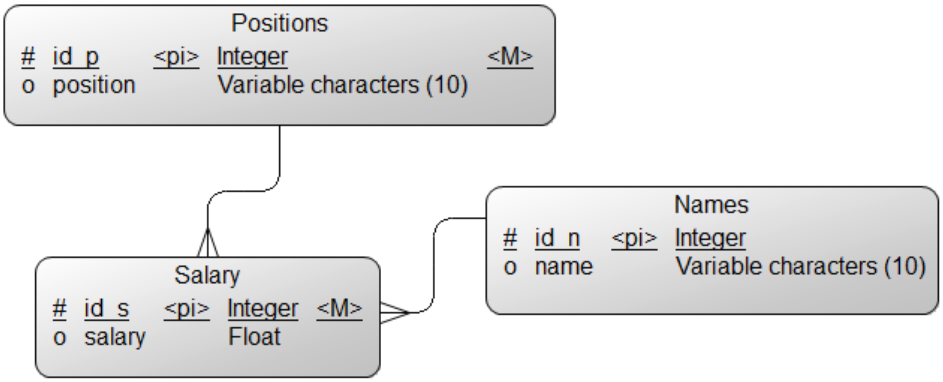


Рис. 7.1 Модель сущность-связь.

На рис. 7.2 представлена физическая модель базы данных.

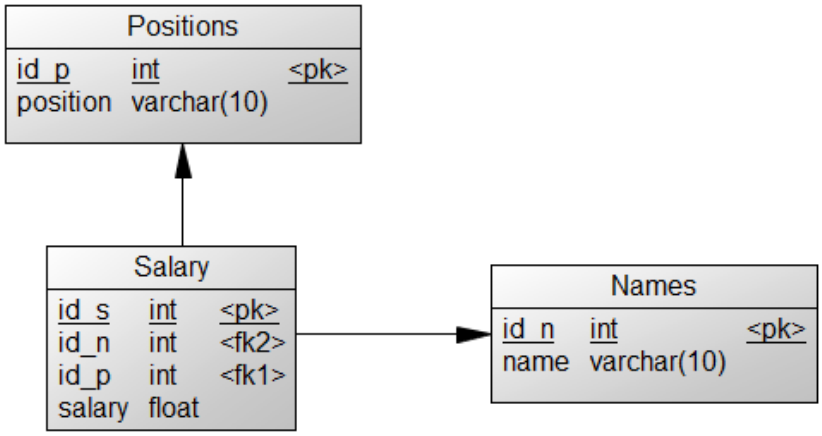


Рис. 7.2 Физическая модель базы данных.

Затем создаем базу данных с помощью функции «connect», здесь стоит отметить, что создать базу данных можно как в оперативной памяти компьютера, так и в виде файла, в нашем примере приведены два способа создания, при реализации необходимо выбрать одни из них. Чтобы обращаться к базе данных нужно создать объект «cursor», создадим его [12]:

```
import sqlite3
connection = sqlite3.connect(":memory:") # в оперативной
```

```
памяти
    connection = sqlite3.connect('db_file.db') # в виде файла
    cursor = connection.cursor()
```

Теперь, когда все необходимые объекты созданы можно обратиться к базе данных с помощью запросов SQL, отмечу, что приведены запросы создания таблиц без учета того, что в базе данных уже такие таблицы могут существовать, потому что наша база создана в оперативной памяти и каждый раз обращаясь к ней при запуске программы, база данных будет пуста. После каждого запроса выполняется метод «commit», для подтверждения изменений в базе данных, это особенно актуально, когда у базы данных есть и другие подключения кроме вашего:

```
cursor.execute("""create table Positions
(
   id_p                 int not null,
   position             char(10),
   primary key (id_p)
);""")
connection.commit()

cursor.execute("""
create table Names
(
   id_n        int not null,
   name        char(10),
   primary key (id_n)
);""")
connection.commit()

cursor.execute("""
create table Salary
(
   id_s                int not null,
   id_n                int,
   id_p                 int,
   salary               float,
   primary key (id_s),
   constraint FK_SALARY_SURNAMES foreign key (id_n)
      references Names (id_n) on delete restrict on update
restrict,
   constraint "FK_SALARY_POSITION" foreign key (id_p)
      references Positions (id_p) on delete restrict on update
restrict
);
""")
connection.commit()
```

Теперь, когда таблицы созданы, можно заполнить их записями. Заполнять будем с помощью запросов и не забываем выполнить метод «commit»:

```
cursor.execute("""
insert into Names (id_n, name) values
(8, 'Jack'),
(16, 'Jason'),
(9, 'Rick');
""")
connection.commit()

cursor.execute("""
insert into Positions (id_p, position) values
(7, 'Driver'),
(18, 'Actor'),
(16, 'Scientist');
""")
connection.commit()

cursor.execute("""
insert into Salary (id_s, id_n, id_p, salary) values
(1, 8, 7, 10000),
(2, 16, 18, 20000),
(3, 9, 16, 10000);
""")
connection.commit()
```

Затем, после создания базы данных и наполнения ее записями, можно эти записи извлечь с помощью запроса SQL, который извлечет записи из двух таблиц, в результате переменной «tbl_data» будет присвоен список кортежей, где каждый кортеж будет содержать пары значений - имя человека и его зарплата. И в завершении работы не забываем закрыть соединение базой данных с помощью метода «close»:

```
cursor.execute(f"SELECT n.name, s.salary FROM salary as s,
names as n where n.id_n = s.id_n")
tbl_data = cursor.fetchall()
print(tbl_data)

connection.close()
```

7.8. Работа с SQLite и Pandas DataFrame

Для чтения результатов SQL-запроса можно использовать функцию `read_sql_query` библиотеки `pandas`. Приведенный ниже код выполнит тот же запрос, который мы только что сделали, но вернет объект типа **DataFrame**. Такой подход несколько преимуществ по сравнению с запросом, выполненным нами ранее, т.к. [13]:

- не требуется создание объекта Cursor или вызова fetchall в конце;
- имена заголовков из таблицы считываются автоматически;

- создаётся объект DataFrame (Таблица), позволяющий отобразить данные в удобочитаемом виде.

```
import pandas as pd
connection = sqlite3.connect("db_file.db")
df = pd.read_sql_query('select * from Names', connection)
df
```

	id_n	name
0	8	Jack
1	16	Jason
2	9	Rick

В результате получаем красиво отформатированный объект DataFrame, с помощью него можно легко обратиться к определённому столбцу таблицы и вывести все имена рабочих: `df['name'].`

```
0     Jack
1    Jason
2     Rick
Name: name, dtype: object
```

Добавить новую строку в таблицу можно с помощью следующего кода:

```
df.loc[ len(df.index )] = [12, 'Andrey']
```

Результат добавления новой строки:

	id_n	name
0	8	Jack
1	16	Jason
2	9	Rick
3	12	Andrey

Создание таблиц с помощью Pandas

Пакет pandas дает нам более быстрый способ создания таблиц. Нам просто нужно сначала создать DataFrame, а затем экспортировать его в таблицу SQL. Сначала мы создадим DataFrame:

```
    from datetime import datetime
    conn = sqlite3.connect("mydatabase.db")
    cur = conn.cursor()
    df = pd.DataFrame(
    [[1,'Sergeev Ivan Andreevich',
'Informatics',datetime(2016,9,29,12,20),
datetime(2016,9,29,12,20)]],
        columns=['id', 'TeachersFIO','subject','lectureBegin',
'lectureEnd']
    )
```

	id	TeachersFIO	subject	lectureBegin	lectureEnd
0	1	Sergeev Ivan Andreevich	Informatics	2016-09-29 12:20:00	2016-09-29 12:20:00

Затем мы сможем вызвать метод to_sql для преобразования df в таблицу в базе данных. Мы устанавливаем параметр if_exists, чтобы удалить и заменить любые существующие таблицы с именем “raspisanie”:

```
df.to_sql('raspisanie',conn, if_exists='replace')
```

Затем мы можем проверить, что все работает, запросив базу данных:

```
pd.read_sql_query("select * from raspisanie", conn)
```

	index	id	TeachersFIO	subject	lectureBegin	lectureEnd
0	0	1	Sergeev Ivan Andreevich	Informatics	2016-09-29 12:20:00	2016-09-29 12:20:00

Редактирование таблиц с помощью Pandas

При работе с данными иногда приходится изменять конфигурацию таблиц. Допустим в ранее созданную таблицу “raspisanie” нам необходимо добавить поле аудитория («room»). Сначала сделаем это с помощью курсора cur:

```
    cur.execute('alter table raspisanie add column room
integer')
    df = pd.read_sql_query("select * from raspisanie", conn)
    df
```

	index	id	TeachersFIO	subject	lectureBegin	lectureEnd	room
0	0	1	Sergeev Ivan Andreevich	Informatics	2016-09-29 12:20:00	2016-09-29 12:20:00	None

Обратите внимание, что в SQLite все столбцы имеют значение null (что в Python переводится как None), поскольку для этого столбца еще нет никаких значений.

В библиотеке Pandas также предусмотрена подобная функция добавления в таблицу столбцов:

```
    df['room'] = 410          # изменяем значении аудитории на
410
    df['building'] = None      # добавляем новый столбец с
номером
                                                    #корпуса
    # Сохраняем в нашу базу
    df.to_sql('raspisanie',conn,if_exists='replace')
```

	level_0	index	id	TeachersFIO	subject	lectureBegin	lectureEnd	building	room
0	0	0	1	Sergeev Ivan Andreevich	Informatics	2022-09-29 12:20:00	2022-09-29 14:10:00	None	410

Изменить значение поля в определенной записи (строке) можно следующим образом:

```
df.loc[0,'lectureEnd'] = '2022-09-29 14:30'
```

Результат изменения поля, находящегося в строке «0» и столбце «lectureEnd»:

	id	TeachersFIO	subject	lectureBegin	lectureEnd	room	building
0	1	Sergeev Ivan Andreevich	Informatics	2022-09-29 12:20:00	2022-09-29 14:30:00	410	None

7.9. Контрольные вопросы для самопроверки

1. В чем разница между первичным и внешним ключами?
2. Какие типы связей могут использоваться при создании реляционной базы данных?
3. Что такое нормализация отношений и сколько нормальных форм вы знаете?
4. Что произойдет при выполнении следующих трёх инструкций:

```
import sqlite3
con = sqlite3.connect(':memory:')
crsr = con.cursor()
```

5. Для чего нужна библиотека pandas?

8. Визуализация модели линейной регрессии

В этом разделе мы познакомимся с одним из базовых методов машинного обучения – линейной регрессией и научимся отображать полученные результаты на графике с помощь библиотеки *matplotlib*.

8.1. Линейная регрессия

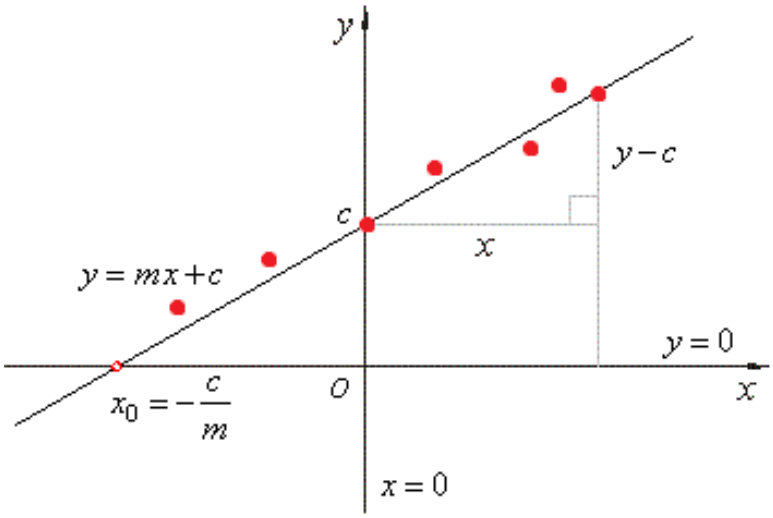
Одной из наиболее распространённых задач машинного обучения является поиск зависимости в данных. Допустим, у нас есть набор некоторых измерений – среднесуточных температур. Наша задача – выяснить зависимость этих значений, т.е. подобрать функцию, которая бы описывала эти значения, т.е. могла бы предсказывать значения в пропусках и прогнозировать их на будущее. В качестве базовой может быть взята любая функция: линейная, квадратичная, экспоненциальная и др. От выбора базовой функции зависит сложность полученной модели и количество параметров, настраиваемых в процессе обучения модели. В нашей работе будем применять метод линейной регрессии.

В статистике линейная регрессия – это линейный подход к моделированию взаимосвязей между зависимой переменной и одной или несколькими независимыми переменными. За основу данного метода берётся линейная функция.

В случае одной независимой переменной данный метод называется *простой линейной регрессией*. Для более чем одной независимой переменной – *множественной линейной регрессией*.

Ознакомимся с простой линейной регрессией. Пусть x – независимая переменная, а y – зависит от x. Есть набор n точек с координатами (x_i, y_i). Задача линейной регрессии состоит в построении такой прямой линии, чтобы все n точек имели минимальное отклонение от неё (рис. 8.1).

Уравнение такой прямой задаётся выражением $y' = mx+c$, где m определяет угол наклона линии, а c – сдвиг относительно начала координат (рис. 8.1). Таким образом задача линейной регрессии заключается в определении параметров m и c, при которых ошибка (отклонение точек от построенной прямой) будет минимальной для данного набора x. Мы будем делать это с помощью метода наименьших квадратов (МНК) [14].



ы

Рис. 8.1 Аппроксимация точек на основе метода линейной регрессии

Для начала необходимо задать **функцию ошибки (потерь),** т.е. посчитать отклонения прямой от реальных данных, которые мы пытаемся описать. Достаточно распространенным вариантом функции потерь является к**вадратичная функция**, которая определяется как:

$$L(x, m, c) = \sum_{i=1}^{n} (y_i - y_i')^2 = \sum_{i=1}^{n} (y_i - mx_i - c)^2 \to min$$

Для минимизации функции потерь необходимо найти частные производные функции ***L*** по параметрам *m* и *c*, приравнять их к 0, и вычислить параметры *m* и *c*:

$$\begin{cases} \dfrac{\partial L(x, m, c)}{\partial m} = -2 \sum_{i=1}^{n} (y_i - mx_i - c)x_i = 0 \\ \dfrac{\partial L(x, m, c)}{\partial c} = -2 \sum_{i=1}^{n} (y_i - mx_i - c) = 0 \end{cases}$$

Решая эту систему уравнений, получим:

$$m = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$c = \bar{y} - m\bar{x}.$$

Здесь $\bar{x}$ – среднее всех входных значений выборки *x*, $\bar{y}$ – среднее всех значений *y*, *n* – количество точек в выборке. Эти формулы и определяют метод наименьших квадратов (МНК).

Таким образом, вычислив коэффициенты *m* и *c*, мы найдём уравнение линейной регрессии для предложенного набора точек.

8.2. Библиотека matplotlib

Установить matplotlib можно одной из двух приведенных команд [15]:

```
python -m pip install matplotlib
```

Пример кода для вывода графиков с помощью библиотеки matplotlib:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5, 6, 7]
y = [6.5, 4.4, 1.2, 4.9, 0, 2.3, 6]

m,c = MNK(x,y) # функцию MNK(x,y) нужно реализовать самим

plt.plot(x, y, 'o', label='Original data', markersize = 8)
plt.plot(x, m*x + c, 'red', label='Fitted line')

plt.xlabel('input')
plt.ylabel('output')

plt.legend()

plt.show()
```

Описание кода, приведенного выше: подключаем пакет `matplotlib.pyplot`, задаем значения x и y – это наши данные для аппроксимации. Вызываем реализованную функцию MNK (нужно реализовать самостоятельно!) для расчета коэффициентов m и c. Формируем графики для входных данных и линии тренда, подписываем оси, добавляем легенду и выводим график на экран.

8.3. Контрольные вопросы для самопроверки

1. Как вы понимаете задачу6 поиск зависимости в данных.
2. Что такое линейная регрессия. Опишите модель.
3. В чем заключается метод наименьших квадратов.
4. Опишите задачу аппроксимации данных
5. Опишите задачу прогнозирования данных.
6. Как можно установить библиотеку «matplotlib» и для чего она нужна?

II. Практический раздел

1. Лабораторная работа №1

Подготовка рабочей среды. Создание первого приложения на Python

Цель работы:

Овладеть навыками настройки среды программирования и написать свое первое приложение на Python.

Задание:

1. Установить Python и PyCharm.
2. Настроить PyCharm.
3. Запустить свою первую программу.

Ход работы:

1. Установка Python.

1) Сначала необходимо зайти на официальный сайт Python: https://www.python.org/

2) Затем скачиваем установочный файл последний версии

3) После скачивания файла выполняем установку Python в ОС Windows.

4) Теперь, когда python установлен, открываем командную строку Windows. Одним из способов, например, нажав сочетание клавиш Win + "X", во всплывшем списке меню нажимаешь "Выполнить", появится окно, в появившемся окне вводим "cmd" и жмем "Ok". Результат, появившаяся командная строка виндоус. Проверить установлен ли Python в системе можно командной:

" python --version "

Если установлен, мы увидим номер установленной версии Python.

2. Установка PyCharm

1) Сначала необходимо зайти на официальный сайт PyCharm: https://www.jetbrains.com/pycharm/download

2) Затем скачиваем установочный файл версии “Community”

3) После скачивания файла выполняем его установку.

4) После установки запускаем PyCharm, указываем путь к интерпретатору Python, который установили на первом шаге.

3. Реализуем своб первую программу, предлагающую пользователю ввести любые два числа и выполняющую операции сложения, умножения и

вычитания над ними. Программа должна будет предложить ввести два числа, а затем произвести математические операции над ними.

Сначала объявим переменные, пусть их будет две:

```
a = int(input('Введите первое число: '))
b = int(input('Введите второе число: '))
```

В приведенных строках кода, происходит следующее, сначала программа предлагает пользователю ввести число с помощью функции “input”, затем полученное значение преобразуется в строку с помощью функции “int”, потому что функция “input” возвращает данные текстового типа. И последнее действие — это присваивание переменной полученного значения в результате описанной работы функций.

Теперь в соответствии с заданием нужно произвести математические операции:

```
print(f"Сумма: {a + b}")
print(f"Произведение: {a * b}")
print(f"Разность: {a - b}")
```

В приведенных строках кода, происходит следующее, для вывода результата используется функция “print”, математические операции выполняются с помощью математических операторов сложение(+), вычитание(-) и произведение(*). Следует отметить, что с помощью f-строк в фигурных скобках производятся математические операции.

Дополнительные материалы по теме лабораторной:

- https://www.python.org/
- https://www.lfd.uci.edu/~gohlke/pythonlibs/
- https://www.jetbrains.com/ru-ru/
- https://www.youtube.com/watch?v=uWLwvhbbEe4
- https://pythonru.com/baza-znanij/poshagovaja-ustanovka-pycharm
- https://python-scripts.com/f-strings

2. Лабораторная работа №2

Последовательности. Циклы. Функции.

Цель работы:

Овладеть навыками создания функций, программирования циклов и обработки последовательностей с помощью циклов.

Задание:

1. Реализовать программу, выполняющую перебор двух последовательностей поэлементно, причем такую, что цикл *for* вложен в цикл *while*, и когда попадаются совпадающие символы, программа должна выводить совпавший элемент и его индекс.

2. Реализовать функцию, принимающую от пользователя любую последовательность и с помощью функций *map* и *lambda* меняющую элементы, таким образом, что если элемент число, тогда производится операция умножения на пять, а если элемент имеет строковый тип, тогда производится размножение элемента в пять раз. Результат работы возвращать в виде списка.

Ход работы:

1. Сначала объявим два списка одинаковой длины:

```
list1 = [2,8,8,3,5,9,4]
list2 = [2,8,0,3,6,9,5]
```

Затем добавим переменную-счетчик:

```
n = 0
```

Теперь зададим цикл «while» и вложим в него цикл «for»

```
while n < len(list1)-1:
for el in list1:
    if el == list2[n]:
        print(f'Совпадение {el}[{n}]')
        break
n += 1
```

Как видно из приведенных выше строк кода в цикл «for» вложен оператор условия – «if», где происходит проверка равенства элементов списков, если элементы равны, то выводится сообщение о совпадении элементов и их индекс. Следует обратить внимание на присутствие инструкции «break», она необходима для выхода из цикла «for» после нахождения двух совпадающих элементов, чтобы начать сравнивать

следующие пары элементов. Также следует обратить внимание на присутствие счетчика в виде переменной «n» для цикла «while» и отсутствие такового для цикла «for», это связано с особенностями циклов.

2. Чтобы создать функцию в Python необходимо воспользоваться инструкцией «def», затем указать имя функции и в скобках необходимые параметры:

```
def func(b):
if not b.isdigit():
    return list(map(lambda i: i * 2, b))[0]
else:
    return list(map(lambda i: int(i) * 2, b))[0]

a = input('Введите первое число: ')
print(func(a))
```

В приведенном коде мы объявили функцию «func». В теле функции объявлен оператор «if», который проверяет тип переменной «b», если переменная оказывается не числом, заметьте условие дословно так и звучит, ведь указан оператор «not», тогда мы просто размножаем ее в количестве равном двум, как строку, а если переменная число, то мы преобразуем ее тип с помощью функции «int» и выполняем математическую операцию умножение.

Обратите внимание на квадратные скобки после скобок функции «list()», так как функция «list()» возвращает список из одного элемента, то чтобы «func» возвращала число, мы просто указывает его индекс в квадратных скобках.

Также в коде приведены функции «map» и «lambda». Функция «map» применяет указанную в ней функцию к каждому элементу переданной ей последовательности. Функция «lambda» содержит аргументы, и после двоеточия выражение, которое необходимо над ними выполнить.

Дополнительные материалы по теме лабораторной:

- https://dev-gang.ru/article/lambda-map-i-filter-v-python/
- https://book.pythontips.com/en/latest/map_filter.html
- https://all-python.ru/osnovy/proverka-na-chislo.html#isdigit-isnumeric-i-isdecimal

3. Лабораторная работа №3

Объектно-ориентированное программирование.

Цель работы:

Овладеть основами методологии объектно-ориентированного программирования.

Задание:

1. Реализовать класс Volume, в котором определить атрибуты: length (длина), width (ширина) и height (высота). Значения данных атрибутов должны передаваться при создании экземпляра класса. Атрибуты сделать защищенными. Определить метод расчета, использовав формулу: длина*ширина*высота. Проверить работу метода.

2. Реализовать базовый класс Employee, в котором определить атрибуты: name, patronymic, surname, salary. Последний атрибут должен быть защищенным и ссылаться на словарь, содержащий элементы: жалование и бонус, например, {"wage": wage, "bonus": bonus}. Создать класс Salary на базе класса Employee. В классе Salary реализовать методы получения полного имени сотрудника (get_full_name) и дохода с учетом премии (get_total_income). Создать экземпляры класса Salary, передать данные, проверить значения атрибутов, вызвать методы экземпляров.

Ход работы:

1.Чтобы создать указанный класс нужно воспользоваться инструкцией «class», а затем через пробел указать его имя. Далее воспользовавшись методом __init__() определим конструктор класса, и в качестве параметров укажем три аргумента не считая параметр «self», а именно длину (length), ширину (width) и высоту(height). Так как по заданию нужно чтобы переменные были защищены, то по правилам Python нужно указать на их защищенность путем знака нижнего подчеркивания (_) перед именем переменной.

```
class Volume:
def __init__(self, length, width, height):
    self._length = length
    self._height  = height
    self._width = width
```

Теперь в созданном классе «Volume», определим еще один метод, который назовем «count_volume», с помощью которого будет рассчитывать

объем, путем перемножения переменных, а именно длины (length), ширины (width) и высоты (height).

```
    def count_volume(self):
        print(f"{(int(self._length) * int(self._height) * \
        int(self._width))}")
```

Затем нужно создать экземпляр класса, путем присваивания переменной, в нашем случае «obj», имени класса и в скобках после имени указать значения наших параметров.

После указываем имя переменной, которая является экземпляром класса и через точку вызываем метод «count_volume()».

```
    obj = Volume(input("Введите длину:"),
                 input("Введите высота:"),
                 input("Введите ширину:"))
    obj.count_volume()
```

2. По схеме, описанной в п1. «ход работы», создаем класс и его конструктор, не забывая указать необходимые переменные в соответствии с заданием.

```
    class Employee:
    def __init__(self, name, patronymic, surname, income,
bonus):
        self.name = name
        self.patronymic = patronymic
        self.surname = surname
        self._salary = {"wage": 0, "bonus": 0}
        self._salary["wage"] = income
        self._salary["bonus"] = bonus
```

Затем создаем дочерний класс «Salary», содержащий два метода «get_full_name» и «get_total_income»

```
    class Salary(Employee):
    def get_full_name(self):
        print(f"Полное имя сотрудника: {self.name}
{self.patronymic} {self.surname}")

    def get_total_income(self):
        print(f"Доход с учетом премии:
{int(self._salary['wage']) + int(self._salary['bonus'])}")
```

Теперь нужно создать экземпляр класса и вызвать методы. Обратите внимание, что при создании экземпляра дочернего класса интерпретатором

используется конструктор родительского класса, что проявляется через указание параметров, которых в дочернем классе мы не указывали.

```
name = input('Введите имя сотрудника:')
surname = input('Введите фамилию сотрудника:')
patronymic = input('Введите отчество сотрудника:')
wage = input('Введите зарплату сотрудника:')
bonus = input('Введите премию сотрудника:')

obj = Salary(name, surname, patronymic, wage, bonus)
obj.get_full_name()
obj.get_total_income()
```

Дополнительные материалы по теме лабораторной:

- https://pythonworld.ru/osnovy/obektno-orientirovannoe-programmirovanie-obshhee-predstavlenie.html
- https://pythonchik.ru/osnovy/osnovy-oop-v-python-klassy-obekty-metody

4. Лабораторная работа №4

Работа с простой базой данных.

Цель работы:

Овладеть основами взаимодействия с базами данным с использованием языка Python.

Задание:

Реализовать создание базы данных из трех связанных таблиц, заполнить таблицы записями и написать один любой запрос к базе данных на извлечение записей.

Ход работы:

В нашей работе будем использовать СУБД sqlite3, для этого нужно совершить импорт библиотеки «sqlite3» с помощью инструкции «import». Sqlite3 уже включена в состав Python, поэтому устанавливать ее дополнительно нет необходимости.

Затем создаем базу данных с помощью функции «connect», здесь стоит отметить, что создать базу данных можно как в оперативной памяти компьютера, так и в виде файла, в нашем примере создадим в памяти. Чтобы обращаться к базе данных нужно создать объект «cursor», создадим его.

```
import sqlite3
connection = sqlite3.connect(":memory:") # можно указать
                                          'db_file.db'
cursor = connection.cursor()
```

Теперь, когда все необходимые объекты созданы можно обратиться к базе данных с помощью запросов SQL, отмечу, что приведены запросы создания таблиц без учета того, что в базе данных уже такие таблицы могут существовать, потому что наша база создана в оперативной памяти и каждый раз обращаясь к ней при запуске программы, база данных будет пуста. После каждого запроса выполняется метод «commit», для подтверждения изменений в базе данных, это особенно актуально, когда у базы данных есть и другие подключения кроме вашего.

```
cursor.execute("""create table Positions
(
   id_p                 int not null,
   position             char(10),
   primary key (id_p)
```

```
);""")
connection.commit()

cursor.execute("""
create table Names
(
   id_n        int not null,
   name        char(10),
   primary key (id_n)
);""")
connection.commit()

cursor.execute("""
create table Salary
(
   id_s                int not null,
   id_n                int,
   id_p                 int,
   salary               float,
   primary key (id_s),
   constraint FK_SALARY_SURNAMES foreign key (id_n)
      references Names (id_n) on delete restrict on update
restrict,
   constraint "FK_SALARY_POSITION" foreign key (id_p)
      references Positions (id_p) on delete restrict on update
restrict
);
""")
connection.commit()
```

Теперь, когда таблицы созданы, можно заполнить их записями. Заполнять будем с помощью запросов и незабываем выполнить метод «commit».

```
    cursor.execute("""
insert into Names (id_n, name) values
(8, 'Jack'),
(16, 'Jason'),
(9, 'Rick');
""")
connection.commit()

cursor.execute("""
insert into Positions (id_p, position) values
(7, 'Driver'),
(18, 'Actor'),
(16, 'Scientist');
```

```
""")
connection.commit()

cursor.execute("""
insert into Salary (id_s, id_n, id_p, salary) values
(1, 8, 7, 10000),
(2, 16, 18, 20000),
(3, 9, 16, 10000);
""")
connection.commit()
```

Затем, после создания базы данных и наполнения ее записями, можно эти записи извлечь с помощью запроса SQL, который извлечет записи из двух таблиц, в результате переменной «tbl_data» будет присвоен список кортежей, где каждый кортеж будет содержать пары значений - имя человека и его зарплата. И в завершении работы не забываем закрыть соединение базой данных с помощью метода «close».

```
cursor.execute(f"SELECT n.name, s.salary FROM salary as s,
names as n where n.id_n = s.id_n")
tbl_data = cursor.fetchall()
print(tbl_data)

connection.close()
```

Дополнительные материалы по теме лабораторной:

- https://pythonru.com/osnovy/sqlite-v-python
- https://pythonru.com/biblioteki/vvedenie-v-sqlite-python
- https://docs.python.org/3/library/sqlite3.html
- https://tproger.ru/translations/sql-recap/

III. Раздел для самостоятельной работы

1. Самостоятельная работа №1

Цель работы:

Обрести навыки применения математических операторов и методов преобразования типов.

Задание:

Реализовать программу, выполняющую операции сложения, умножения, деления и вычитания над любыми числами, при этом количество переменных должно быть не меньше трех. Программа должна производить вывод применяя три разных способа форматирования строк (Си-стиль, метод format, f-строки). А также программа должна преобразовывать тип вводимых значений из строкового в целые и вещественные числа с указанием количества знаков после запятой.

2. Самостоятельная работа №2

Цель работы:

Обрести навыки применения условной инструкции и усовершенствовать навыки работы с математическими операторами.

Задание:

Напишите программу для решения восьми уравнений, приведенных под номерами с 1-го по 8-й. Предусмотрите проверку деления на ноль с помощью оператора `if`. Все необходимые переменные пользователь вводит через консоль. Вывод результата оформить с помощью f-строк.

Для выполнения работы вам понадобится библиотека **math**. Подключить её можно с помощью команды: **import math** в самом начале программы.

1)

$$k = ln\left|(y - \sqrt{|x|}) \cdot \left(x - \frac{y}{z + x^2 / 4}\right)\right|$$

2)

$$d = \frac{2cos(x - \pi / 6)}{1/2 + sin^2(y)} + \frac{|y - x|}{3}$$

3)

$$w = \frac{(x / y) \cdot (z + x) \cdot e^{|x-y|} + \ln(1+e)}{\sin^2(y) - (\sin(x) \cdot \sin(y))^2}$$

4)

$$b = \frac{3 + e^{y-1}}{1 + x^2 \cdot |y - tg(z)|}$$

5)

$$p = \begin{cases} \sqrt{|a \cdot b|} + 2 \cdot c, a \cdot b < -2 \\ a^3 + b^2 - c^2, -2 \le a \cdot b \le 2 \\ a^c - b, a \cdot b > 2 \end{cases}$$

6)

$$h = \begin{cases} arctg(x + |y|), x < y \\ arctg(|x| + y), x > y \\ (x + y)^2, x = y \end{cases}$$

7)

$$b = \begin{cases} \ln(x / y) + (x^2 + y)^3, x / y > 0 \\ \ln|x / y| + (x^2 + y)^3, x / y < 0 \\ (x^2 + y)^3, y \neq 0, x = 0 \\ 0, y = 0 \end{cases}$$

8)

$$b = \begin{cases} \sin(x + y) + 2 \cdot (x + y)^2, x - y > 0 \\ \sin(x - y) + (x - y)^3, x - y < 0 \\ |x^2 + \sqrt{y}|, y \neq 0, x = 0 \\ 0, y = 0 \end{cases}$$

3. Самостоятельная работа №3

Цель работы:

Обрести навыки работы с циклами и последовательностями.

Задание:

Реализовать следующие алгоритмы сортировки:

- Сортировка выбором;
- Сортировка вставками;
- Сортировка “Методом пузырька”;
- Сортировка Шелла;

- Быстрая сортировка.

Программа должна сгенерировать список случайных неотсортированных чисел и вывести его на экран.

Затем этот список должен быть отсортирован каждым из пяти перечисленных выше алгоритмов сортировки. Необходимо также зафиксировать время работы каждого алгоритма сортировки и при выводе отсортированного массива на экран выводить время его сортировки.

4. Самостоятельная работа №4

Цель работы:

Обрести навыки работы с файлами.

Задание:

На основе самостоятельной работы №3 реализовать программу, так, чтобы в файл формата *.xlsx, по столбцам, записывались все отсортированные массивы с указанием, в начальных ячейках, времени их сортировки и названия алгоритма сортировки, а в самый первый столбец файла записать неотсортированный массив. При реализации программы все алгоритмы сортировки должны быть представлены в виде функций, а также реализовать функцию записи в файл формата *.xlsx.

	A	B	C
1	неотсорт	пузырек_00:10:00	вставками_00:01:00
2	3	2	2
3	7	3	3
4	4	4	4
5	5	5	5
6	2	7	7

5. Самостоятельная работа №5

Цель работы:

Обрести навыки применения объектно-ориентированного программирования.

Задание:

1. Создать класс *ThreeStates* и определить у него поле *state* и метод *hasstate*. Атрибут *state* реализовать как приватный. В рамках метода реализовать переключение состояний: первое, второе, третье. Продолжительность первого состояния составляет 5 секунд, второго — 3 секунды, третьего — 2 секунды. Переключение между состояниями должно

осуществляться только в указанном порядке (первое, второе, третье). Проверить работу примера, создав экземпляр и вызвав описанный метод.

2.Реализовать класс *Volume*, в котором определить защищенные поля: *length* (длина), *width* (ширина) и *height* (высота). Значения данных атрибутов должны передаваться при создании экземпляра класса. Определить метод расчета, использовав формулу: длина*ширина*высота. Проверить работу метода.

3. Реализовать базовый класс *Employee*, в котором определить атрибуты: *name*, *patronymic*, *surname*, *salary*. Последний атрибут должен быть защищенным и ссылаться на словарь, содержащий элементы: жалование и бонус, например, *{"wage": wage, "bonus": bonus}*. Создать класс *Salary* на базе класса *Employee*. В классе *Salary* реализовать методы получения полного имени сотрудника (get_*full*_name) и дохода с учетом премии (*get_total_income*). Создать экземпляры класса *Salary*, передать данные, проверить значения атрибутов, вызвать методы экземпляров.

4. Реализуйте базовый класс *Airplane*. У данного класса должны быть следующие атрибуты: *speed, color, name, is_jet* (булево). А также методы: *go*, *stop, direction*, которые должны сообщать, что самолет летит, не летит, повернул в полете. Опишите несколько дочерних классов самолетов: *FastAirplane, Biplane, ArmyAirplane*. Добавьте в базовый класс метод *show_speed*, который должен показывать текущую скорость самолета. Для класса *FastAirplane* переопределите метод *show_speed*. При значении скорости свыше 1300 (*FastAirplane*) должно выводиться сообщение о сверхзвуковой скорости. Создайте экземпляры классов, передайте значения атрибутов. Выполните доступ к атрибутам, выведите результат. Выполните вызов методов и также покажите результат.

5. Реализовать класс *MathOperations*. Определить в нем атрибуты *first_num*, *second_num* и метод *calc*. Метод выводит сообщение “Запуск операции”. Создать три дочерних класса *my_sum* (сложение), *my_sub* (вычитание), *my_mult* (умножение). В каждом из классов реализовать переопределение метода *calc*. Для каждого из классов метод должен выполнять математическую операцию, соответствующую названию класса. Создать экземпляры классов и проверить работу методов.

6. Самостоятельная работа №6

Цель работы:

Обрести навыки работы с базами данных.

Задание:

Создать программу для работы с базой данных. База данных должна содержать не менее 5 связанных таблиц. Реализовать вывод данных из созданной базы данных, запросы должны выбирать данные из нескольких таблиц. Допускается данные для разрабатываемой базы данных сгенерировать с помощью модуля «random», либо использовать из сторонних источников. Применить ООП стиль программирования.

7. Самостоятельная работа №7

Цель работы:

Обрести навыки реализации алгоритмов машинного обучения.

Задание:

1. Создать или загрузить таблицу с данными для аппроксимации и прогноза. Постарайтесь подобрать данные, связанные с вашей основной специальностью или научной работой.

2. Построить три различных аналитических графика с помощью библиотеки *matplotlib*.

3. Написать функции, вычисляющие коэффициенты *m* и *c* линейной регрессии методом наименьших квадратов, по загруженным/созданным данным.

4. Вывести линию тренда, полученную с помощью МНК.

5. Расширить список значений *x* и спрогнозировать значения *y* с помощью полученной линии тренда. Результат отобразить на отдельном графике.

Заключение

Учебное пособие составлено с целью оказания помощи обучающимся в обретении практических навыков программирования при изучении вопросов создания программ на языке Python. Материалы пособия включают как базовые темы, то, без чего невозможно реализовать алгоритмы на языке Python, например, последовательности и циклы, так и более узкоспециализированные, к таким темам можно отнести работу с базами данных и реализацию алгоритмов машинного обучения.

Ожидается, что по итогам ознакомления с изложенными в пособии материалами у обучающихся сложится представление о возможностях языка Python и как эти возможности следует применять на практике.

Для обучающихся, кто действительно захочет развиваться в программирования на Python, дальнейшими шагами по развитию навыков могут стать: изучение анализа данных, веб-программирование или создание игр с помощью средств языка Python.

Список литературы

1. Лутц М. Изучаем Python, том 1, 5-е изд.: Пер. с англ. — СПб.: ООО “Диалектика”, 2019. — 832 с.

2. Любанович Б. Простой Python. Современный стиль программирования. 2-е изд. — СПб.: Питер, 2021. — 592 с., ил. ISBN 978-5-4461-1639-3

3. Ввод и вывод данных. — Текст: электронный // Лаборатория линуксоида: [сайт]. — URL: https://younglinux.info/python/input (дата обращения: 25.02.2023)

4. Урок 5. Логический тип данных. — Текст: электронный // Пошаговые инструкции: [сайт]. — URL: https://gospodaretsva.com/urok-5-logicheskij-tip-dannyx.html (дата обращения: 25.02.2023)

5. Условные конструкции в Python. — Текст: электронный // pythonic way: [сайт]. — URL: http://pythonicway.com/python-conditionals (дата обращения: 25.02.2023)

6. Файлы. Работа с файлами. — Текст: электронный // Python 3 для начинающих: [сайт]. — URL: https://pythonworld.ru/tipy-dannyx-v-python/fajly-rabota-s-fajlami.html (дата обращения: 25.02.2023).

7. Creating Excel files with Python and XlsxWriter. — Текст: электронный // XlsxWriter: [сайт]. — URL: https://xlsxwriter.readthedocs.io/ (дата обращения: 05.11.2022).

8. Что такое объектно-ориентированное программирование. — Текст: электронный // Лаборатория линуксоида: [сайт]. — URL: https://younglinux.info/oopython/oop (дата обращения: 25.02.2023)

9. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. – М.: Издательский дом "Вильямс", 2005. – 1328 с.

10. Нормальная форма. — Текст: электронный // Википедия: [сайт]. — URL: https://ru.wikipedia.org/wiki/Нормальная_форма (дата обращения: 25.02.2023)

11. sqlite3 — DB-API 2.0 interface for SQLite databases. — Текст: электронный // Python: [сайт]. — URL: https://docs.python.org/3/library/sqlite3.html (дата обращения: 25.02.2023).

12. A thorough guide to SQLite database operations in Python. — Текст: электронный // Sebastianraschka: [сайт]. — URL: https://sebastianraschka.com/Articles/2014_sqlite_in_python_tutorial.html (дата обращения: 25.02.2023).

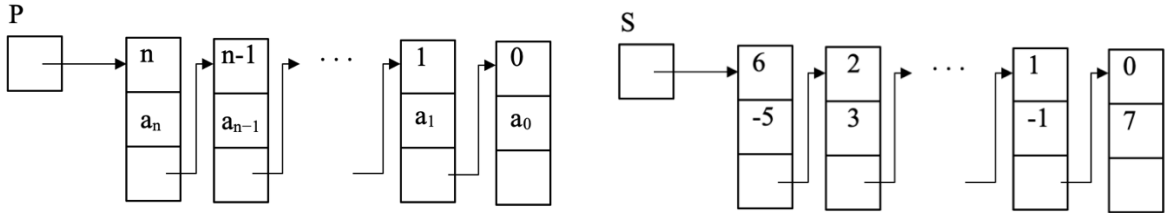
13. pandas.read_sql_query. — Текст: электронный // pandas: [сайт]. — URL: https://pandas.pydata.org/docs/reference/api/pandas.read_sql_query.html (дата обращения: 25.02.2023)

14. Least Squares Regression in Python. — Текст: электронный // Python Numerica Methods: [сайт]. — URL: https://pythonnumericalmethods.berkeley.edu/notebooks/chapter16.04-Least-Squares-Regression-in-Python.html (дата обращения: 25.02.2023).

15. Matplotlib 3.7.0 documentation. — Текст: электронный // Matplotlib: [сайт]. — URL: https://matplotlib.org/stable/index.html (дата обращения: 25.02.2023).

Приложение А. Дополнительные самостоятельные работы

1. Многочлен $P(x)=a_n x^n+a_{n-1}x^{n-1}+...+a_1 x+a_0$ с целыми коэффициентами можно представить в виде списка. При этом, если $a_i=0$, то соответствующий элемент не включается в список. На рисунке показано общее представление многочлена и пример для $S(x)=-5x_6+3x_2-x+7$:



Необходимо описать тип данных, соответствующий предложенному представлению многочленов, а также разработать процедуру Add(P,Q,R) вычисления суммы многочленов Q и R, результат – многочлен P.

2. Создать типизированный файл записей со сведениями о телефонах абонентов; каждая запись имеет поля: фамилия абонента, год установки телефона, номер телефона. По заданному номеру телефона абонента выдать его фамилию, реализовав алгоритм бинарного поиска. Определить количество установленных телефонов с N-го года.

3. Реализовать алгоритм K средних, с графическим представлением его работы.

4. Создать приложение, которое позволяет загружать текстовый файл с html-кодом или css-кодом и выводить список всех ссылок, встречающихся в этом коде. Ссылки должны быть без повторов, однако с указанием количества повторений. Добавить возможность выбора ссылки и замены на другую по всему файлу. Добавить возможность создания списка запрещенных сайтов. Все ссылки, попадающие в черный список, можно автоматически заменить на [запрещенная ссылка].

5. Создать приложение, осуществляющее математические операции сложения, вычитания и произведения над матрицами.