

Министерство цифрового развития, связи и массовых коммуникаций  
Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Сибирский государственный университет телекоммуникаций и информатики»  
(СибГУТИ)

**А.А. Агалаков**  
**К.И. Дементьева**

## **Программирование на языке Python. Базовый уровень**

*Учебное пособие*

Новосибирск  
2024

УДК 004.432(075.8)

*Утверждено редакционно-издательским советом СибГУТИ*

Рецензенты канд. техн. наук, доц. О.И. Солонская,  
канд. техн. наук, доц. А.А. Ракитский

**Агалаков А.А., Дементьева К.И.** Программирование на языке Python. Базовый уровень: учебное пособие / Сибирский государственный университет телекоммуникаций и информатики; каф. прикладной математики и кибернетики. – Новосибирск, 2024. – 117 с.

Аннотация. В пособии рассматриваются теоретические основы современных технологий и методов программирования, практические вопросы создания программ, а также основные алгоритмические конструкции и их реализация на языке высокого уровня Python. Рассмотрены основные алгоритмы линейной и циклической структуры, обработка одномерных и многомерных массивов, функции, списки, работа с файлами и наиболее популярными библиотеками. Применение алгоритмов иллюстрируется большим количеством примеров. Изложенные в пособии алгоритмы студенты реализуют в виде программ на практических занятиях и в курсовой работе по соответствующей дисциплине. Имеются вопросы для самопроверки. Пособие предназначено для студентов очной и заочной форм обучения, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника», профиль «Программное обеспечение средств вычислительной техники и автоматизированных систем», и изучающих дисциплину «Программирование».

© Агалаков А.А., Дементьева К.И., 2024

© Сибирский государственный университет  
телекоммуникаций и информатики, 2024

## Оглавление

1. Установка Python, IDE, простые типы данных .....	6
1.1. Установка Python и IDE. Онлайн платформы .....	6
1.2. Типы данных .....	10
1.2.1 Логический тип данных .....	11
1.2.2. Числа .....	11
1.3. Переменные .....	13
1.4. Ключевые слова .....	15
1.5. Оператор вывода .....	15
1.6. Оператор ввода .....	16
1.7. Арифметические операции .....	18
1.8. Комментарии .....	21
Контрольные вопросы .....	22
2. Условия .....	23
2.1. Условный оператор if .....	23
2.2. Тернарный оператор .....	27
2.3. Вложенные условные инструкции .....	28
2.4. Логические операции .....	28
2.5. Другие виды логических операторов .....	29
Контрольные вопросы .....	30
3. Циклы и последовательности .....	31
3.1. Цикл while .....	31
3.2. Цикл For .....	33
Контрольные вопросы .....	36
4. Изменяемые и неизменяемые последовательности. Строки. ....	37
4.1. Строка как неизменяемый тип данных .....	37
4.2. Срезы строк .....	38
4.3. Специальные символы .....	40
4.4. Проход по элементам строки .....	41
4.5. Функции и методы работы со строками .....	41
4.6. Форматирование строк .....	44

Контрольные вопросы .....	47
5. Списки .....	48
5.1. Список как изменяемая последовательность .....	48
5.2. Способы объявления списка .....	49
5.3. Генератор списка .....	49
5.4. Методы для обработки списков .....	50
5.5. Изменение списков .....	51
5.6. Вложенные списки .....	53
5.7. Сортировка списка .....	54
5.8. Кортежи .....	55
Контрольные вопросы .....	56
6. Словари и множества .....	58
6.1. Словари .....	58
6.2. Множества .....	60
Контрольные вопросы .....	65
7. Функции .....	67
7.1. Понятие функции .....	67
7.2. Глобальные и локальные переменные .....	69
Контрольные вопросы .....	71
8. Обработка исключений .....	72
8.1. Исключения, конструкция try — except, finally, оператор raise .....	72
8.2. Оператор assert .....	75
Контрольные вопросы .....	76
9. Файлы. Работа с файловой системой .....	77
9.1. Работа с текстовыми файлами формата txt .....	77
9.2. Работа с файлами формата csv .....	80
9.3. Работа с файлами формата xlsw .....	85
9.4. Работа с файловой системой .....	90
Контрольные вопросы .....	96
10. Основы работы с популярными библиотеками .....	97
10.1. Библиотека Random .....	97
10.2. Библиотека Math .....	98

10.3. Библиотека Numpy .....	100
10.4. Библиотека Matplotlib .....	107
Контрольные вопросы .....	119
Список литературы .....	121

## 1. Установка Python, IDE, простые типы данных

### 1.1. Установка Python и IDE. Онлайн платформы

В данном пособии приведены примеры программ по изложенным темам. Для отработки навыка программирования рекомендуется использовать примеры, пользуясь онлайн платформами для разработки и выполнения программного кода либо установив специальные программы на компьютер.

Среди наиболее популярных онлайн-платформ выделяются:

- OnlineGDBCompiler [1]
- Google Collab [2]
- JupyterLite [3]

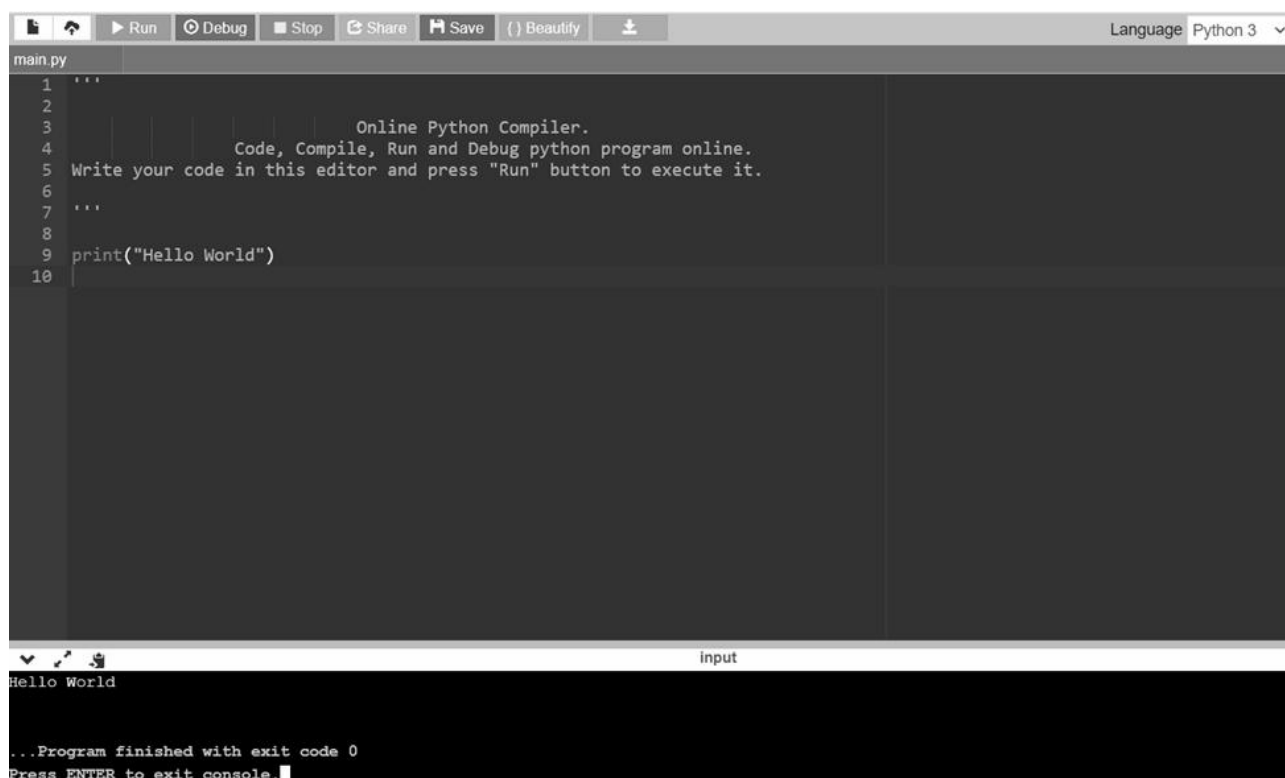


Рисунок 1.1.1 – OnlineGDBCompile

OnlineGDBCompile подойдет для более простых программ. В правом верхнем углу необходимо убедиться, что язык программирования "Python 3". Для запуска программы необходимо на верхней панели кликнуть "Run". Поле ввода и вывода – интерактивная консоль в нижней области страницы.

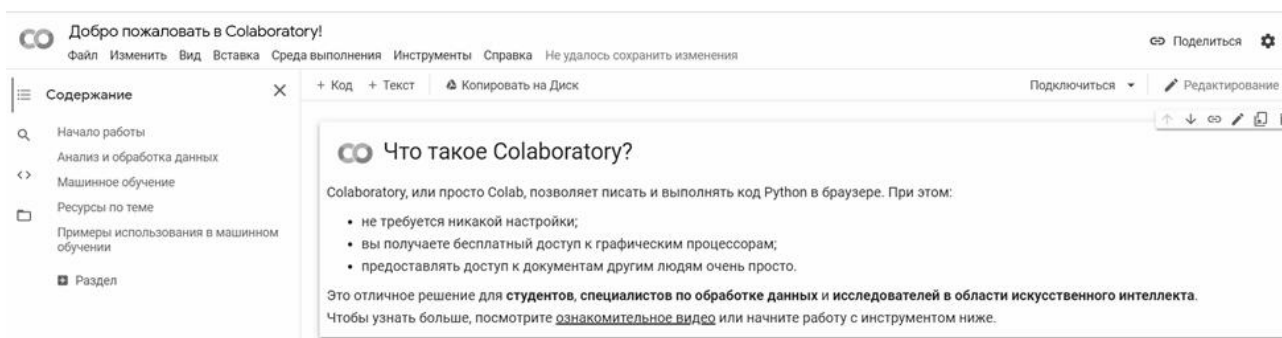


Рисунок 1.1.2 – Приветственная страница Google Colab

Google Colab позволяет писать и выполнять код Python в браузере, подходит как для простых программ, так и для более сложных проектов. Для работы необходим вход из-под аккаунта Google. Блокноты Colab будут храниться на вашем Google Диске. После нажатия "Файл" – "Создать блокнот" будет доступен блокнот Colab, позволяющий писать и выполнять код. В данном блокноте можно создать раздел Текст, который можно использовать для написания комментариев, текста заданий и т.п., либо раздел Код, который соответственно можно использовать для написания кода на языке Python. Блоков кода и текста можно создавать несколько, каждый ввод и вывод данных находится под соответствующим блоком кода.

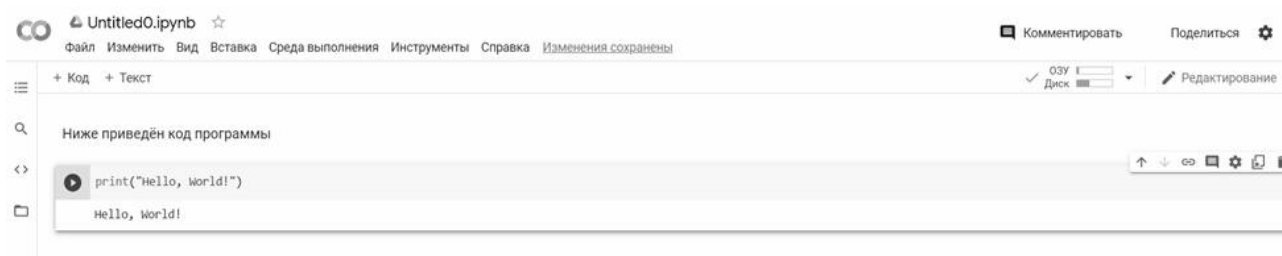


Рисунок 1.1.3 – Блокнот Google Colab

JupyterLite – удобная онлайн-платформа для выбора запуска кода единой программой, либо запуска кода частично, разделяя программу на ячейки, либо в консоле. Для выполнения программы единым файлом при создании следует выбрать "Python File", а для выполнения программы по ячейкам – "Python (Pyodide)" в разделе "Notebook".

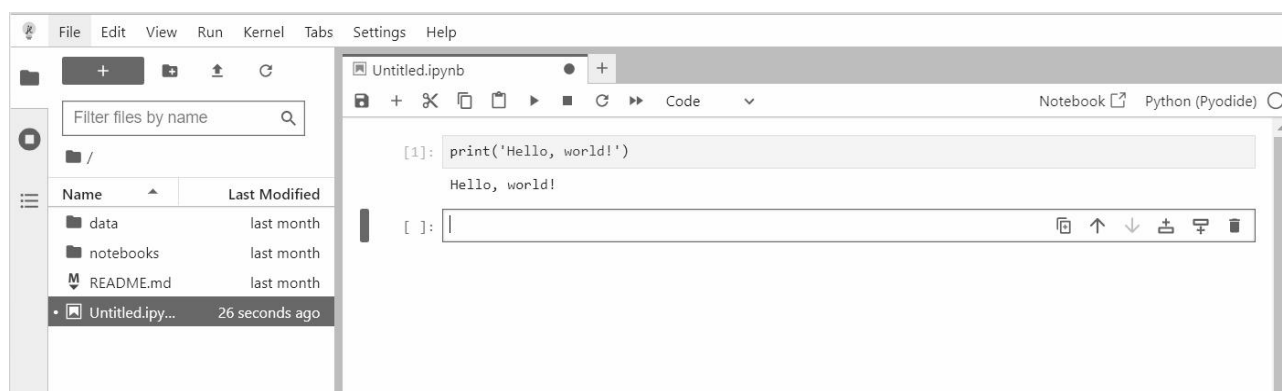


Рисунок 1.1.4 – Блокнот JupyterLite

При необходимости запуска программ на персональном компьютере следует установить специальные программы. Курс будет основан на версии Python 3.7. Скачать необходимые средства можно на сайте <https://www.python.org/>. Установка на Windows/macOS не отличается от стандартной установки программы, за исключением того, что на первом экране необходимо указать флаг "Add Python 3.7 to PATH".

После установки вам будет доступен интерпретатор языка Python через терминал, с помощью команды:

```
python
```

На Linux-base системах можно произвести установку пакетов python3, python3-pip (менеджер модулей Python) с помощью менеджера пакетов, что будет являться эквивалентом установки, представленной выше.





Рисунок 1.1.5 – Окно при установке Python.

Также программировать на Python можно используя такие среды разработки как:

- Spyder [4]
- The Jupyter Notebook [5]
- PyCharm [6]

Для перечисленных интегрированных сред разработки необходимо установить библиотеки Python.

Установка библиотек, необходимых на протяжении курса, производится с помощью утилиты `pip`, которая является системой управления пакетами. Установка производится с помощью следующей команды:

```
pip install <имя пакета>
```

Примечание: если терминал вывел предупреждение или ошибку о том, что используется старая версия `pip`, например: "You are using pip version 10.0.1, however version 18.0 is available", то необходимо выполнить следующую команду:

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

Примеры популярных пакетов, требующих установки:

- NumPy (pip install numpy) – библиотека, содержащая множество математических алгоритмов, такие как работа с матрицами, последовательностями, срезами и т.п.
- Pandas (pip install pandas) – библиотека для обработки и анализа данных.
- Matplotlib (pip install Matplotlib) – библиотека для визуализации данных двумерной (2D) графикой (3D графика также поддерживается)

## 1.2. Типы данных

В Python есть несколько стандартных типов данных:



Рисунок 1.1.6 – Стандартные типы данных.

Эти типы данных можно классифицировать по нескольким признакам:



Рисунок 1.1.7 – Классификация типов данных.

Типы данных булевы значения и числа описаны в текущей главе, остальные перечисленные типы данных описаны в соответствующих разделах.

### 1.2.1 Логический тип данных

Логический тип данных (`bool` или булевый тип) в Python представлен двумя константами `True` (истина) и `False` (ложь).

1.2.1	<pre>a = True print(type(a))  b = False print(type(b))</pre>
OUT:	<pre>&lt;class 'bool'&gt; &lt;class 'bool'&gt;</pre>

В Python данный тип данных является фундаментальным элементом для работы с логическими операциями и условными выражениями. Он предоставляет простой и интуитивно понятный способ представления истинности или ложности выражения.

В Python можно преобразовывать различные типы данных в булевый с помощью функции `bool()`.

Так же есть правила преобразования:

Ненулевые числа, пустые множества, словари, кортежи, строки, списки преобразуются в `True`.

Нулевые значения (`None`), непустые множества, словари, кортежи, строки, списки преобразуются в `False`.

Например:

1.2.2	<pre>bool(0) bool(1) bool(7 + 1 == 8)</pre>
OUT:	<pre>False True True</pre>

Таким образом, видно, что 0 соответствует значению `False`, а 1 – `True`, а результат сложения 7 и 1 равен 8 – `True`.

### 1.2.2. Числа

В языке Python для работы с различными типами чисел используются специальные классы. Например, целые числа представляются классом `int`, числа с плавающей запятой – классом `float`, а комплексные числа – `complex`. Все эти классы относятся к общей категории числовых типов данных.

Ниже приведены примеры различных типов данных:

```
1.2.2    a = 1
          b = 2.3
          c = 1 + 2j
```

В данном случае `a` – целое число, `b` – число с плавающей запятой, `c` – комплексное число.

В Python целочисленная переменная способна хранить число практически любой величины (как положительное, так и отрицательное). Интерпретатор автоматически назначает ей необходимое количество памяти для хранения результата вычислений. Это позволяет осуществлять точные вычисления с большим количеством значащих цифр без затруднений.

Чтобы работать с числами в различных системах счисления, Python предоставляет удобные инструменты. Давайте рассмотрим несколько функций, которые помогут вам переводить числа в разные системы счисления.

Начнем с функции `bin()`, которая позволяет получить двоичное представление числа.

```
1.2.4    num = 18
          res = bin(num)
          print(res)
```

```
OUT:      0b10010
```

Получившаяся число хранится в строке и имеет префикс `0b`, указывающий на основание системы счисления (двоичная).

Аналогично, функция `hex()` позволяет получить шестнадцатеричное значение, а функция `oct()` восьмеричное:

```
1.2.5    num = 18
          res1 = hex(num)
          res2 = oct(num)
          print(res1)
          print(res2)
```

```
OUT:      0x12
          0o22
```

Получившиеся строки также имеют префиксы: 0x – указывает, что число записано в шестнадцатеричной системе счисления, а 0o – число в восьмеричной.

### 1.3. Переменные

Переменная – это именованная область памяти для хранения данных, которая может изменяться в процессе исполнения программы. Python относится к языкам с динамической типизацией, что означает, что тип данных, хранимых в переменной, может быть изменен в любое время. Например, переменная может хранить числовое значение, а затем быть переназначена для хранения текста.

```
1.3.1  n = 2023
       print(type(n))

       n = 20.23
       print(type(n))

       n = 'hi!'
       print(type(n))
```

```
OUT:   <class 'int'>
       <class 'float'>
       <class 'str'>
```

Разберем код, написанный выше. В первой строке создадим переменную `n` к которой присвоим 2023, т.е. эта переменная целого типа (`int`). Для присваивания значения переменной используется `=`.

Далее с помощью метода `type(<имя переменной>)` мы можем узнать какого типа является переменная. И первый `print()`, во второй строке программы, показывает нам, что `n` действительно является `int`. Строкой ниже мы присваиваем `n` новое значение вещественного типа (`float`), это доказывает наше утверждение из первого абзаца. На самом деле переменная в Python является лишь ссылкой на объект в памяти. Когда вы создаете переменную любого типа, то в нее записывается ссылка на объект, который хранится в оперативной памяти. Возможно, что несколько переменных могут указывать на один объект. Это утверждение можно проверить с помощью метода `id(<имя переменной>)`, который возвращает адрес памяти объекта.

```
1.3.2  n = 2023
       print(id(n))
       m = 2023
       print(id(m))
```

---

OUT:	4347827120
	4347827120

---

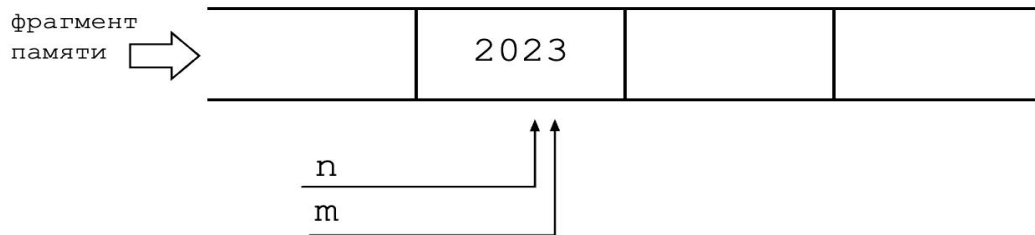


Рисунок 1.3.1 – Демонстрация переменных

Если теперь дописать в конец программы строку `n = 3`, то с объектом со значением 2023 ничего не случится, ведь он неизменяемый. Создается новый объект со значением 3, ссылка `n` отвяжется от объекта со значением 2023 и привяжется к новому объекту 3. При этом к объекту 2023 останется привязана ссылка `m`.

Если изменить и значение переменной `m`, то у объекта 2023 не останется ссылок на него. Поэтому он может быть безболезненно уничтожен при сборке мусора, ведь получить к нему доступ уже невозможно – на него не ссылается ни одна переменная.

Существует несколько стилей написания имен переменных. В некоторых языках используют `camelCase`, в котором говорится, что в имени переменных первая буква каждого слова, кроме первого, заглавная, а все остальные строчные. Например:

```
1.3.3    age = 18
         numOfCandies = 42
         phoneNumber = '+7123'
```

Но в Python обычно используют "змеиный" регистр `snake_case`, т.е. все буквы пишутся в нижнем регистре, а каждое слово отделяется нижним подчеркиванием. Например:

```
1.3.4    num_of_candies = 42
         phone_number = '+7123'
```

Хотя жесткого правила, обязывающего писать имена переменных в змеином регистре, нет, но в PEP8 (официальное руководство по стилю Python) используют именно его. Соблюдение правил, описанных в PEP8, гарантирует, что ваш код будет легко читаем и упрощает совместную работу в проекте.

В Python каждая переменная, функция, объект и другие элементы кода имеют уникальное имя, которое называется идентификатором. Идентификаторы в Python могут быть любой длины и чувствительны к регистру: "A" и "a" – это разные имена.

Идентификаторы могут состоять из букв (от "A" до "Z", как заглавных, так и строчных), цифр (но не в начале идентификатора) и символа подчеркивания "\_".

#### 1.4. Ключевые слова

Следующие ключевые слова являются зарезервированными, и ни не могут быть использованы как обычные идентификаторы (имена переменных или функций).

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

Рисунок 1.4.1 – Ключевые слова

#### 1.5. Оператор вывода

Для вывода значения на экран в языке Python существует функция `print()`. Она может быть вызвана как с одним аргументом, так и с несколькими, которые необходимо вывести. В примере ниже представлен код, который выводит строку "Hello student". Здесь "Hello student" – это аргумент передаваемый в функцию `print()`, он написан в кавычках для того, чтобы Python интерпретировал его как строку, а не что-то еще.

```
1.5.1 print("Hello student")
```

---

OUT:	Hello student
------	---------------

---

Теперь рассмотрим пример вывода нескольких значений сразу. Все переданные через запятую аргументы (могут быть разными по типу), будут выведены на одной строке. Обратим внимание на то, что все аргументы на экране будут отделяться друг от друга пробелами.

---

1.5.2	<code>print(5, "плюс", 7)</code>
-------	----------------------------------

---

OUT:	5 плюс 7
------	----------

---

Помимо строк и переменных `print()` может выводить решения выражений.

---

1.5.3	<code>print(7 + 10 * 4)</code>
-------	--------------------------------

---

OUT:	47
------	----

---

Данная функция позволяет гибко управлять форматированием выводимых данных с помощью двух именованных параметров `sep` и `end`. `sep` – сепаратор (или разредитель). По умолчанию значение `sep` – пробел. Для того чтобы изменить сепаратор, необходимо в `print` написать его новое значение. В примере ниже с помощью `print()` на экран выводятся числа от 5 до 1, и между каждым числом вместо стандартного пробела будет стоять символ `'+'`.

---

1.5.4	<code>print(5, 4, 3, 2, 1, sep='+')</code>
-------	--

---

OUT:	5+4+3+2+1
------	-----------

---

`end` – символ, который будет печататься в конце строки. По умолчанию `end` хранит символ переноса строки, поэтому вызвав `print()` несколько раз подряд значения будут выведены в разных строках. Изменить это тоже достаточно просто – указать новое значение.

---

1.5.5	<code>print(5)</code> <code>print(4)</code> <code>print(3, end='-')</code> <code>print(2, end='!')</code>
-------	--

---

OUT:	5 4 3-2!
------	----------------

---

## 1.6. Оператор ввода



Для получения данных от пользователя во время выполнения программы используется функция `input()`, которая предоставляет простой способ взаимодействия с пользователем. Она считывает ввод с клавиатуры и возвращает его в виде текстовой строки. Давайте рассмотрим пример ввода имени пользователя с клавиатуры:

1.6.1	<pre>print('Введите своё имя') name = input() print('Здравствуйте, ' + name + '!')</pre>
IN:	Иван
OUT:	Здравствуйте, Иван!

В данном случае введенное значение “Иван” будет храниться в переменной `name`, а затем выведено с помощью функции `print()`.

Для того, чтобы считать целое число с помощью функции `input()` следует привести считанное значение к типу `int` (целое число) с помощью функции `int()`. Разница типа считывания показана на примерах 1.6.2 и 1.6.3.

1.6.2	<pre>a = input() b = input()  print(type(a)) print(type(b))  print(a + b)</pre>
IN:	15 7
OUT:	<class 'str'> <class 'str'> 157

1.6.3	<pre>a = int(input()) b = int(input())  print(type(a)) print(type(b))  print(a + b)</pre>
IN:	15 7

---

```

OUT:      <class 'int'>
          <class 'int'>
          22

```

---

При необходимости ввода нескольких строковых значений через пробел существует возможность использовать метод `.split()`. В примере 1.6.4 показан ввод строки "Иван Мария", разделение по пробелу на 2 значения и их запись в соответствующие переменные (`name1 = "Иван"`, `name2 = "Мария"`).

1.6.4	<code>name1, name2 = input().split()</code> <code>print('Здравствуйте, ' + name1 + ', ' + name2 + '!')</code>
IN:	Маша Саша
OUT:	Здравствуйте Маша, Саша!

По умолчанию метод `.split()` разделяет строку по пробелам, но если нужно разделить ее по другому символу – данный символ в кавычках передается как аргумент метода. Например, вводим в программу 1.6.5 строку "кошки, собаки, лягушки".

1.6.5	<code>name1, name2, name3 = input().split(',')</code> <code>print(name1, '?', name2, '!', name3, '...')</code>
IN:	кошки, собаки, лягушки
OUT:	кошки ? собаки ! лягушки ...

## 1.7. Арифметические операции

Python предоставляет набор базовых операций, которые позволяют выполнять различные арифметические вычисления. Символы, с помощью которых выполняются математические действия называют операторами.

При выполнении нескольких математических операций в одном выражении, Python следует определенному порядку, чтобы гарантировать корректный результат. Этот порядок определяется приоритетом операций.

Таблица 1.7.1 – Приоритеты операции

Приоритет	Операция
1	Возведение в степень
2	Унарный минус (отрицание)

3	Умножение и деление (выполняются слева направо и имеют одинаковый приоритет)
4	Сложение и вычитание (выполняются слева направо и имеют одинаковый приоритет)

Для того чтобы изменить порядок действий необходимо использовать скобки.

Таблица 1.7.2 – Математические операторы

Оператор	Описание	Пример
+	Сложение	$a = 2 + 4$
-	Вычитание	$b = 4 - 2$
*	Умножение	$c = a * b$
**	Возведение в степень	$e = 2 ** 4$ ( $e = 2^4$ )

Стоит уделить особое внимание операции деления. Чтобы вычислить целую часть от деления двух чисел  $a$  и  $b$ , необходимо использовать оператор `//`.

1.7.1	<pre>a = 4 b = 3 c = a // b print(c)</pre>
OUT:	1

При этом переменная  $b$  не должна быть равна 0, иначе интерпретатор покажет ошибку.

1.7.2	<pre>a = 4 b = 0 c = a // b print(c)</pre>
OUT:	<pre>Traceback (most recent call last):   File "&lt;string&gt;", line 3, in &lt;module&gt; ZeroDivisionError: integer division or modulo by zero</pre>

Для того чтобы вычислить остаток от деления двух чисел  $a$  и  $b$ , необходимо использовать оператор `%`.

1.7.3	<pre> a = 4 b = 3 d = a % b print(d) </pre>
OUT:	1

Но обращаем внимание, что должны выполняться следующие утверждения (обозначим  $c$  – целую часть от деления  $a$  на  $b$ , а  $d$  – остаток деления  $a$  на  $b$ ):

$$a = b \times c + d$$

$$0 \leq d < b$$

1.7.4	<pre> a = -4 b = 3 c = a // b d = a % b print(c) print(d) </pre>
OUT:	-2 2

Если  $b < 0$ , то выполняются следующие утверждения:

$$a = b \times c + d$$

$$b < d \leq 0$$

1.7.5	<pre> a = 4 b = -3 c = a // b d = a % b print(c) print(d) </pre>
OUT:	-2 -2

Кроме сложения чисел существует сложение строк. Складывать число со строкой (и наоборот) нельзя, но можно воспользоваться функцией `str()`, которая переведет числовой тип данных в строковый. `Str` – это сокращение от слова `string`, которое можно перевести на русский как "строка, которая представляет собой последовательность символов". Например, задачу про вывод  $5 + 3 = 8$  можно решить и таким способом:

1.7.6	<pre>answer = '5 + 3 = ' + str(5 + 3) print(answer)</pre>
OUT:	5 + 3 = 8

## 1.8. Комментарии

Комментарии нужны для того, чтобы разработчик мог оставлять заметки в программе, например, о том, что делает данная функция, почему, зачем и что происходит в данном фрагменте кода. Это бывает полезно при работе в команде. Среди основных причин использовать комментарии можно выделить:

1. Пояснение кода: комментарии помогают другим программистам или даже вам самим лучше понять, что делает определенный участок кода.

2. Улучшение читаемости: комментарии могут помочь сделать код более понятным и легким для чтения, особенно если он сложен или содержит нетривиальные алгоритмы.

3. Документирование: комментарии могут использоваться для создания документации к коду, что поможет другим разработчикам понять, как использовать и модифицировать вашу программу.

4. Отладка: комментарии могут помочь в поиске ошибок в коде, поскольку они могут содержать информацию о том, что именно должен выполнять этот участок кода.

5. Развитие командной работы: комментарии помогают участникам команды лучше понимать код, с которым они работают, что способствует совместной разработке и поддержке программного обеспечения.

В Python строки с комментариями должны начинаться с символа `#`. Они могут занимать как всю строку, так и только часть, в этом случае символ начала комментария `#` должен стоять в строке после кода и будет распространяться до конца строки. В примере 1.8.1 приведен фрагмент кода, в котором комментарии для наглядности выделены **полужирным шрифтом**.

1.8.1	<pre><b>#Ниже переменные описывающие героя</b> name = 'Sherman' age = 20 <b>#Возраст героя</b></pre>
-------	--

Выше были описаны однострочные комментарии. Технически Python явно не поддерживает многострочные комментарии, но существует несколько обходных вариантов, решающих данных изъян.

Первый способ сделать многострочные комментарии – это объединить несколько однострочных.

1.8.2	<pre>#Артемиc version 0.7 #</pre>
-------	-----------------------------------

```
#Параметры:
#-t (--text): демонстрация текстового интерфейса
#-h (--help): показывает окно подсказки
```

Второй способ (1.8.3) проще, чем первая версия. Изначально данный способ использовался для написания документаций, но также ее можно использовать для создания многострочных комментариев.

Обычно на практике код начинается с нескольких строк комментариев, где указывают информацию о проекте, назначению файла, программисте, который работал над ним, лицензии.

```
1.8.3  """
        Артемис version 0.7

        Параметры:
        -t (--text): демонстрация текстового интерфейса
        -h (--help): показывает окно подсказки
        """
```

## Контрольные вопросы

1. Какие функции нужно использовать для ввода и вывода информации?
2. Что такое переменная? Как происходит процесс присваивания ей значения?
3. Какой приоритет операции у операции сложения?

## 2. Условия

### 2.1. Условный оператор if

Условный оператор – это ключевой элемент любого языка программирования, который позволяет программе принимать решения, т.е. выполнять разные ветви кода в зависимости от условий.

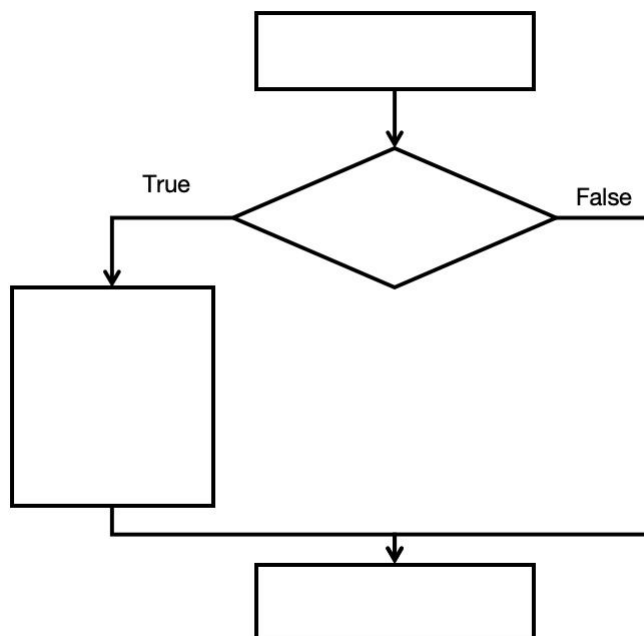


Рисунок 2.1.1 – Блок схема условного оператора if

В языке Python условный оператор if имеет следующий синтаксис:

```
if условие:  
    блок инструкций, который будет выполнен, если условие  
    истинно
```

Под условиями понимается логическое выражение, которое имеет следующий вид:

<переменная или арифметическое выражение> <знак сравнения> <переменная или арифметическое выражение>

В логических выражениях допустимы следующие знаки сравнений:

Таблица 2.1.1 – Операторы сравнения

Знак сравнения	Оператор	Пример
<	Меньше	a < b

>	Больше	c > d
<=	Меньше либо равно	e <= f
>=	Больше либо равно	g >= h
==	Равно	name1 == name2
!=	Не равно	name1 != name2

Python позволяет писать сложные логические выражения, содержащие несколько знаков сравнения. В нем все сравнения обладают одинаковым приоритетом, который меньше, чем у любой арифметической операции.

2.1.1	<pre>a = 10 b = 15 - 5 print(13 &lt; 15 &lt; 16) print(10 &lt; 60 &lt; 30) print(a &lt;= b &lt;= 10)</pre>
OUT:	<pre>True False True</pre>

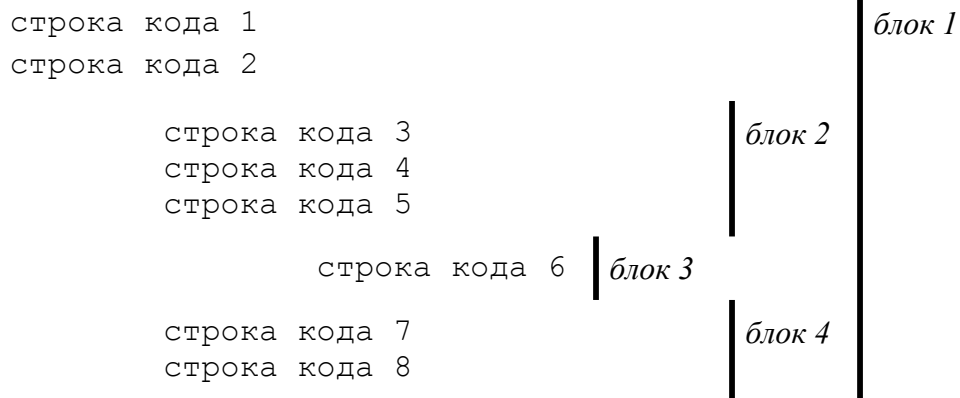
Строки также могут сравниваться между собой. При этом сравнение происходит в лексикографическом порядке (как упорядочены слова в словаре, по алфавиту).

2.1.2	<pre>check_1 = 2345 &gt; 9876 check_2 = 'dragons' &gt; 'cows' print(check_1, check_2, sep=' ')</pre>
OUT:	<pre>False True</pre>

Также логические выражения могут использоваться в циклах, и других различных конструкциях программирования для принятия решений на основе определенных условий.

Вернемся к синтаксису условного оператора. Важно отметить, что в Python для выделения блока команд, относящихся к инструкции `if`, используются отступы. Инструкции внутри одного блока должны иметь одинаковый отступ, обычно составляющий 4 пробела. Это отличие синтаксиса Python от большинства других языков программирования, где блоки выделяются специальными словами или фигурными скобками. Эта особенность синтаксиса Python обеспечивает более читаемый и структурированный код.





Так же существует другая конструкция if-else, которая предоставляет возможность выполнить один из двух блоков кода в зависимости от результата проверки условия. Это позволяет создавать программы, которые могут изменять свое поведение в зависимости от входящих данных.

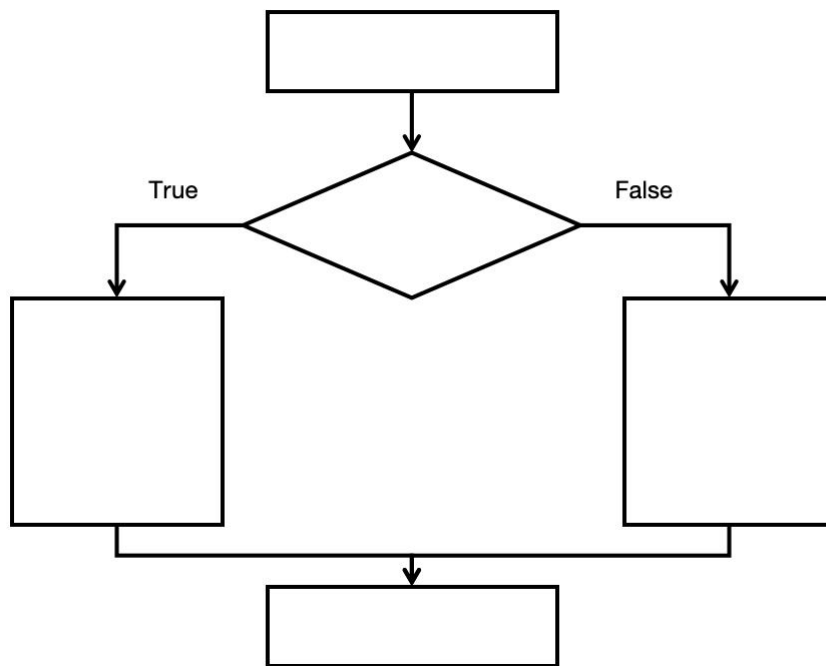


Рисунок 2.1.2 – Блок схема условного оператора if-else

Оператор if-else имеет следующий синтаксис:

```

if условие:
    блок инструкций, который будет выполнен, если условие
    истинно
else:
    блок инструкций, который будет выполнен, если условие
    ложно

```

В качестве дополнительной возможности в оператор if-else можно добавить elif, которая позволяет проверить дополнительное условие, если предыдущее было ложным.

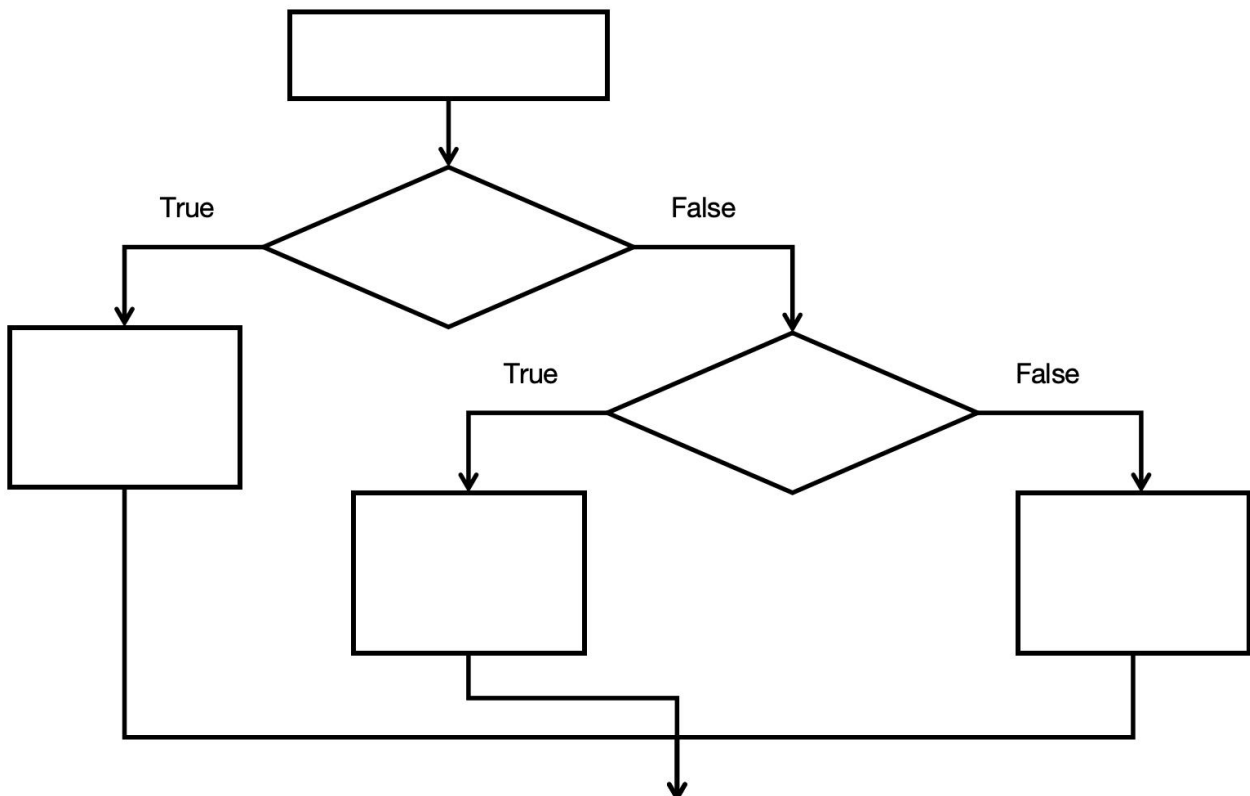


Рисунок 2.1.3 – Блок схема условного оператора if-elif-else

```
if условие 1:  
    блок инструкций, который будет выполнен, если условие 1  
    истинно  
elif условие 2:  
    блок инструкций, который будет выполнен, если условие 2  
    истинно  
else:  
    блок инструкций, который будет выполнен, если  
    предыдущие условия ложны
```

Рассмотрим пример: задача о переводе оценки из цифрового вида в словесный.

```
2.1.3 grade = int(input(Введите вашу оценку [1, 5]: ))  
if grade == 5:  
    print("Отлично!")  
elif grade == 4:  
    print("Хорошо!")  
elif grade == 3:
```

```
print("Удовлетворительно! Надо подучить материал")
else:
    print("Плохо. Стоит больше внимания уделить учебе")
```

В данной программе сперва происходит ввод оценки с клавиатуры, и преобразования ее в число (`int()`), для последующего сравнения.

Далее начинается проверка условий и вывод на экран различных строк, в зависимости от введенной оценки.

Если после `if` записано не логическое выражение, то оно будет приведено к логическому, как если бы от него была вызвана функция `bool`. Однако, злоупотреблять этим не следует, т.к. это ухудшает читаемость кода.

`elif` можно использовать несколько раз, для проверки нескольких условий.

## 2.2. Тернарный оператор

В Python существует тернарный оператор, который предоставляет возможность записывать условные выражения более компактно. Синтаксис выглядит следующим образом:

```
x = значение_если_истина if условие else значение_если_ложь
```

В этом выражении, если условие верно (истина), будет присвоено `значение_если_истина`, в противном случае будет присвоено `значение_если_ложь`. Это эквивалентно условной конструкции `'if-else'`, но записано в одной строке, что делает код более читаемым.

Пример использования тернарного оператора:

```
2.2.1  age = 25
        status = "yes" if age >= 18 else "no"
        print(status)
```

```
OUT:    yes
```

В этом примере, в зависимости от значения переменной `age` будет присвоено различное значение переменной `status`.

Тернарный оператор часто используется для коротких условных присваиваний или при выводе значения в зависимости от условия, что позволяет сделать код более компактным и удобочитаемым.

## 2.3. Вложенные условные инструкции

Часто одного условия бывает недостаточно для реализации определенной логики, поэтому используют вложенное ветвление, т. е. после первой развилки в работе программы появляется другая. При этом вложенные блоки имеют больший размер отступа (например, 8 пробелов), дополнительные 4 пробела к внешнему блоку. Например, ниже представлена программа, которая в зависимости от пола и возраста предлагает варианты подарков.

```
2.3.1 gender = input("Введите пол М или Ж :")
age = int(input("Возраст: "))

if gender == "М":
    if age <= 8:
        print("Машинки, роботы, конструктор")
    elif 9 <= age <= 18:
        print("Комиксы, сноуборд, поход на хоккей")
    else:
        print("Часы, набор туриста")
elif gender == "Ж":
    if age <= 8:
        print("Куклы, домики, мягкие игрушки")
    elif 9 <= age <= 18:
        print("Косметика, ювелирные украшения")
    else:
        print("Сертификат в SPA")
else:
    print("Некорректный ввод пола")
```

## 2.4. Логические операции

Для записи сложных логических выражений в Python используются логические операторы “и”, “или” и “не”, которые представляются символами `and`, `or` и `not` соответственно.

Операции `and` и `or` являются бинарными, т.е. требуют двух операндов, например, `x < 3 or y > 2`. Операция `not` является унарной и требует одного операнда, который записывается перед ней, например, `not x`.

В выражениях приоритет логических операций ниже, чем у операций сравнения и арифметических операций. Из логических операций наивысший приоритет имеет `not`, затем `and`, а `or` обладает наименьшим приоритетом. Для управления порядком выполнения операций, как и в арифметических выражениях, используются скобки.

Одним из примеров использования логического выражения является проверка на делимость. Например, чтобы проверить, является ли число четным, необходимо сравнить остаток от деления этого числа на два с нулём:

2.4.1	<pre>number = 1456 is_even = number % 2 == 0 print(is_even)</pre>
OUT:	True

Рассмотрим задачу, в которой требуется определить, пересекаются ли два события во времени, каждое из которых описывается годом начала и годом окончания. Для определения пересечения необходимо проверить, что начало первого события не позже конца второго, а начало второго события не позже конца первого. Например, если первое событие началось в 2020 году и закончилось в 2022 году, а второе событие началось в 2021 году и закончилось в 2023 году, то эти события пересекаются.

2.4.2	<pre>answer = start_1 &lt;= finish_2 and start_2 &lt;= finish_1 print(answer)</pre>
-------	---

## 2.5. Другие виды логических операторов

Операторы принадлежности в Python `in` и `not in`. Они проверяют, есть ли значение или переменная в последовательности (строке, списке, кортеже и т.д.). Иначе говоря, проверяют вхождение элемента в коллекцию.

2.5.1	<pre>x = 'Hello, world!' print('!' in x) print('Hello' not in x) print('Мир' not in x)</pre>
OUT:	True False True

Таблица 2.5.1 – Побитовые операторы

Оператор	Значение	Пример
&	Побитовое И	a & b
	Побитовое ИЛИ	a   b
^	Исключающее ИЛИ	a ^ b
~	Инверсия битов	~a
<<	Побитовый сдвиг влево для a на b	a << b

>>	Побитовый сдвиг вправо для a на b	a >> b
----	-----------------------------------	--------

Однако не все логические операции присутствуют в питоне, например, отсутствует логическая операция импликация, которую можно определить как операцию “<=” (Пример 2.6.2).

2.5.2	<pre>for a in 0, 1:     for b in 0, 1:         print(a, b, a &lt;= b)</pre>
OUT:	<pre>0 0 True 0 1 True 1 0 False 1 1 True</pre>

### Контрольные вопросы

1. Существует ли разница между “=” и “==”?
2. Что такое тернарный оператор?
3. Для чего использовать отступы в теле условия?
4. Чему будет равна переменная a?

<pre>a = 0 b = 123  if b &gt; 1 and a != 0:     a = 1 elif b &gt; 1:     if a &gt; - 1:         a = 2     if a &gt; 0:         a = 3     else:         a = 4 else:     a = 5</pre>	Правильный ответ: 3
--	---------------------

### 3. Циклы и последовательности

Цикл – это конструкция для организации повторения блока кода определенное количество раз или до выполнения определенного условия. В Python есть два типа циклов:

- `while` (цикл с предусловием)
- `for` (цикл с параметром)

#### 3.1. Цикл `while`

Цикл `while` повторяет блок кода пока не будет выполнено условия завершения цикла. Обычно `while` используется тогда, когда мы заранее не знаем, сколько раз должны повторять некоторое действие. Как и `if`, `while` содержит свое условие и тело, которое написано с отступами.

Синтаксис цикла `while` выглядит так:

```
while условие:
    тело цикла
```

Цикл `while` будет выполняться до тех пор, пока условие будет истинно. На каждой итерации (шаге), после окончания выполнения блока команд, управление возвращается на строку с условием и если оно выполнено, то выполнение тела цикла повторяется, а если нет – то продолжается выполнение команд, записанных после `while`.

```
3.1.1  n = 1
        while n < 10:
            print(n)
            n = n + 1
```

```
OUT:    1
         2
         3
         4
         5
         6
         7
         8
         9
```

Разберем программу, представленную выше. Сначала переменной `n = 1`. Затем идет `while`, тело цикла которого будет выполняться пока `n` меньше 10. В теле цикла 2 строки. В первой строке на экран выводится значение `n`, а во второй строке значение переменной `n` увеличивается на единицу.

Выполнение цикла происходит в девять шагов, которые расписаны ниже.

Номер шага	Значение переменной n	Проверка условия цикла	Что происходит на данном шаге
1	1	$1 < 10$ (True)	Вывод n $n = n + 1 = 2$
2	2	$2 < 10$ (True)	Вывод n $n = n + 1 = 3$
3	3	$3 < 10$ (True)	Вывод n $n = n + 1 = 4$
4	4	$4 < 10$ (True)	Вывод n $n = n + 1 = 5$
5	5	$5 < 10$ (True)	Вывод n $n = n + 1 = 6$
6	6	$6 < 10$ (True)	Вывод n $n = n + 1 = 7$
7	7	$7 < 10$ (True)	Вывод n $n = n + 1 = 8$
8	8	$8 < 10$ (True)	Вывод n $n = n + 1 = 9$
9	9	$9 < 10$ (True)	Вывод n $n = n + 1 = 10$
10	10	$10 < 10$ (False)	вывод не происходит увеличение значения переменной не происходит

Закрепим полученные знания на задаче суммирования цифр в числе.

Обычно данные задачи решаются арифметическим способом, т.е. над ним должны выполняться определенные арифметические операции для извлечения из него цифр, которые будут суммироваться в другую переменную.

Чтобы извлекать цифры из числа будем использовать деление нацело и нахождение остатка. Если разделить число нацело, то произойдет удаление из числа последней цифры (например, дано число 123, если его нацело разделить на 10, то получится число 12). Но перед удалением из числа последней цифры, необходимо ее запомнить, это делается нахождением остатка (например, также



дано число 123, если его разделить на 10, то остатком деления будет последняя цифра – 3), получив эту цифру мы можем ее добавить к сумме цифр.

3.1.2	<pre>num = int(input()) #число, введенное пользователем sum = 0            #переменная для подсчета суммы цифр  while num &gt; 0:     #пока, введенное пользователем     temp = num % 10 #число больше 0     num = num // 10 #извлекаем последнюю цифру     sum = sum + temp #удаляем из числа последнюю цифру                     #прибавляем к сумме извлеченную                     #цифру</pre>
IN:	123
OUT:	6

Еще цикл `while` используют в качестве бесконечного цикла, который будет выполняться до тех пор, пока `break` внутри него не сработает. `Break` – инструкция прерывания цикла. Ниже приведен пример программы, в которой пользователь будет вводить цифры до тех пор, пока не введет 9.

3.1.3	<pre>num = int(input('Введите цифру от 0 до 9'))  while True:     num = int(input('Введите цифру от 0 до 9'))     if num == 9:         break  print(num)</pre>
OUT:	9

В условии цикла стоит просто `True` (Истина), из этого следует, что код внутри будет выполняться всегда, поэтому называют бесконечным. Но он будет завершен с помощью `break`, в том случае если `num` станет равной 9.

Также существует еще и другая инструкция управления циклом – `continue` (продолжение цикла). Если `continue` встречается в середине тела цикла, то все остальные инструкции до конца цикла пропускаются, и цикл продолжается со следующей итерации.

## 3.2. Цикл For

Цикл `for` в Python используется для итерации по элементам последовательностей. Он состоит из ключевого слова `for`, имени переменной и последовательности, по которой происходит итерация. Общий синтаксис цикла `for` в Python выглядит так:

```
for переменная in последовательность:  
    тело цикла
```

Например, перед нами стоит задача вывести на экран цвета светофора:

```
3.2.1  for color in ('красный', 'желтый', 'зеленый'):  
        print(color)
```

```
OUT:   красный  
        желтый  
        зеленый
```

В данном примере ('красный', 'желтый', 'зеленый') является кортежем, состоящем из 3-х строк. На каждой итерации на место переменной `color` будут поочередно подставляться значения из кортежа.

Также как и `while` все действия, которые должны выполняться в `for`, должны выделяться отступом.

Цикл `for` используют для повторения команд определенное количество раз. Например, чтобы вывести все цифры на экран можно сделать следующее:

```
3.2.2  for c in (0, 1, 2, 3, 4, 5, 6, 7, 8, 9):  
        print(c, end=' ')
```

```
OUT:   0 1 2 3 4 5 6 7 8 9
```

Чтобы не перечислять все цифры, можно воспользоваться функцией `range()`, которая позволяет генерировать последовательности.

```
3.2.3  for c in range(10):  
        print(c, end=' ')
```

```
OUT:   0 1 2 3 4 5 6 7 8 9
```

Функция `range(n)` создает последовательность чисел от 0 до  $n-1$ .

Существует вариант `range()` с двумя параметрами `range(a, b)`, где  $a$  – число, с которого начинается последовательность,  $(b - 1)$  – число которым заканчивается.

Также существует `range()` с тремя параметрами `range(a, b, step)`, где `step` – шаг, с которым будут сгенерированы числа от `a` до `b - 1`.  
Пример:

3.2.4	<code>range(1, 100, 10)</code> – функция сгенерирует следующую последовательность: (1, 11, 21, 31, 41, 51, 61, 71, 81, 91)
-------	--

Ниже представлены примеры программ, в которых одна написана без использования циклов, в другой использован цикл `while`, а в последней решается та же задача только с использованием цикла `for`.

3.2.5 Без циклов	<pre> budget = 0 budget += 50000 budget += 50000 budget += 50000 budget += 50000 budget += 50000 budget += 50000 budget += 50000 budget += 50000 print(budget) </pre>
C while	<pre> budget = 0 i = 0  while i &lt; 8:     budget += 50000     i += 1  print(budget) </pre>
C for	<pre> budget = 0  for i in range(8):     budget += 50000  print(budget) </pre>

Можно заметить, что код, написанный с использованием цикла `for` занимает меньше строк, чем остальные вариации программы.

Инструкции `break` и `continue` работают точно так же, как и при работе с циклом `while`.

Часто тело цикла включает в себя другой цикл, который называют вложенным. Возможно создание любого количества вложенных циклов. При переходе на каждый новый уровень вложенности необходимо увеличивать отступ.

Вложенные циклы используются, когда необходимо выполнить несколько циклов одновременно или когда один цикл должен быть выполнен внутри другого цикла. Например, при генерации комбинаций символов или чисел, обработке списков или матриц, анализе данных и других задачах.

В следующем фрагменте кода цикл `for` вложен в тело другого цикла `for`. Данная программа будет выводить на экран треугольник из цифр:

```
3.2.6   for i in range(5):  
        for j in range(i + 1):  
            print(j, end = ' ')  
        print()
```

```
OUT:    0  
        0 1  
        0 1 2  
        0 1 2 3  
        0 1 2 3 4
```

## Контрольные вопросы

1. Что такое итерация?
2. Как написать бесконечный цикл?
3. Для чего в цикле `for` используют функцию `range()`?
4. Сколько символов "\*" будет выведено на экран?

```
for i in range(1, 5, 2):  
    print('*' * i)
```

Правильный ответ: 4

## 4. Изменяемые и неизменяемые последовательности. Строки.

### 4.1. Строка как неизменяемый тип данных

Строка в Python - это неизменяемый упорядоченный тип данных, состоящий из символов, заключенных в кавычки. Можно использовать одинарные или двойные кавычки для создания строки главное, чтобы с обеих сторон от содержимого строки кавычек было одинаковое количество. Для создания строк, которые занимают несколько строк кода, используются тройные кавычки `'''` или `"""`:

```
4.1.1  s_1 = "simple string"

        s_2 = '''multi
        multi
        multiline
        string'''

        print(s_1)
        print(s_2)
```

```
OUT:    simple string
        multi
        multi
        multiline
        string
```

Стоит отметить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку. Помимо создания и вывода строк на экран с ними можно выполнять следующие операции:

1. Строки можно складывать (или склеивать или конкатенировать). В результате этой операции возникает новая строка, которая содержит все символы из исходных строк. Создание третьей переменной происходит из-за того, что строки неизменяемы, поэтому мы не изменяем строку, вместо этого мы объединяем их вместе и присваиваем новой переменной. Пример конкатенации двух строк представлен в 4.1.2.

```
4.1.2  s_1 = 'В мире'
        s_2 = 'животных'
        s = s_1 + s_2
        print(s)
```

```
OUT:    В мирезживотных
```

2. Строку можно умножать на число. Это удобный способ повторить строку несколько раз. Данная операция создает новую строку, которая содержит исходную строку, повторенную указанное количество раз.

4.1.3	<pre>s = '*' * 7 print(s)</pre>
OUT:	*****

## 4.2. Срезы строк

Срезы являются мощным инструментом для работы со строками, позволяющий выделять подстроки и извлекать отдельные символы из них. Так как строки являются упорядоченным типом данных, это позволяет обращаться к символам в строке по порядковому номеру, который называется индексом. Этот номер указывает на позицию символа в строке. Для получения элемента строки используют `[n]` квадратные скобки, внутри которых указывается индекс элемента. Рассмотрим пример 4.2.1 и рисунок.

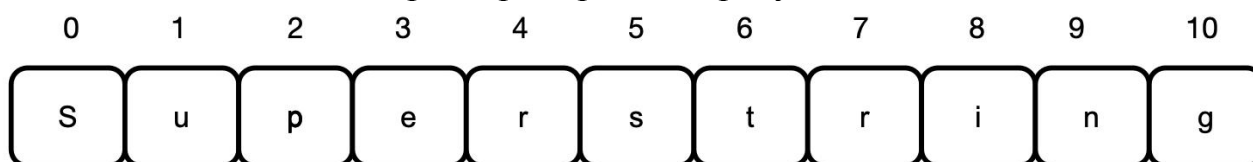


Рисунок 4.2.1 – Положительная индексация строки

4.2.1	<pre>st = 'superstring' print(st[4])</pre>
OUT:	r

Нумерация всех символов в строке идет с нуля. Но, если нужно обратиться к какому-то по счету символу, начиная с конца, то можно указывать отрицательные значения (на этот раз с единицы). Ниже приведен рисунок 4.2.2 и пример 4.2.2.

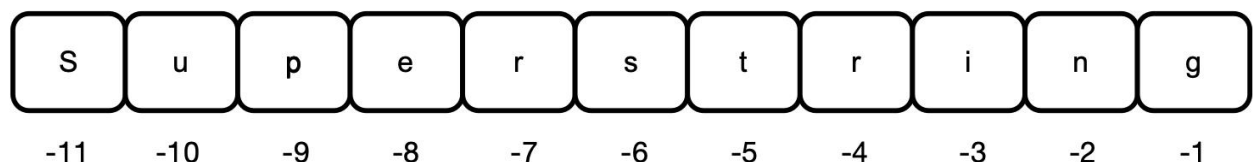


Рисунок 4.2.2 – Отрицательная индексация строки

4.2.2	<pre>st = 'superstring' print(st[1])</pre>
-------	--

	<code>print(st[-1])</code>
OUT:	u g

Помимо прямого доступа к определенному символу в строке, можно извлекать ее части, используя указание диапазона индексов. Такое извлечение называется срезом. Срез строки включает символы от начального номера до конечного, не включая сам конечный номер. Ниже представлен пример среза 4.2.3 и рисунок 4.2.3.

4.2.3	<code>st = 'superstring'</code> <code>print(st[0:5])</code>
OUT:	super

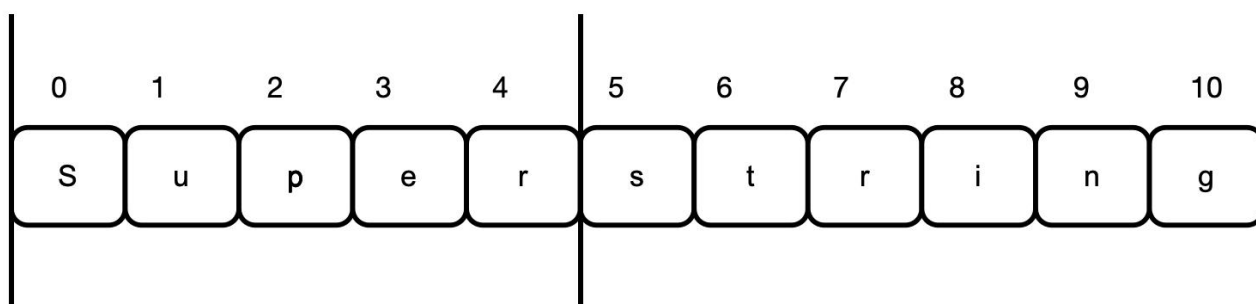


Рисунок 4.2.3 – Отрицательная индексация строки

При отсутствии первого числа в срезе, он начинается с самого начала строки. Если не указано второе число, то срез продолжается до конца строки. Таким образом, без указания ни первого, ни второго числа в квадратных скобках, выводится вся строка целиком. В примере 4.2.4 показаны такие варианты срезов.

4.2.4	<code>st = 'superstring'</code> <code>print(st[:5], (st[5:], sep=';'))</code> <code>print(st[:])</code>
OUT:	super;string superstring

Допустим вам необходимо сделать срез из последних трех символов строки:

4.2.5	<code>st = 'superstring'</code> <code>print('superstring'[-3:])</code>
-------	---

---

OUT:	ing
------	-----

---

При использовании срезов вы также можете задать шаг. Это позволяет извлекать элементы с определенным интервалом. Например, чтобы получить только символы, стоящие на нечетных индексах, необходимо использовать шаг 2:

4.2.6	<pre>a = 'superstring' print(a[1::2])</pre>
-------	---

---

OUT:	uesrn
------	-------

---

Дословно запись `a[1::2]` значит, что получаем символы из строки `a`, начиная с 1го символа и до конца строки с шагом 2 (т.е. каждый 2ой символ, начиная с 1го). Помним, что нумерация начинается с 0.

А таким образом можно получить все символы, стоящие на четных позициях строки `a`:

4.2.7	<pre>a = 'superstring' print(a[::2])</pre>
-------	--

---

OUT:	sprtig
------	--------

---

Срез с шагом -1 можно использовать для получения строки в обратном порядке:

4.2.8	<pre>a = '0123456789' print(a[::-1]) print(a[::-1])</pre>
-------	---

---

OUT:	superstring gnirtsrepus
------	----------------------------

---

### 4.3. Специальные символы

Строка может содержать ряд специальных символов – управляющих последовательностей или escape-последовательности. Некоторые из них:

Таблица 4.3.1 – Специальные символы

Символ	Описание
<code>\</code>	для добавления обратной косой черты ( <code>\</code> ) (слэша)
<code>\'</code>	для добавления одинарной кавычки



<code>\''</code>	для добавления двойной кавычки
<code>\n</code>	переводит курсор на следующую строку
<code>\t</code>	вставляет символ табуляции (4 отступа)

#### 4.4. Проход по элементам строки

Часто возникает задача обработки каждого символа в строке. Например, может потребоваться заменить определенные символы или подсчитать количество вхождений определенного символа. Вы можете использовать цикл `for` для итерации по каждому символу в строке.

4.4.1	<pre>string = "hello" for char in string:     print(char)</pre>
OUT:	<pre>h e l l o</pre>

Аналогично можно получить каждый символ с помощью цикла `for` и прохода по индексам, это популярно в тех случаях, когда важны порядковые номера символов:

4.4.2	<pre>string = "world" for i in range(0, len(string)):     print(string[i])</pre>
OUT:	<pre>w o r l d</pre>

В примере 4.4.2 функция `len(string)` возвращает длину строки `string`, а `range(len(s))` генерирует последовательность чисел от 0 до `len(string) - 1`. Затем каждое число используется как индекс для доступа к символам строки `string`.

#### 4.5. Функции и методы работы со строками

Отличие функция и методов для работы со строками заключается в том, функция является независимой от конкретного объекта и может быть вызвана как отдельно, так и для любой строковой переменной, в то время как метод применяется к конкретному объекту (в данном случае, к строке) и вызывается через оператор точки после имени переменной.

Таблица 4.5.1 – Функции строк

Функция	Описание	Пример
<code>len()</code>	Вычисления длины строки	<pre>s = "world" print(len(s))</pre> <p>Результат: 5</p>
<code>max()</code>	Нахождение максимального символа в строке (по алфавиту)	<pre>s = "world" print(max(s))</pre> <p>Результат: w</p>
<code>min()</code>	Нахождение минимального символа в строке (по алфавиту)	<pre>s = "world" print(min(s))</pre> <p>Результат: d</p>

Проверка типов элементов в строке выполняется следующими методами:

Таблица 4.5.2 – Методы строк

Метод	Описание
<code>isdigit()</code>	Возвращает True, если строка состоит только из цифр
<code>isalpha()</code>	Возвращает True, если строка состоит только из букв
<code>isalnum()</code>	Возвращает True, если строка состоит из букв или цифр
<code>isupper()</code>	Возвращает True, если строка содержит символы только верхнего регистра
<code>islower()</code>	Возвращает True, если строка содержит символы только нижнего регистра
<code>isascii()</code>	Возвращает True, если строка содержит только ASCII-символы
<code>isspace()</code>	Возвращает True, если строка содержит только пробельные символы
<code>istitle()</code>	Возвращает True, если ли каждое из «слов» строки начинается с заглавной буквы

Примеры использования перечисленных методов:

```
4.5.2 print("1a0".isdigit(), end=" ")
      print("10".isdigit(), end="\n\n")
```

```

print("a".isalpha(), end=" ")
print("a100".isalpha(), end="\n\n")

print("a10_".isalnum(), end=" ")
print("a10".isalnum(), end="\n\n")

print("Hello".isupper(), end=" ")
print("HELLO".isupper(), end="\n\n")

print("Hello".islower(), end=" ")
print("hello".islower())

```

OUT:      True False

         False True

         False True

         False True

Таблица 4.5.2 – Методы строк

Метод	Описание	Дополнительно
capitalize()	Метод создает копию строки, переводя первую буквы в верхний регистр, а остальные в нижний	
count(sub)	Метод подсчитывает количество непересекающихся вхождений в неё указанной подстроки sub	s.count(sub, start, end) sub – искомая подстрока, start – начальная позиция в строке для поиска подстроки, end – конечная позиция в строке для поиска подстроки
find(sub)	Метод возвращает наименьший индекс, с которого начинается указанная подстрока sub в исходной . Если подстрока не найдена – возвращает -1	s.find(sub, start, end) sub – искомая подстрока, start – начальная позиция в строке для поиска подстроки, end – конечная позиция в строке для поиска подстроки
rfind(sub)	Метод возвращают индекс последнего вхождения искомой подстроки sub. Если подстрока не найдена – возвращает значение -1	s.rfind(sub, start, end) sub – искомая подстрока, start – начальная позиция в строке для поиска подстроки, end – конечная позиция в строке для поиска подстроки
index(sub)	Метод возвращает наименьший индекс, с	s.index(sub, start, end) sub – искомая подстрока,

	которого начинается подстрока <code>sub</code> в исходной. В случае отсутствия подстроки в исходной строке, генерируется исключение.	<code>start</code> – начальная позиция в строке для поиска подстроки, <code>end</code> – конечная позиция в строке для поиска подстроки
<code>lower()</code>	Метод возвращает копию исходной строки с символами, приведенными к нижнему регистру	
<code>upper()</code>	Метод возвращает копию исходной строки с символами, приведенными к верхнему регистру	
<code>lstrip(chars)</code>	Метод возвращает копию указанной строки, с начала (слева <code>l</code> – <code>left</code> ) которой устранены указанные символы	<code>chars</code> – строка с символами, которые требуется устранить. Если не указана, будут устранены пробельные символы
<code>replace(old, new)</code>	Метод возвращает копию строки, в которой заменены все вхождения указанной строки <code>old</code> указанным значением <code>new</code>	<code>s.replace(old, new, count)</code> <code>old</code> – подстрока, которую следует заменить. <code>new</code> – подстрока, на которую следует заменить искомую. <code>maxcount</code> – максимальное требуемое количество замен. Если не указано, будут заменены все вхождения искомой строки
<code>split()</code>	Метод разбивает строку на части, используя разделитель, и возвращает эти части списком. Направление разбиения: слева направо. По умолчанию разделяет по пробелам	<code>s.split(sep, maxsplit)</code> <code>sep</code> – строка-разделитель. Может содержать как один, так и несколько символов. <code>maxsplit</code> – максимальное количество разбиений, которое требуется выполнить. Если <code>-1</code> , то количество разбиений не ограничено
<code>strip()</code>	Метод возвращает копию указанной строки, с обоих концов которой устранены указанные символы	<code>s.strip(chars)</code> <code>chars</code> – строка с символами, которые требуется устранить. Если не указана, будут устранены пробельные символы
<code>swapcase()</code>	Метод возвращает копию строки, в которой каждая буква будет иметь противоположный регистр.	

## 4.6. Форматирование строк

Форматирование строк в Python – это процесс вставки значений в строки с использованием различных методов и операторов. Основные способы форматирования строк включают использование оператора %, метода `format()`, f-строк.

Оператор % – самый простой способ форматирования строк, который позволяет вставить значения в строки используется знак процента и спецификаторы формата.

Таблица 4.6.1 – Спецификаторы формата

Спецификатор	Описание
%s	Задаёт строку
%d	Задаёт целое число
%f	Задаёт дробное число
%<к_пробелов>.<к_знаков>f	Задаёт дробное число <к_пробелов> – количество пробелов, которые нужно добавить перед печатью числа <к_знаков> – количество знаков после запятой
%x/%X	Задаёт шестнадцатеричное представление значения
%o	Задаёт восьмеричное представление значения

Пример:

4.6.1	<pre>name = 'Кирилл' date = 20  print('Уважаемый, %s. Вы приглашены %d мая на день первокурсника.' % (name, date))</pre>
OUT:	<pre>Уважаемый, Кирилл. Вы приглашены 20 мая на день первокурсника.</pre>

Следующий способ форматирования строк с помощью метода `format()`. Данный метод принимает позиционные и именованные аргументы, которые заменяются соответствующими значениями во время выполнения программы. Вот пример использования метода:

4.6.2	<pre>name = 'Кирилл' date = 20</pre>
-------	--------------------------------------

	<pre>print('Уважаемый, {}. Вы приглашены {} мая на день первокурсника.'.format(name, date))</pre>
OUT:	Уважаемый, Кирилл. Вы приглашены 20 мая на день первокурсника.

Данный метод также поддерживает использование именованных аргументов, что делает код более понятным.

4.6.3	<pre>name = 'Кирилл' date = 20  print('Уважаемый, {n}. Вы приглашены {d} мая на день первокурсника.'.format(n=name, d=date))</pre>
OUT:	Уважаемый, Кирилл. Вы приглашены 20 мая на день первокурсника.

Еще одним способом является передача аргументов в метод `format()` по очереди и вставка их в формируемую строку с указанием номера аргумента в фигурных скобках (нумерация начинается с нуля).

4.6.4	<pre>name = 'Кирилл' date = 20 print('Уважаемый, {0}. {0} приглашен {1} мая на день первокурсника.'.format(name, date))</pre>
OUT:	Уважаемый, Кирилл. Кирилл приглашен 20 мая на день первокурсника.

В этом примере `{0}` относится к первому аргументу 'Кирилл', `{1}` – 20.

F-строки — это новый способ форматирования строк в Python, который был добавлен в версии 3.6. Они позволяют выполнять вызовы функций и методов прямо внутри строки, что делает код более компактным и понятным. F-строки записываются с символом «f» перед открывающей кавычкой и содержат выражения внутри фигурных скобок, где указываются имена переменных для подстановки.

4.6.5	<pre>name = 'Кирилл' date = 20  print(f'Уважаемый, {name}. {name} приглашен {age} мая на день первокурсника.')</pre>
OUT:	Уважаемый, Кирилл. Кирилл приглашен 20 мая на день первокурсника.

F-строки, как и метод `format()` поддерживают расширенное форматирование чисел. Значение, двоеточие, затем ширина строки в фигурных скобках, точка, требуемая точность в фигурных скобках.

<b>10</b>	<b>.</b>	<b>5</b>	<b>f</b>
ширина символов числа		количество знаков после запятой	

```
4.6.6 pi = 3.1415926535

print(f'Число пи = {pi}')
print(f'Число пи = {pi:.2f}')
print(f'Число пи = {pi:10.5f}')

print('Число пи = {}'.format(pi))
print('Число пи = {:.2f}'.format(pi))
print('Число пи = {:10.5f}'.format(pi))
```

```
OUT: Число пи = 3.1415926535
      Число пи = 3.14
      Число пи = 3.14159
      Число пи = 3.1415926535
      Число пи = 3.14
      Число пи = 3.14159
```

## Контрольные вопросы

1. Как узнать, содержит ли строка только алфавитные символы в Python?
2. Как вывести на экран в обратном порядке три последних символа строки.
3. Каким способом узнать есть ли в строке “Сегодня я узнал о строках” слово “я”?
4. Что будет выведено на экран?

```
s = "Сессия"
s = s.swapcase()
print(s.count('C'))
```

Правильный ответ: 2

## 5. Списки

### 5.1. Список как изменяемая последовательность

Список в Python является аналогом массивов в других языках программирования, предоставляющий способы хранения и работы с коллекциями данных. Они представляют собой упорядоченные, изменяемые последовательности элементов. Список представляет собой набор ссылок на объекты.

Характеристики списков:

1. Упорядоченность. Элементы в списке расположены в определенном порядке и проиндексированны.

2. Изменяемость. Присутствует возможность изменять содержимое списка после его создания, добавляя, удаляя или изменяя элементы.

3. Разнотипность. Список может содержать элементы разных типов данных - строки, числа, булевы значения, объекты и даже другие списки.

В примере 5.1.1 показан список с числами от 1 до 10:

```
5.1.1 my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      print(my_list)
```

```
OUT:    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

К спискам также применима функция `len()` и срезы, которые работают также как в строках.

Для списков характерна изменяемость, то есть можно взять определенный элемент списка и изменить его, как показано на примере 5.1.2.

```
5.1.2 my_list = [1, 2, 3]
      my_list[1] = 4
      print(my_list)
```

```
OUT:    [1, 4, 3]
```

Для вывода элементов списка без внешних скобок следует перед названием списка поставить символ `*`:

```
5.1.3 my_list = [0, 9, 8, 7, 6, 5]
      print(*my_list)
```

```
OUT:    0 9 8 7 6 5
```



## 5.2. Способы объявления списка

Объявить (создать) пустой список можно с помощью функции `list()` или с помощью пустых квадратных скобок `[]`:

```
5.2.1  my_list_1 = list()
       my_list_2 = []
       print(my_list_1, my_list_2)
```

OUT: [] []

---

При необходимости создать список, состоящих из `n` нулей можно воспользоваться методом `append()` для добавления нулей в пустой список, либо при создании указать, что нулевой элемент повторяется в списке `n` раз:

```
5.2.2  n = 5
       my_list_1 = [0] * n
       n = 10
       my_list_2 = list()
       for i in range(n):
           my_list_2.append(0)

       print(my_list_1, my_list_2, sep='\n')
```

OUT: [0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

---

Список можно создать из строки, множества или кортежа (см. раздел 6) с помощью функции `list()`:

```
5.2.3  my_tuple = (3, 4, 5)
       my_str = '012'
       my_set = [6, 7, 8]
       print(list(my_tuple))
       print(list(my_str))
       print(list(my_set))
```

OUT: [3, 4, 5]
 ['0', '1', '2']
 [6, 7, 8]

---

## 5.3. Генератор списка

Генератор списка – это удобный способ создания списков на основе какого-либо выражения или итерации. Генератор списка позволяет создавать список из другого списка, строки или любой другой итерируемой последовательности, применяя к каждому элементу какое-либо выражение.

Синтаксис генератора списка выглядит следующим образом:

```
[выражение for элемент in последовательность if условие]
```

где:

выражение – любое вычисляемое выражение.

элемент – переменная, которая пробегает по элементам последовательности.

последовательность – итерируемая последовательность (например, список, кортеж, строка и т.д.).

условие (необязательно) – фильтр, который определяет, останется ли элемент в итоговом списке.

Пример использования генератора списка:

```
5.3.1  nums = [1, 2, 3, 4, 5]
       denominators = [1 / x for x in nums if x % 2 == 0]
       print(denominators)
```

OUT: [0.5, 0.25]

В данном примере создается список чисел `nums`. Генератор списка `denominators` создаёт список из чётных элементов `x` списка `nums`, которые будут знаменателями в дроби  $1/x$ . Если убрать условие четности, то все элементы списка `nums` войдут в список `denominators` после деления  $1/x$ :

```
5.3.2  nums = [1, 2, 3, 4, 5]
       denominators = [1 / x for x in nums]
       print(denominators)
```

OUT: [1.0, 0.5, 0.3333333333333333, 0.25, 0.2]

## 5.4. Методы для обработки списков

Таблица 5.4.1 – Методы, не изменяющие список и возвращающие значение

Метод	Описание
<code>count(x)</code>	Подсчитывает число вхождений значения <code>x</code> в список
<code>index(x)</code>	Находит позицию первого вхождения значения <code>x</code> в список
<code>index(x, from)</code>	Находит позицию первого вхождения значения <code>x</code> в список, начиная с позиции <code>from</code>

Таблица 5.4.2 – Методы, не возвращающие значение, но изменяющие список

Метод	Описание
<code>append(x)</code>	Добавляет значение <code>x</code> в конец списка
<code>extend(other_list)</code>	Добавляет все содержимое списка <code>other_list</code> в конец списка. В отличие от операции <code>+</code> изменяет объект, к которому применен, а не создает новый
<code>insert(index, x)</code>	Вставляет число <code>x</code> в список так, что оно оказывается на позиции <code>index</code> . Число, стоявшее на позиции <code>index</code> и все числа правее него сдвигаются на один вправо
<code>remove(x)</code>	Удаляет первое вхождение числа <code>x</code> в список
<code>reverse()</code>	Разворачивает список (меняет значение по ссылке, а не создает новый список как <code>myList[::-1]</code> )

Таблица 5.4.3 – Методы, возвращающие значение и изменяющие список

Метод	Описание
<code>pop()</code>	Возвращает последний элемент списка и удаляет его
<code>pop(index)</code>	Возвращает элемент списка на позиции <code>index</code> и удаляет его

## 5.5. Изменение списков

Рассмотрим такой пример:

```
5.5.1  a = ['cat', 'dog', 'rabbit']
        b = a
        b[0] = 100
        print(a)
OUT:    [100, 'dog', 'rabbit']
```

В первой строке создается список `a`, содержащий три строки `'cat'`, `'dog'`, `'rabbit'`.

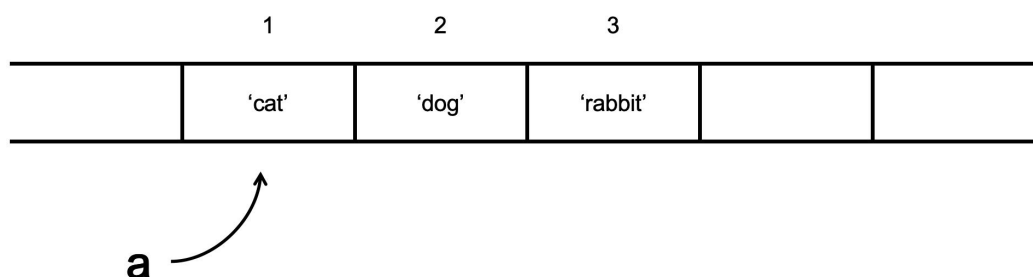


Рисунок 5.5.1 – Список a в памяти

Во второй строке происходит присваивание ссылки: `b = a`, т.е. python не копирует список a в b, а создает ссылку на него. Теперь переменные a и b указывают на один и тот же список в памяти (рисунок 5.5.2).

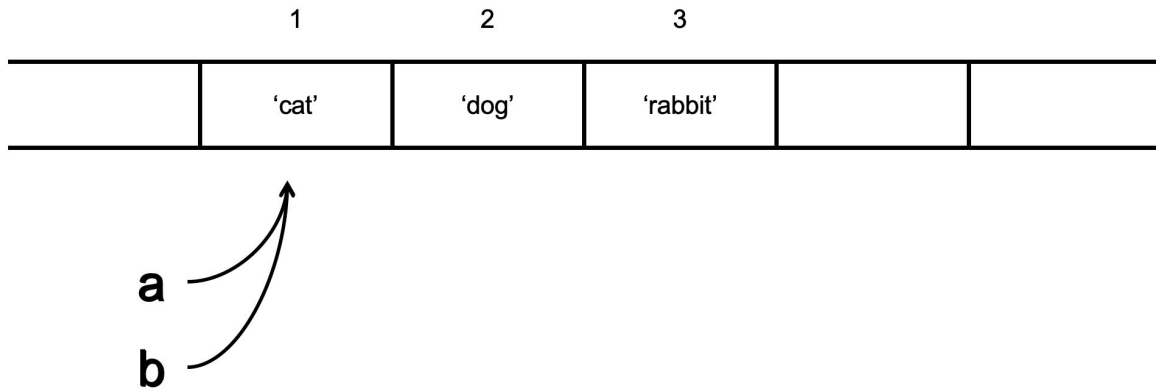


Рисунок 5.5.2 – Список a и b в памяти

В третьей строке происходит изменение элемента: `b[0] = 100` изменение первого элемента списка. Так как a и b ссылаются на один и тот же список, изменение через b также отражается в a.

Рассмотрим следующий случай:

```
5.5.2  a = ['cat', 'dog', 'rabbit']
        b = ['cat', 'dog', 'rabbit']
        a[0] = 1000
        print(b)
```

OUT: ['cat', 'dog', 'rabbit']

Если же написать программу 5.5.2. то будет выведено `['cat', 'dog', 'rabbit']`. Несмотря на то, что объекты имеют одинаковое значение из-за их изменяемости для каждого значения будет создан отдельный объект и ссылки a и b будут показывать на разные объекты (рисунок 5.5.3). Изменение одного из них, естественно, не приводит к изменению другого.

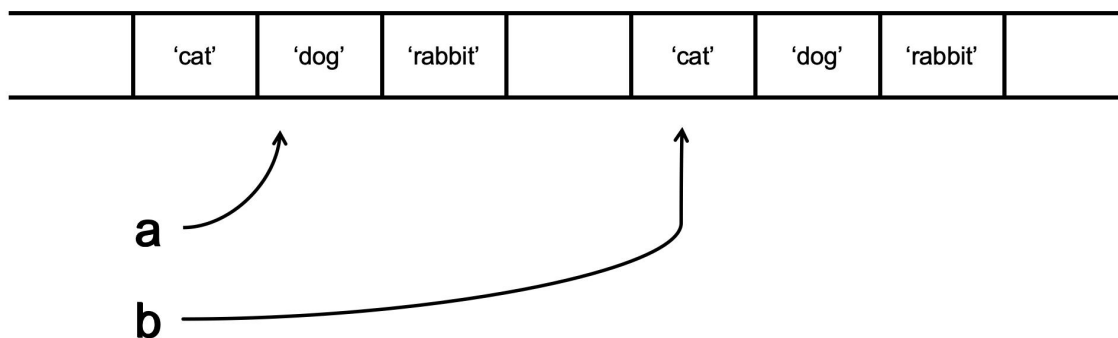


Рисунок 5.5.3 – Список a и b в памяти

## 5.6. Вложенные списки

Вложенные списки в Python представляют собой списки, которые содержат в себе в качестве элементов другие списки. То есть каждый элемент внешнего списка является сам по себе списком. Например, вот такая структура:

```
5.6.1 my_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

В примере 5.6.1 `my_list` – это вложенный список, который содержит три элемента, каждый из которых является отдельным списком. Таким образом, вложенные списки позволяют удобно хранить и обрабатывать структурированные данные, такие как матрицы, таблицы и др.

Можно использовать генератор списка: создадим список из  $n$  элементов, каждый из которых будет списком, состоящих из  $m$  нулей:

```
5.6.2 n = 4
      m = 3
      my_list = [[0] * m for i in range(n)]
      print(my_list)
OUT:  [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Считывание вложенных списков происходит несколькими вариантами. Если элементы списка вводятся через клавиатуру (каждая строка на отдельной строке, всего  $n$  строк, числа в строке разделяются пробелами), для ввода списка можно использовать следующую программу:

```
5.6.3 n = 3
      my_list = list()
      for i in range(n):
          elem = [int(i) for i in input().split()]
          my_list.append(elem)
      print(my_list)
```

---

IN:	<code>[[0, 9, 8, 7], [6, 5], [4, 3, 2, 1, 0]]</code>
-----	--

---

OUT:	<code>0 9 8 7</code>
	<code>6 5</code>
	<code>4 3 2 1 0</code>

---

В примере 5.6.3 для элементов списка использована функция `int()` для конвертации строки в число, так как метод `split()` возвращает список строк, а не чисел.

Для перебора элементов вложенного списка следует использовать несколько циклов. В примере цикл `for` по переменной `i` проходит по каждому внутреннему списку, а цикл `for` по переменной `j` проходит по каждому элементу внутри каждого внутреннего списка:

---

5.6.4	<pre>my_list = [[0, 1, 2], [3, 4, 5, 6], [7, 8, 9]] for i in range(len(my_list)):     for j in range(len(my_list[i])):         print(my_list[i][j], end=' ')     print() #перевод строки</pre>
-------	--

---

OUT:	<code>0 1 2</code>
	<code>3 4 5 6</code>
	<code>7 8 9</code>

---

## 5.7. Сортировка списка

Сортировка – это процесс упорядочивания элементов в последовательности по заданному критерию (по возрастанию или по убыванию). Многие алгоритмы, работающие с данными, требуют, чтобы данные были предварительно отсортированы (например, поиск элемента в отсортированном списке значительно быстрее, чем в неотсортированном). Также отсортированные данные легче анализировать и визуализировать.

В Python существует два способа сортировки. Первый из них сортировка “на месте”. т.е. изменяется исходный список.

---

5.7.1	<pre>my_list = [7,2,8,1,0,5,3,7] my_list.sort() print(*my_list)</pre>
-------	---

---

OUT:	<code>0 1 2 3 5 7 7 8</code>
------	------------------------------

---

В примере 5.7.1 используется метод `sort()`, применяемый к списку, который упорядочивает список `my_list`. Такой метод определен только для объектов типа список, его нельзя применить к кортежу или строке. Если вы хотите отсортировать по не возрастанию (по убыванию), то нужно в метод `sort()` добавить аргумент `reverse=True`.

```
5.7.2 my_list = [7,2,8,1,0,5,3,7]
      my_list.sort(reverse=True)
      print(*my_list)
```

```
OUT: 8 7 7 5 3 2 1 0
```

Второй способ состоит в применении функции `sorted()`, которая возвращает отсортированный список, но не изменяет значение своего параметра:

```
5.7.2 my_list = [7,2,8,1,0,5,3,7]
      sorted_list = sorted(my_list)
      print(*sorted_list)
```

```
OUT: 0 1 2 3 5 7 7 8
```

Использование функции `sorted()` оправдано в случае, если исходные данные нужно сохранить в неизменном виде с какой-то целью. Данную функцию можно применять не только для списков, а также для кортежей и строк.

## 5.8. Кортежи

Помимо списков в Python существует неизменяемый тип данных – кортеж. Он также служит для хранения упорядоченной последовательности элементов, которые могут быть произвольных типов, но их нельзя менять.

Синтаксис создания кортежа:

```
имя_кортежа = (элемент_1, элемент_2, ...)
```

Также для создания кортежей может быть использована функция `tuple()`. Если вызвать эту функцию без аргументов, будет создан пустой кортеж. Если передать текст в качестве аргумента функции `tuple()`, результатом будет кортеж, элементами которого являются буквы из текста (каждый элемент кортежа представлен как текст). Чтобы создать кортеж на основе списка или другого существующего кортежа, необходимо указать этот список или кортеж в качестве аргумента функции `tuple()`.

Если кортеж является единственным выражением слева или справа от оператора присваивания, скобки можно опустить. Однако во всех остальных случаях скобки опускать нельзя, так как это может привести к ошибкам.

Ниже представлен пример с несколькими способами создания кортежей:

```
5.8.1 astronauts = ('Гагарин', 'Леонов', 'Терешкова')
      planets = 'Марс', 'Венера', 'Земля'
      favorite_films = ()
      letters = tuple("Hello")      #('H', 'e', 'l', 'l', 'o')
      numbers = tuple([5, 2, 9])
```

	<code>print(type(astronauts))</code>	<code>#вывод типа данных astronauts</code>
	<code>print(type(planets))</code>	<code>#вывод типа данных planets</code>
	<code>print(type(favorite_films))</code>	<code>#вывод типа данных favorite_films</code>
	<code>print(type(letters))</code>	<code>#вывод типа данных letters</code>
OUT:	<code>&lt;class 'tuple'&gt;</code> <code>&lt;class 'tuple'&gt;</code> <code>&lt;class 'tuple'&gt;</code> <code>&lt;class 'tuple'&gt;</code>	

Утверждение о том что кортежи относятся к неизменяемым типам данных не означает, что с ними нельзя выполнять операции.

Пример создания нового кортежа на основе суммы двух других:

5.8.2	<code>letters = tuple("Hello")</code> <code>numbers = (3, 5, 6)</code>  <code>symbols = letters + numbers</code>  <code>print(symbols)</code>	
OUT:	<code>('H', 'e', 'l', 'l', 'o', 3, 5, 6)</code>	

При сложении создаётся новый кортеж, содержащий элементы первого и второго кортежей (как при сложении строк). Кортеж также можно умножить на число, что аналогично умножению строки на число.

5.8.3	<code>numbers = (3, 5, 6)</code> <code>symbols = numbers * 2</code>  <code>print(symbols)</code>	
OUT:	<code>(3, 5, 6, 3, 5, 6)</code>	

Для того чтобы узнать размер кортежа используется функция `len()`. Как и к строкам, операцию среза можно применять и к кортежам.

Кортежи в Python используют для хранения данных, которые должны быть защищены от случайных или намеренных изменений, для передачи информации между функциями или для возврата результатов, для сравнения с использованием операторов, индексации, итерации, форматирования строк и в других случаях.

## Контрольные вопросы

1. В чём разница между списками и кортежами в Python?
2. Как можно удалить элемент из списка?
3. Как отсортировать список в порядке невозрастания?
4. Что будет выведено на экран?



```
l = [1, 2, '6', 3, 'стол']
```

```
if '2' in l:  
    print(1, end = '')
```

```
if 2 in l:  
    print(2, end = '')
```

```
if [2] in l:  
    print(3, end = '')
```

```
if (2) in l:  
    print(4, end = '')
```

Правильный ответ: 24

## 6. Словари и множества

### 6.1. Словари

Словарь (dict) – это структура данных, в которой каждый элемент вместо индекса имеет уникальный ключ. Они используются для организации и быстрого доступа к данным, а также для реализации таких функций, как подсчет элементов, экономия памяти и установка соответствий между объектами.

Для создания словаря используются фигурные скобки {} или функция dict().

Синтаксис создания словаря выглядит следующим образом:

```
имя_словаря = {ключ_1: значение_1, ключ_2: значение_2, ...}
```

Еще несколько примеров создания словаря:

```
students = dict()           #создание пустого словаря
favorite = dict(
    product = 'Ananas',
    movie = 'Dune 2',
    book = 'Crime and Punishment'
)
```

Обратим внимание, что в строке `students = {}` будет создан не пустой словарь, а множество (см. следующий раздел).

Пример словаря, в котором ключами являются имена \

хозяев питомцев, а значениями – клички животных:

```
6.1.1  pets = { 'Маша': 'Том',
                'Гриша': 'Кеша',
                'Данил': 'Муська' }
```

Добавить в словарь пару ключ-значение достаточно, необходимо сделать следующее:

```
6.1.2  pets = { 'Маша': 'Том',
                'Гриша': 'Кеша',
                'Данил': 'Муська' }

pets[ 'Кирилл' ] = 'Себастьян'      #добавление
                                    #ключа - "Кирилл"
                                    #значения - "Себастьян"
```

Обновление значение в словаре происходит точно также:

```
6.1.3  pets = { 'Маша': 'Том',
               'Гриша': 'Кеша',
               'Данил': 'Муська' }

        pets['Гриша'] = 'Аркаша'      #изменение значения по
                                     #ключу - "Гриша".
                                     #новое значение - "Аркаша"
```

Получить значение можно с помощью ключа:

```
6.1.4  pets = { 'Маша': 'Том',
               'Гриша': 'Кеша',
               'Данил': 'Муська' }

        print(pets['Данил'])
```

OUT: Муська

Но если ключа не будет в словаре, то возникнет ошибка:

```
6.1.5  pets = { 'Маша': 'Том',
               'Гриша': 'Кеша',
               'Данил': 'Муська' }

        print(pets['Мирон'])
```

OUT: Traceback (most recent call last):  
 File "main.py", line 5, in <module>  
 print(pets['Мирон'])  
 KeyError: 'Мирон'

Чтобы проверить существование ключа используют операцию `key in dict`. Например:

```
6.1.6  pets = { 'Маша': 'Том',
               'Гриша': 'Кеша',
               'Данил': 'Муська' }

        print('Маша' in pets)
        print('Юля' in pets)
```

OUT: True  
 False

Используя цикл `for`, можно вывести на экран только ключи словаря:

```
6.1.6  pets = {'Маша': 'Том',
              'Гриша': 'Кеша',
              'Данил': 'Муська'}

        for i in pets:
            print(i)
```

```
OUT:    Маша
        Гриша
        Данил
```

Чтобы вывести все значения:

```
6.1.7  pets = {'Маша': 'Том',
              'Гриша': 'Кеша',
              'Данил': 'Муська'}

        for i in pets:
            print(owners[i])
```

```
OUT:    Том
        Кеша
        Муська
```

Удаление элемента из словаря осуществляется с помощью специальной команды `del`. Эта команда не является функцией; после слова `del` следует пробел, затем указывается имя словаря, а затем, в квадратных скобках, ключ, который нужно удалить.

```
6.1.8  pets = {'Маша': 'Том',
              'Гриша': 'Кеша',
              'Данил': 'Муська'}

        del pets['Гриша']
```

## 6.2. Множества

Множество (`set`) в Python – это изменяемый набор уникальных и неупорядоченных элементов, которые могут быть разного типа данных.

Существует несколько способов создания множества. Первый – с помощью встроенной функции `set()`. Второй – с помощью фигурных скобок `{}`. Если при создании присутствовало несколько одинаковых элементов, то они попадут в множество в единственном экземпляре.

Несколько примеров создания множеств:

```
students = {}                                # {}
```

```

numbers = {1, 2, 3, 4}          # {2, 4, 1, 3}
numbers_2 = {2, 2, 2, 2, 2}    # {2}      !!только уникальные!!
letters = set('abcdddd')       # {'a', 'c', 'b', 'd'}

```

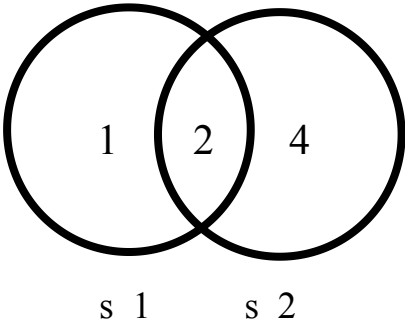
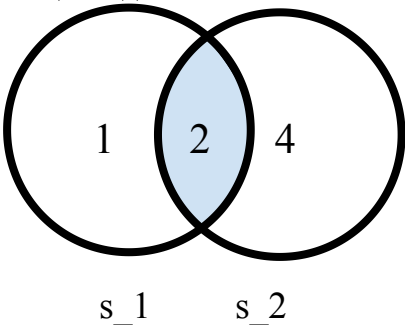
т.к. множество неупорядоченный тип данных порядок элементов может быть другой.

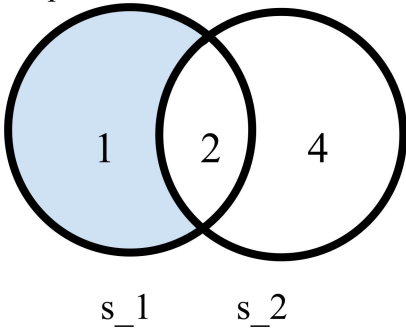
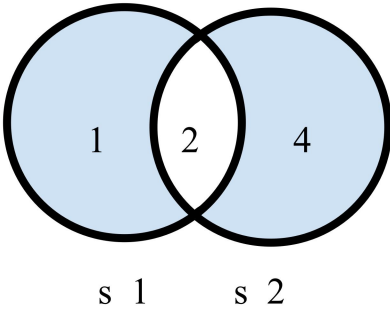
Существует тип данных `frozenset` – это неизменяемый аналог множества.

Множества в Python используются для хранения элементов в единственном экземпляре и удаления дубликатов.

Таблица 6.2.1 – Методы работы со множествами

Метод	Описание	Пример
<code>add()</code>	Добавление элемента. Если вы по какой-то причине захотите добавить элемент, который уже присутствует во множестве, то никакой ошибки не появится, но так как множество содержит только уникальные элементы, то повторный элемент во множество занесен не будет.	<pre>s = {1, 2, 5, 3} s.add("6")</pre>
<code>remove()</code>	Удаление элемента. Если вы попытаетесь удалить элемент, которого не будет во множестве, то появится ошибка <code>KeyError</code> .	<pre>s = {1, 2, 5, 3} s.remove(2)</pre>
<code>discard()</code>	Метод используется для удаления элемента из множества, если он присутствует в множестве. Если элемент отсутствует в множестве, то исключение не возникает и исходное множество остаётся неизменным.	<pre>s = {1, 2, 5, 3} s.discard(5)</pre>
<code>pop()</code>	Метод используется для удаления случайного элемента из множества и возврата его значения. Если множество пустое, возникает ошибка. Так как множества не	<pre>s = {1, 2, 3} t = s.pop()  #s = {2, 3} #t = 1</pre>

	упорядочены, нельзя точно сказать, какой элемент будет первым.	
<code>clear()</code>	Очистка множества	<pre>s = {1, 2, 3} s.clear()</pre>
<code>union()</code>	<p>Метод возвращает новое множество со всеми отдельными элементами из двух множеств.</p>  <p style="text-align: center;">s_1      s_2</p>	<pre>s_1 = {1, 2} s_2 = {2, 4} s_3 = s_1.union(s_2)  #s_3 = {1, 2, 4}</pre>
<code>update()</code>	Метод обновляет содержимое множества, добавляя в него элементы из другого множества, списка, кортежа или словаря.	<pre>s_1 = {1, 2} s_2 = {2, 4} s_1.update(s_2)  #s_1 = {1, 2, 4}</pre>
<code>intersection()</code>	<p>Метод возвращает новое множество с элементами, общими для всех множеств.</p>  <p style="text-align: center;">s_1      s_2</p>	<pre>s_1 = {1, 2} s_2 = {2, 4} s_3=s_1.intersection(s_2)  #s_3 = {2}</pre>
<code>intersection_update()</code>	Метод используется для модификации исходного множества, сохраняя только общие элементы с другими заданными множествами. Он обновляет исходное множество на месте и удаляет элементы, которые не	<pre>s_1 = {1, 2} s_2 = {2, 4} s_1.intersection_update(s_2)  #s_1 = {2}</pre>

	присутствуют в другом.	
<code>difference()</code>	<p>Метод возвращает новое множество с элементами, которые присутствуют в первом, но отсутствуют во втором.</p>  <p style="text-align: center;">s_1      s_2</p>	<pre>s_1 = {1, 2} s_2 = {2, 4} s_3 = s_1.difference(s_2)  #s_3 = {1}</pre>
<code>difference_update()</code>	<p>Метод используется для обновления множества путём удаления элементов, которые находятся в другом множестве или итерируемом объекте. Этот метод изменяет текущее множество на месте и не создает новый объект.</p>	<pre>s_1 = {1, 2} s_2 = {2, 4} s_1.difference_update(s_2)  #s_1 = {1}</pre>
<code>symmetric_difference()</code>	<p>Метод возвращает симметричную разность двух множеств. Симметричная разность – это множество элементов, которые находятся либо в первом множестве, либо во втором, но не пересекаются.</p>  <p style="text-align: center;">s 1      s 2</p>	<pre>s_1 = {1, 2} s_2 = {2, 4} s_3 = symmetric_difference(s_2)  #s_3 = {1, 4}</pre>
<code>symmetric_difference_update()</code>	<p>Метод обновляет множество путем удаления общих для двух множеств элементов.</p>	<pre>s_1 = {1, 2} s_2 = {2, 4} s_1.symmetric_difference_update( s_2)</pre>

		#s_1 = {1, 4}
copy()	Метод создаёт копию текущего множества. Копия множества содержит те же элементы, что и исходное множество, но является отдельным объектом.	s_1 = {1, 2} s_2 = s_1.copy()

Таблица 6.2.2 – Операторы работы со множествами

Оператор	Описание	Пример
	Аналогичен методу union()	s_1 = {1, 2} s_2 = {2, 4} s_3 = s_1   s_2  #s_3 = {1, 2, 4}
&	Аналогичен методу intersection()	s_1 = {1, 2} s_2 = {2, 4} s_3 = s_1 & s_2  #s_3 = {2}
-	Аналогичен методу difference()	s_1 = {1, 2} s_2 = {2, 4} s_3 = s_1 - s_2  #s_3 = {1}
^	Аналогичен методу symmetric_difference()	s_1 = {1, 2} s_2 = {2, 4} s_3 = s_1 ^ s_2  #s_3 = {1, 4}

Таблица 6.2.3 – Сравнение множеств

Оператор	Описание	Пример
==	Проверка равенства двух множеств	s_1 = {1, 2, 3, 4} s_2 = {1, 4, 2, 3}  print(s_1 == s_2) #True
!=	Проверка не равенства двух множеств	s_1 = {1, 2, 3, 4} s_2 = {1, 2, 3, 5}



		<code>print(s_1 != s_2) #True</code>
<code>&lt;</code>	Проверка является ли левое множество подмножеством правого	<code>s_1 = {1, 2}</code> <code>s_2 = {1, 2, 5, 6}</code> <code>s_3 = {1, 2}</code>  <code>print(s_1 &lt; s_2) #True</code> <code>print(s_1 &lt; s_3) #False</code>
<code>&lt;=</code>	Проверка, является ли множество s_1 подмножеством s_2	<code>s_1 = {2, 4}</code> <code>s_2 = {1, 2, 3, 4}</code> <code>s_3 = {2, 4}</code>  <code>print(s_1 &lt;= s_2) #True</code> <code>print(s_1 &lt;= s_3) #True</code>
<code>&gt;</code>	Проверка является ли правое множество подмножеством левого	<code>s_1 = {1, 2, 3, 4}</code> <code>s_2 = {1, 2}</code> <code>s_3 = {1, 2}</code>  <code>print(s_1 &gt; s_2) #True</code> <code>print(s_2 &gt; s_3) #False</code>
<code>&gt;=</code>	Проверка, является ли множество s_2 подмножеством s_1	<code>s_1 = {1, 2, 3, 4}</code> <code>s_2 = {2, 4}</code> <code>s_3 = {1, 2}</code>  <code>print(s_1 &gt;= s_2) #True</code> <code>print(s_2 &gt;= s_3) #True</code>

## Контрольные вопросы

1. Как проверить существование ключа в словаре Python?
2. Как найти общие элементы в трех множествах?
3. Можно ли удалить в словарь добавить две пары элементов, в которых одинаковые ключи?
4. Что будет выведено на экран?

```
puts = {0: 1,
        1: 1,
        2: 3,
        3: 4
}
puts[1] = 2

for i in puts:
    print('*' * puts[i])
```

Правильный ответ:

\*  
\* \*  
\* \* \*  
\* \* \* \*

## 7. Функции

### 7.1. Понятие функции

Функции в Python — это блоки кода, которые выполняют определённые действия и могут быть вызваны из других частей программы. Они позволяют структурировать код, делать его более читаемым, повторно использовать фрагменты кода и упрощать разработку программ.

В разделах ранее уже использовались готовые функции, такие как `len()`, `sum()`, `sorted()`, `type()`, `input()` и др. Эти функции описаны в стандартной библиотеке или других подключаемых библиотеках.

Давайте рассмотрим способы создания своих функций. Функции в Python создаются с помощью ключевого слова `def`. Это действие создает объект функции и присваивает ему имя, которое становится ссылкой на объект-функцию.

Пример определения функции:

```
7.1.1  def print_stars():  
        print('*****')
```

`print_stars` — это имя функции. Тело функции начинается с новой строки с отступом, который указывает на вхождение в блок кода функции. В теле функции может присутствовать инструкция `return` (может быть несколько), которая прерывает выполнение функции и возвращает значение в основную ветку программы. `return` может возвращать не только одно значение, а несколько (например, `return a, b`).

Рассмотрим пример 7.1.2:

```
7.1.2  1  def calculate_pi():  
        2      pi = 0  
        3      for i in range(10000):  
        4          pi += 4 * (-1)**i / (2 * i + 1)  
        5      return pi  
        6  
        7  a = calculate_pi()  
        8  print(f"Приближенное значение числа Пи: {a}")
```

Этот код вычисляет приближенное значение числа Пи с помощью формулы Лейбница. Реализация данной формулы находится в функции `calculate_pi()`. Вызов данной функции находится на 7 строке, именно с нее начинается программа. Этапы работы программы продемонстрированы на рисунке:

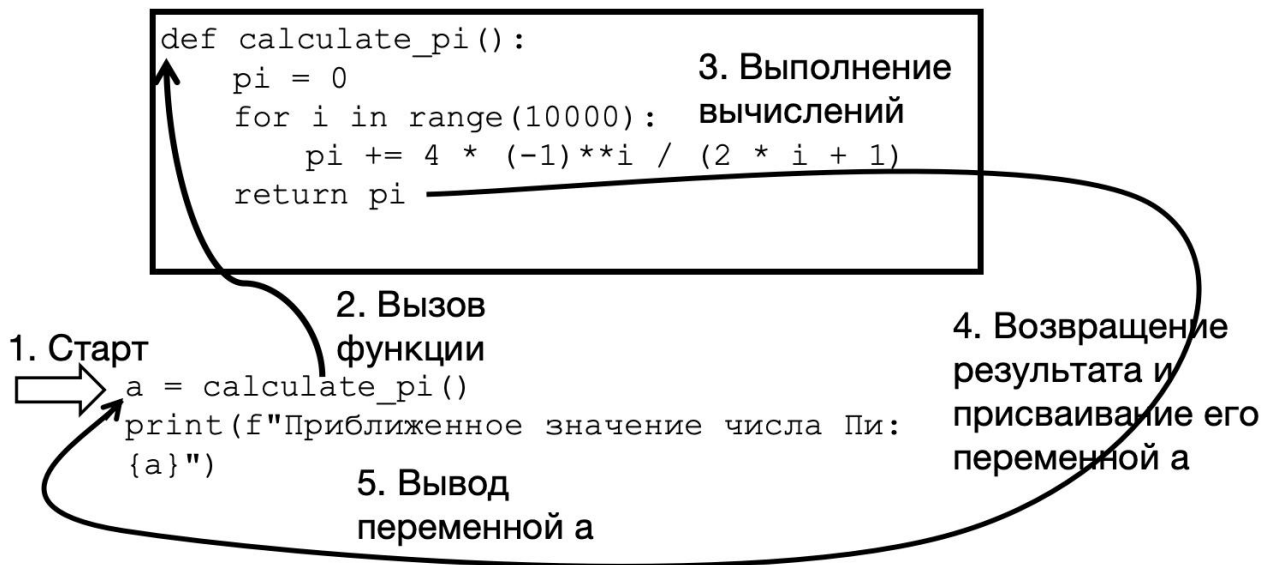


Рисунок 7.1.1 – Этапы выполнения программы

Действия внутри функции точно такие же, как в обычной программе, кроме дополнительной команды `return`. Команда `return` возвращает значение функции (оно должно быть записано через пробел после слова `return`) и прекращает её работу. Возвращенное значение подставляется на то место, где осуществлялся вызов функции.

Функция выполняет код внутри себя так же, как обычная программа `return` завершает работу функции и передает возвращаемое значение в то место, где функция была вызвана.

Команда `return` может встречаться в любом месте функции. После того как она выполнится, работа функции будет прекращена. Здесь есть некоторая аналогия с командой `break`, применяемой для выхода из цикла.

## Вызов функции и возврат значения

Пример вызова функции:

```
7.1.3 a = calculate_pi()
```

Если в функции отсутствует команда `return`, то при вызове функции в основную программу ничего не возвращается. В этом случае переменной `a` не присваивается никакое значение.

Важно знать, что если функция не возвращает значение явно с помощью `return`, то по умолчанию она возвращает специальное значение `None`.

## Передача параметров в функцию

Функции могут становиться более гибкими, принимая аргументы, которые передаются в круглых скобках при вызове функции. Эти аргументы используются для инициализации переменных параметров, заданных в круглых скобках при определении функции.

Количество переменных параметров, указанных при определении функции, должно соответствовать количеству аргументов, переданных при вызове. Если у функции нет параметров, то при ее вызове все равно необходимо использовать пустые скобки. Ниже в примере 7.1.4 и рисунке 7.1.2 показана программа, которая выводит приветствие с именем.

```
7.1.4  def hello(name):  
        print("Hello", name)  
  
        n = "Danil"  
        hello(n)  
OUT:   Hello Danil
```

---

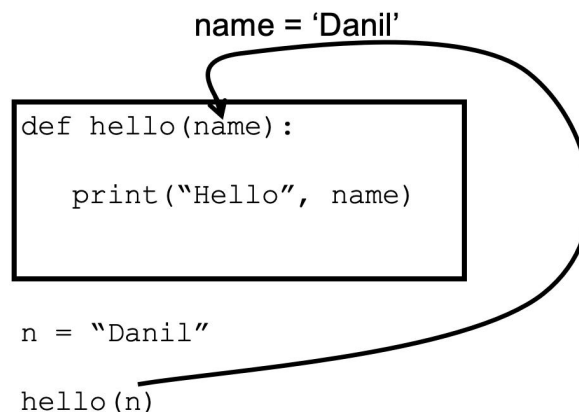


Рисунок 7.1.2 – Передача параметра в функцию

Объяснение того зачем передавать переменную `n` в функцию `hello`, а не использовать ее напрямую, написаны в следующем параграфе.

## 7.2. Глобальные и локальные переменные

Существует понятие область видимости переменной. Это часть программы, где переменная доступна для использования. В Python, область видимости переменных определяется тем, где они объявлены. В данном примере 7.1.3 две зоны - глобальная и локальная.

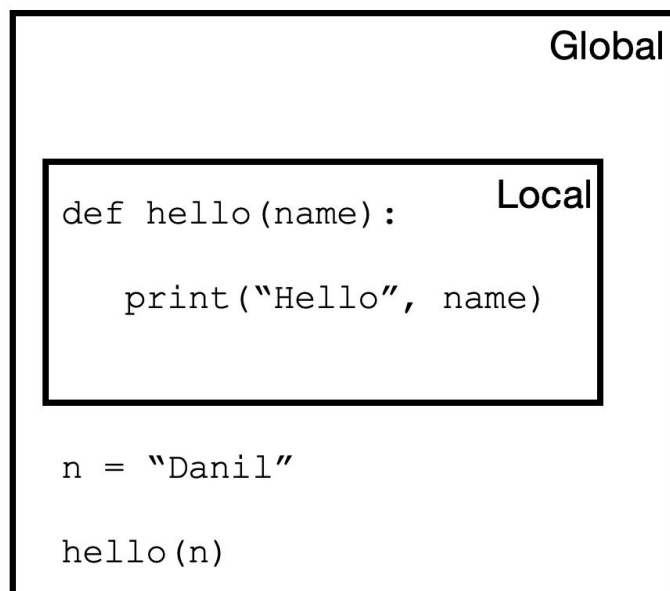


Рисунок 7.2.1 – Зоны видимости переменных

Локальные переменные объявляются внутри функции, доступны только внутри той функции, где они объявлены и не доступны извне.

Глобальные переменные объявляются вне функций, доступны из любой части программы. Если необходимо изменить значение глобальной переменной внутри функции, необходимо использовать ключевое слово `global`.

Если мы изменим программу следующим образом (не станем туда отправлять строку с именем), то она отработает все равно правильно:

```
7.2.1 def hello():
        print("Hello", n)

n = "Danil"
hello()
```

Это происходит потому, что переменная `n` - глобальная. На момент вызова функции `hello()` переменная `n` уже создана, хотя описание функции и идет раньше присваивания.

Тогда может возникнуть вопрос: зачем отправлять глобальные переменные в функции, если они там видны? Они не доступны для изменения. Это сделано для того, чтобы избежать непреднамеренных изменений глобальных переменных, которые могут привести к ошибкам и непредсказуемому поведению программы. Давайте попробуем изменить значение переменной `n` в функции.

```
7.2.3 def hello():
      n = "Dasha"
```

```

        print("In function: ", n)

    n = "Danil"
    hello()
    print("In global:", n)
OUT:      In function:  Dasha
        In global:  Danil

```

Значение `n` поменялось, но только в функции, так как она стала локальной переменной для данной функции, а глобальная `n` осталась без изменений.

И рассмотрим вариант изменение значения глобальной переменной в функции:

```

7.2.4    def hello():
            global n
            n = "Dasha"
            print("In function: ", n)

    n = "Danil"
    hello()
    print("In global:", n)
OUT:      In function:  Dasha
        In global:  Dasha

```

## Контрольные вопросы

1. Какая команда используется для прерывания функции?
2. Что такое локальные и глобальные переменные в Python?
3. Ограничена ли количество параметров, отправляемых в функцию?

## 8. Обработка исключений

### 8.1. Исключения, конструкция try — except, finally, оператор raise

Исключения (Exceptions) — это ошибки, которые возникают во время выполнения программы. Они сигнализируют о том, что что-то пошло не так и программа не может продолжить работу в обычном режиме.

Рассмотрим пример демонстрации исключения – деление строки на число.

```
8.1.1  print("Hello"/5)

OUT:    Traceback (most recent call last):
        File "main.py", line 1, in <module>
          print("Hello"/5)
        TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

В данном примере интерпретатор сообщает о том, что он поймал исключение и напечатал информацию (Traceback (most recent call last)).

Далее имя файла (File "main.py") и строка в файле (line 1);  
Выражение, в котором произошла ошибка (print("Hello"/5)).

Название исключения (TypeError) и краткое описание исключения (unsupported operand type(s) for /: 'str' and 'int').

Далее приведен список исключений, которые есть в Python. Но возможно создавать и свои.

Таблица 8.1.1 – Исключения

Исключение	Описание
ArithmeticError	Арифметическая ошибка
BufferError	Операция, связанная с буфером, не может быть выполнена
EOFError	Обнаружение маркера конца файла (EOF - end of file) в ходе считывания данных
FileExistsError	Создание файла или директории, которая уже существует
FileNotFoundError	Файл или директория не существует
FloatingPointError	Неудачное выполнение операции с плавающей запятой
ImportError	Не удалось импортировать модуль или его атрибута
IndentationError	Неправильные отступы
IndexError	Индекс не входит в диапазон элементов
IsADirectoryError	Ожидался файл, но это директория
KeyboardInterrupt	Порождается при прерывании программы пользователем (обычно сочетание клавиш Ctrl+C)
KeyError	Несуществующий ключ (в словаре, множестве или др.)
MemoryError	Недостаточно памяти



NameError	Не найдено переменной с таким именем
NotADirectoryError	Ожидалась директория, но это файл
OverflowError	Результат арифметической операции слишком велик для представления, не появляется при обычной работе с целыми числами
PermissionError	Не хватает прав доступа
RuntimeError	Исключение не попадает ни под одну из других категорий
SyntaxError	Синтаксическая ошибка
SystemError	внутренняя ошибка
TabError	Смешивание в отступах табуляции и пробелов
TimeoutError	Закончилось время ожидания
TypeError	Операция применена к объекту несоответствующего типа
UnicodeError	Ошибка, связанная с кодированием / декодированием unicode в строках
ValueError	Функция получает аргумент правильного типа, но некорректного значения
Warning	Предупреждение

В Python мы можем управлять ошибками, которые могут возникнуть во время выполнения программы. Для этого используется конструкция `try...except`. Блок `try` содержит код, который может вызвать ошибку. Если в этом блоке возникает ошибка, то выполнение кода в `try` прекращается, и управление передается блоку `except`. В блоке `except` указывается какую ошибку нужно перехватить и какие действия следует выполнить, если эта ошибка произошла. Таким образом, мы можем обеспечить более устойчивое выполнение программы, обработав возникшие ошибки и предотвратив непредвиденное поведение. Рассмотрим в 8.1.2 использование блока `try...except` на примере деления числа на ноль.

```
8.1.2    try:
          num = 2024 / 0
          except ZeroDivisionError:
              num = 'Деление на ноль'
          print(num)
```

OUT: Деление на ноль

Если не указывать конкретный тип ошибки после `except`, то будут перехвачены все возможные ошибки, включая прерывание выполнения программы пользователем или системные ошибки. Однако такой подход не рекомендуется, поскольку он может скрыть важные ошибки в программе. В место этого рекомендуется использовать `except Exception`, чтобы перехватить все исключения, или перехватывать конкретные типы исключений для более точной обработки ошибок.

При обработке исключений так же используются два ключевых элемента: блоки `finally` и `else`. Блок `finally` гарантирует выполнение определенных инструкций вне зависимости от возникновения исключения. Это особенно полезно для действий, которые необходимо выполнить в любом случае, например, закрытие открытых файлов или освобождение ресурсов. Блок `else` выполняется только тогда, когда исключение не было вызвано. Он позволяет выполнить определенные действия, которые требуются только в случае успешного завершения кода без ошибок.

```
8.1.3  string = '1 3 a 7 9'
       ints = []
       try:
           for symb in string.split():
               ints.append(int(symb))
       except ValueError:
           print('Это не число. Выходим.')
       except Exception:
           print('Что-то непонятное')
       else:
           print('Всё хорошо.')
       finally:
           print(ints)
           print('Финал')
```

```
OUT:   Это не число. Выходим.
       [1, 3]
       Финал
```

В примере 8.1.3 элементы строки `string` добавляются в список `ints`. В выводе видно, что после того, как сработало исключение программа завершает выполнение. Если при добавлении видим не число – то срабатывает исключение `ValueError`, в других случаях срабатывает общее исключение `Exception`. В случае если исключения не возникнут – выведется “Всё хорошо”. После выполненных блоков сработает `finally`. Порядок: `try`, группа `except`, затем `else`, и только потом `finally`.

Оператор `raise` применяется когда возникает необходимость вручную сгенерировать исключение. Пример:

```
8.1.4  raise Exception("текст исключения")
```

И для обработки созданного исключения необходимо выполнить:

```
8.1.5  except Exception as e:
       print(e)
```

Пример обработки некорректно введенных данных посредством исключений:

```

8.1.6  try:
        m = int(input("Введите номер месяца, в котором вы
        родились: "))
        if m < 1 or m > 12:
            raise Exception("Некорректный номер месяца")
        print("Отлично, в базу записано число:", m)
    except ValueError:
        print("Введены некорректные данные")
    except Exception as e:
        print(e)
    finally:
        print("Завершение программы")

```

IN: 10000

OUT: Введите номер месяца, в котором вы родились: 13  
 Некорректный номер месяца  
 Завершение программы

Важно отметить, что с помощью `raise` исключение только создаётся. Чтобы оно работало его необходимо прописать в блоке `except`.

## 8.2. Оператор `assert`

Оператор `assert` используется во время выполнения программы для проверки истинности выражения. Он позволяет задать условие, которое должно быть истинным в указанном месте программы. Если выражение оценивается как ложное, то возбуждается исключение `AssertionError`. Синтаксис оператора `assert`:

```

8.2.1  assert выражение, "Сообщение об ошибке"

```

В данном случае:

выражение – это условие, которое мы проверяем на истинность.

"Сообщение об ошибке" – необязательная часть, которая будет выведена в случае возникновения исключения `AssertionError`.

```

8.2.2  age = 18
        assert age == 18, "Значение age должно быть равно 18"

```

Если значение переменной `age` не равно 18, то будет выброшено исключение `AssertionError` с сообщением "Значение `age` должно быть равно 18".

Оператор `assert` часто используется для проверки предположений о значениях переменных, состоянии объектов или любых других условий, которые должны быть верными в определенный момент времени в программе.

Еще один пример использования оператора `assert`:

```
8.2.3 def sum(ints):
        assert len(ints) != 0, 'Список ints пуст'
        return round(sum(ints))
ints = []
print("Сумма элементов:", sum(ints))
```

```
OUT: Traceback (most recent call last):
      File "main.py", line 5, in <module>
        print("Сумма элементов:", sum(ints))
      File "main.py", line 2, in sum
        assert len(ints) != 0, 'Список ints пуст'
AssertionError: Список ints пуст
```

## Контрольные вопросы

1. Как устроен механизм исключений?
2. Можно ли сгенерировать исключение самостоятельно?
3. В чем отличие `else` от `finally` в обработке исключений?
4. Что будет выведено на экран?

```
try:
    name = input('Введите ваше имя')
    res = 1 + 'пользователь :' + name
    print(res)
except TypeError:
    print('Упс, что-то пошло не так')
```

Правильный ответ: Упс, что-то пошло не так

## 9. Файлы. Работа с файловой системой

### 9.1. Работа с текстовыми файлами формата txt

Обработка текстовых данных – яркий пример задачи, в которой заранее неизвестно, сколько данных нам предстоит считать, а именно, заранее не известно, сколько строк будет введено.

В Python для открытия файла используется функция `open()`, принимающая два аргумента: имя файла и режим открытия ("r" для чтения и "w" для записи). Эта функция возвращает ссылку на объект типа файл. Иногда в функции `open` используют аргумент `encoding` для указания кодировки символов, которая будет использоваться для чтения или записи файла. Файлы могут быть сохранены в различных кодировках, таких как UTF-8, ASCII и т.д. Если кодировка файла не совпадает с кодировкой, используемой вашей программой, то есть вероятность столкнуться с проблемами при чтении или записи данных. После окончания работы с файлами нужно вызвать для них методы `close()`.

Пример открытия файла `input.txt` на чтение:

```
9.1.1 inFile = open('input.txt', 'r', encoding='utf8')
```

Таблица 9.1.1 – Режимы открытия файлов

Аргумент	Режим	Описание
'r'	Чтение	Открытие файла для чтения (по умолчанию).
'w'	Запись	Открытие файла для записи. Если файл уже существует – содержимое будет стерто, если файла не существует – он будет создан
'a'	Добавление	Открытие файла для добавления данных в конец файла. Если файла не существует – он будет создан
'b'	Бинарный режим	Открытие файла в бинарном режиме, что позволяет работать с двоичными данными. Этот режим может быть добавлен к основным режимам открытия файлов, образуя комбинации, такие как 'rb', 'wb', 'ab' и

		так далее.
' + '	Обновление	Чтение и запись данных в файл. Часто применяется в сочетании с другими режимами, например, 'r+' для чтения и обновления существующего файла или 'w+' для записи и обновления нового или существующего файла.

Конструкция `with open ... as ...` предлагает удобный способ работы с файлами, гарантируя их автоматическое закрытие после завершения операций. Внутри блока `with` файл доступен для чтения или записи, а по завершении блока Python автоматически закрывает его, избавляя от необходимости явного вызова метода `file.close()`.

```
9.1.2  with open('input.txt', 'r') as file:
        #Работа с данными
        #Работа с данными
        #В этой точке файл уже закрыт
```

В данном примере мы открываем файл в режиме чтения с помощью `'r'`. Открытый файл будет доступен в переменной `file` в блоке `with`. После завершения блока `with`, файл будет закрыт автоматически, обеспечивая безопасность и правильное завершение работы с файлом.

После открытия файла следует считать содержимого файла и сохранения его в переменной для дальнейшей обработки.

Таблица 9.1.2 – Методы чтения

Метод чтения	Описание
<code>read()</code>	Чтение содержимого файла и сохранение его в переменной
<code>read(size)</code>	Считывание из файла указанное количество символов
<code>readline()</code>	Построчное чтение
<code>readlines()</code>	Чтение всех строк в виде списка строк

Пример построчного чтения и вывода файла на экран:

```
9.1.3  with open("file.txt", "r") as file:
        for line in file:
            print(line)
```

Запустив эту программу, вы заметите необычное поведение: после каждой выведенной строки будет следовать пустая строка. Это объясняется тем, что в текстовом файле каждая строка заканчивается одним или несколькими символами конца строки, которые воспринимаются программой как отдельный символ, вызывая вывод пустой строки.

Для того, чтобы программы могли правильно обрабатывать текст, разделяя его на отдельные строки, в конце каждой строки используется специальный символ — маркер конца строки. Без него, вся информация в файле воспринималась бы как единый непрерывный текст, а в текстовом редакторе все строки слились бы в одну.

После того, как строка прочитана из файла, маркер конца строки можно удалить с помощью метода `rstrip()`. Этот метод, который работает с любыми строками, удаляет все пробельные символы, включая пробелы, табуляции и маркеры конца строки, с правого края строки. Метод возвращает новую строку, где эти символы удалены. Модифицированная версия программы:

```
9.1.4   with open("file.txt", "r") as file:
        for line in file:
            print(line.rstrip())
```

Метод `read()` позволяет считать все содержимое файла в одну строковую переменную (при этом содержащую в себе переводы строки `\n`).

Пример чтения и вывода всего файла на экран:

```
9.1.5   with open("my_file.txt", "r") as file:
        content = file.read()
        print(content)
```

Чтобы добавить данные в файл, предварительно открытый для записи или добавления, можно воспользоваться методом `write`. Этот метод принимает в качестве входного параметра строку, которую необходимо записать в файл.

Если требуется записать данные других типов, их можно предварительно преобразовать в строковый формат с помощью функции `str`. Для записи нескольких значений в файл можно предварительно объединить их в одну длинную строку или вызывать метод `write` несколько раз.

В отличие от функции `print`, метод `write` не добавляет автоматически символ перевода строки (`\n`) при записи в файл. Поэтому разработчику необходимо самостоятельно позаботиться о том, чтобы значения, которые должны находиться на разных строках в файле, были явно разделены символами конца строки.

Пример записи в файл:

```
9.1.6    text = "Hello world"

        with open('output.txt', 'w') as file:
            file.write(text)
```

## 9.2. Работа с файлами формата csv

CSV (Comma-Separated Values), известный как формат представления табличных данных, широко применяется для хранения информации, полученной из таблиц или баз данных.

В CSV каждая строка файла соответствует строке таблицы. Хотя название формата указывает на запятую как разделитель, на практике могут использоваться и другие символы, например, табуляция.

Несмотря на то, что форматы с другими разделителями могут иметь свои собственные названия, например, TSV (Tab Separated Values), под термином “CSV” обычно подразумевают все варианты с разделителями, не ограничиваясь только запятой.

В качестве примера представлен файл в формате CSV с названием `sw_data.csv`:

	A	B	C	D
1	hostname,vendor,model,location			
2	sw1,Cisco,3750,London			
3	sw2,Cisco,3850,Liverpool			
4	sw3,Cisco,3650,Liverpool			
5	sw4,Cisco,3650,London			
6				

Рисунок 9.1.1 – Файл, иллюстрирующий формат CSV

Для удобной работы с файлами в формате CSV в Python предусмотрена стандартная библиотека, включающая модуль `csv`. Чтобы использовать его возможности, необходимо подключить этот модуль в свой код с помощью инструкции `import csv`.

### Чтение

Рассмотрим пример чтения файла `films.csv`, содержащего информацию о мультфильмах студии “Союзмультфильм”, с использованием функции `open()` и метода `csv.reader()`.



```
9.2.1 import csv
      with open('films.csv') as f:
          reader = csv.reader(f)
          for row in reader:
              print(row)
```

```
OUT:  ['id', 'Название', 'Год', 'Жанр']
      ['1', 'Колобок', '1936', 'Сказка']
      ['2', 'Заяц-портной', '1937', 'Сказка']
      ['3', 'Мойдодыр', '1939', 'Сказка']
      ['4', 'Конёк-Горбунок', '1947', 'Фэнтези']
      ['5', 'Каштанка', '1950', 'Повесть']
```

Первый список в результате обработки `csv.reader()` содержит названия столбцов, а последующие списки – значения, соответствующие этим столбцам. Важно отметить, что `csv.reader()` возвращает итератор, а не список.

```
9.2.2 import csv
      with open('films.csv') as f:
          reader = csv.reader(f)
          print(reader)
```

```
OUT:  <_csv.reader object at 0x7af898bd3e60>
```

Чтобы получить список из данных, полученных от `csv.reader()`, можно использовать функцию `list()`:

```
9.2.3 import csv
      with open('films.csv', 'r', encoding='utf-8')) as f:
          csv_reader = csv.reader(f)
          print(csv_reader)
```

```
OUT:  [['id', 'Название', 'Год', 'Жанр'], ['1', 'Колобок',
      '1936', 'Сказка'], ['2', 'Заяц-портной', '1937',
      'Сказка'], ['3', 'Мойдодыр', '1939', 'Сказка'], ['4',
      'Конёк-Горбунок', '1947', 'Фэнтези'], ['5', 'Каштанка',
      '1950', 'Повесть']]
```

В целях удобства, заголовки столбцов можно получить отдельным объектом. Для этого можно использовать функцию `next()`.

```
9.2.4 import csv

      with open('films.csv', 'r', encoding='utf-8')) as f:
          csv_reader = csv.reader(f)
          column_names = next(csv_reader)
          print('Заголовки: ', column_names)
          for row in csv_reader:
              print(row)
```

---

```
OUT: Заголовки: ['id', 'Название', 'Год', 'Жанр']
      ['1', 'Колобок', '1936', 'Сказка']
      ['2', 'Заяц-портной', '1937', 'Сказка']
      ['3', 'Мойдодыр', '1939', 'Сказка']
      ['4', 'Конёк-Горбунок', '1947', 'Фэнтези']
      ['5', 'Каштанка', '1950', 'Повесть']
```

---

Для более удобной обработки данных из CSV-файла часто требуется получить информацию в виде словарей, где ключи представляют собой названия столбцов, а значения - соответствующие значения из каждой строки. В модуле `csv` для этой цели предусмотрен класс `DictReader`:

```
9.2.4 import csv

      with open('films.csv') as f:
          csv_reader = csv.DictReader(f)
          for row in column_names:
              print(row)
```

---

```
OUT: {'id': '1', 'Название': 'Колобок', 'Год': '1936', 'Жанр':
      'Сказка'}
      {'id': '2', 'Название': 'Заяц-портной', 'Год': '1937',
      'Жанр': 'Сказка'}
      {'id': '3', 'Название': 'Мойдодыр', 'Год': '1939',
      'Жанр': 'Сказка'}
      {'id': '4', 'Название': 'Конёк-Горбунок', 'Год': '1947',
      'Жанр': 'Фэнтези'}
      {'id': '5', 'Название': 'Каштанка', 'Год': '1950',
      'Жанр': 'Повесть'}
```

---

## Запись

Запись данных в формат CSV также осуществляется с помощью модуля `csv`. В следующем примере строки из списка записываются в файл, после чего содержимое файла выводится на экран:

```
9.2.5 import csv

      data = [['id', 'Название', 'Год', 'Жанр'],
              ['11', 'Тараканище', '1963', 'Приключение'],
              ['12', 'Варежка', '1967', 'Драма']]

      with open('films_2.csv', 'w') as f:
          csv_writer= csv.writer(f)
          csv_writer.writerows(data)
      with open('films_2.csv', 'r') as f:
          print(f.read())
```

---

```
OUT: id,Название,Год,Жанр
      11,Тараканище,1963,Приключение
```

---

Иногда в значении ячейки необходимо использовать символ запятой. Чтобы единое значение впоследствии работы с форматом csv не разделилось запятой на несколько значений следует всё значение ячейки записывать в кавычки. В таком случае значение ячейки является целой строкой. Модуль csv обрабатывает запятые как разделители, если они не находятся внутри кавычек.

Чтобы обеспечить запись всех строк в CSV-файл в кавычках, в модуле csv предусмотрена возможность управления этим параметром. Для этого необходимо добавить параметр `quoting` в метод `csv.writer()`:

```
9.2.6 data = [['id', 'Название', 'Год', 'Жанр'],
              ['11', 'Тараканище', '1963', 'Приключение, Сказка']]

with open('films_2.csv', 'w') as f:
    writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    for row in data:
        writer.writerow(row)

with open('films_2.csv') as f:
    print(f.read())
```

```
OUT: "id", "Название", "Год", "Жанр"
      "11", "Тараканище", "1963", "Приключение, Сказка"
```

---

Теперь все значения с кавычками. И поскольку у фильма указано два жанра через запятую, тут они тоже в кавычках, что позволяет принимать запись "Приключение, Сказка" как единую, а не отдельную единицу данных.

В дополнение к методу `writerows()`, модуль csv предоставляет метод `writerow()`, который позволяет записывать в файл данные из любого итерируемого объекта в виде отдельных строк. Например, предыдущий пример можно переписать с использованием `writerow()`.

```
9.2.7 import csv

data = [['id', 'Название', 'Год', 'Жанр'],
        ['11', 'Тараканище', '1963', 'Приключение, Сказка'],
        ['12', 'Варезка', '1967', 'Драма']]

with open('films_2.csv', 'w') as f:
    csv_writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    for row in csv_writer:
        csv_writer.writerow(row)

with open('films_2.csv') as f:
    print(f.read())
```

---

OUT:	"id", "Название", "Год", "Жанр"
	"11", "Тараканище", "1963", "Приключение, Сказка"
	"12", "Варежка", "1967", "Драма"

---

## Разделитель

В случаях, когда в качестве разделителя используется символ, отличный от запятой, например, точка с запятой, модуль csv позволяет задать этот разделитель. Для этого используется параметр `delimiter` в методе `reader()`. Иногда в качестве разделителя используются другие значения. В таком случае должна быть возможность подсказать модулю, какой именно разделитель использовать.

```
9.2.8 import csv
```

```
with open('films_2.csv') as f:
    print(f.read())
with open('films_2.csv') as f:
    reader = csv.reader(f, delimiter=';')
    for row in reader:
        print(row)
```

```
OUT:  "id";"Название";"Год";"Жанр"
      "11";"Тараканище";"1963";"Приключение, Сказка"
```

```
['id', 'Название', 'Год', 'Жанр']
['11', 'Тараканище', '1963', 'Приключение, Сказка']
```

## 9.3. Работа с файлами формата `xlsx`

Для чтения и записи файлов формата `xlsx` (программы Excel) с помощью Python можно использовать дополнительный модуль `Pandas`. Это работает по аналогии с другими форматами. В этом разделе рассмотрим, как это делается с помощью `DataFrame`.

`DataFrame` - двумерный проиндексированный массив (таблица), в котором столбцами являются объекты класса `Series`. `Series` (серия) представляет собой одномерный массив, который визуально напоминает пронумерованный список. Слева в колонке представлены индексы элементов, а справа - сами значения элементов.

### Запись в файл Excel с python

Для хранения информации, предназначенной для записи в файл формата `XLSX` Excel, следует использовать объект `DataFrame`. Встроенная функция `to_excel()` позволяет записать данные из `DataFrame` в файл Excel.

В примере далее создаются данные о героях романа «Властелин колец» Джона Р. Р. Толкина. Для создания импортируется модуль `pandas`. Затем используется словарь для заполнения `DataFrame`:

```

9.3.1 import pandas as pd
      df = pd.DataFrame({'Name':    ['Gandalf',    'Gollum',
      'Galadriel', 'Legolas', 'Arwen'],
      'Gender': ['Male', 'Male', 'Female', 'Male', 'Female'],
      'Hair': ['White', 'Gray', 'Platinum', 'White', 'Black']})

```

Ключи в словаре будут использоваться как названия столбцов, а значения - как строки с данными. Для записи содержимого словаря в файл Excel можно воспользоваться функцией `to_excel()`, передав ей в качестве аргумента путь к файлу. Ключи в словаре — это названия колонок. А значения станут строками с информацией.

```

9.3.2 df.to_excel('./characters.xlsx')

```

А вот и созданный файл Excel:

	A	B	C	D	E
1		Name	Gender	Hair	
2	0	Gandalf	Male	White	
3	1	Gollum	Male	Gray	
4	2	Galadriel	Female	Platinum	
5	3	Legolas	Male	White	
6	4	Arwen	Female	Black	
7					

Рисунок 9.3.1 – Итоговый файл в формате `xlsx`

В данном примере для записи в файл Excel не использовались дополнительные параметры. Это означает, что имя листа по умолчанию будет “Sheet1”. Также в файле может появиться дополнительная колонка с числами, которые представляют собой индексы из исходного DataFrame.

Чтобы изменить название листа в файле Excel, необходимо добавить параметр `sheet_name` при вызове функции `to_excel()`.

```

9.3.3 df.to_excel('./characters.xlsx', sheet_name='Info',
      index=False)

```

Чтобы убрать колонку с индексами из файла Excel, можно добавить параметр `index=False` при вызове метода `to_excel()`. В результате, файл Excel будет выглядеть следующим образом:

	A	B	C	D	E
1	Name	Gender	Hair		
2	Gandalf	Male	White		
3	Gollum	Male	Gray		
4	Galadriel	Female	Platinum		
5	Legolas	Male	White		
6	Arwen	Female	Black		
7					
8					
<div> <span>◀</span> <span>▶</span> <span>Info</span> <span>⊕</span> </div>					

Рисунок 9.3.2 – Файл в формате xlsx с именем листа 'Info'

### Запись нескольких DataFrame в файл Excel

Также возможно записать несколько DataFrame в файл Excel. Для этого нужно указать отдельный лист для каждого объекта. Пример: создание файла с зарплатами трёх групп работников (все совпадения имён и численных показателей случайны и не имеют общего с реальностью):

```
9.3.4 import pandas as pd

# Создание DataFrame для каждой группы футболистов

players1 = pd.DataFrame({'Name': ['Роналдиньо', 'Криштиану Роналду', 'Ян Облак'], 'Salary': [6500, 3300, 12500]})
players2 = pd.DataFrame({'Name': ['Кевин Де Брюйне', 'Неймар Жуниор', 'Роберт Левандовски'], 'Salary': [2500, 1300, 1900]})
players3 = pd.DataFrame({'Name': ['Алиссон', 'Марк-Андре тер Штеген', 'Мохаммед Салах'], 'Salary': [1500, 2400, 2600]})

# Создание словаря, содержащего DataFrame для каждой группы
salary_sheets = {'Группа 1': players1, 'Группа 2': players2, 'Группа 3': players3}

# Создание объекта ExcelWriter для записи в файл Excel
writer = pd.ExcelWriter('./ players.xlsx',
engine='xlsxwriter')

# Запись каждого DataFrame в отдельный лист в Excel
for sheet_name in salary_sheets.keys():
```

```
salary_sheets[sheet_name].to_excel(writer,
sheet_name=sheet_name, index=False)

# Заккрытие объекта ExcelWriter
writer.close()
```

OUT:

В случае возникновения ошибки 'No module named 'xlsxwriter' дополнительно стоит установить данный модуль. Пример установки:

```
9.3.5 python3 -m pip install xlsxwriter
```

В этом примере создаются три отдельных DataFrame с уникальными названиями, содержащие информацию об именах сотрудников и их заработной плате. Каждый DataFrame заполняется соответствующими данными из словаря.

Объединим эти DataFrame в едином словаре salary\_sheets, где ключи будут представлять имена листов в Excel, а значения - соответствующие DataFrame.

Наконец, сохранение сгенерированного файла должно выглядеть как команда writer.close(), которая нужна для внесения данных в файл, закрытие файла и сохранения файла на диске. Сгенерированный файл выглядит следующим образом:

	A	B	C	D	E
1	<b>Name</b>	<b>Salary</b>			
2	Роналдиньо	6500			
3	Криштиану Роналду	3300			
4	Ян Облак	12500			
5					
6					

Рисунок 9.3.3 – Файл в формате xlsx, включающих 3 листа данных

В получившемся файле Excel будут три листа: “Группа 1”, “Группа 2” и “Группа 3”. На каждом из этих листов отображены имена сотрудников и их зарплаты, которые соответствуют данным из трех DataFrame в коде.



## Чтение файлов Excel

Аналогично записи DataFrame в файл Excel, можно также читать данные из файлов Excel и сохранять их в объект DataFrame. Для этого используется функция `read_excel()`. Содержимое полученного DataFrame можно просмотреть с помощью метода `head()`.

```
9.3.6  players = pd.read_excel('./players.xlsx')
        players.head()
```

OUT:

*Примечание: метод чтения `read_excel()` из файла Excel является наиболее простым, но он позволяет получить данные только из первого листа.*

В Pandas при использовании метода `read_excel()` по умолчанию строкам DataFrame присваивается метка или числовой индекс. Однако, это поведение можно изменить, передав в качестве параметра `index_col` имя одной из колонок из файла Excel:

```
9.3.7  players = pd.read_excel('./players.xlsx',
        index_col='Name')
        players.head()
```

Name	FC
Роналдиньо	Флуминенсе
Криштиану Роналду	Ювентус
Ян Облак	Атлетико Мадрид

Рисунок 9.3.4 – Вывод метода `head()` листа с информации о футбольном клубе игроков

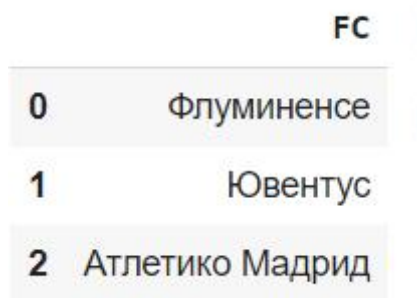
В данном примере стандартный индекс был заменен на колонку “Name” из файла. Важно помнить, что этот способ применим только в случае, если файл содержит колонку с уникальными значениями, которые можно использовать в качестве альтернативы индексам.

## Чтение определенных колонок из файла Excel

В некоторых случаях возникает необходимость получить доступ к конкретному элементу таблицы. Например, может потребоваться считать значение элемента и присвоить его полю объекта.

Для этого можно использовать метод `read_excel()` с параметром `usecols`. Например, чтобы ограничить чтение только определенными колонками, можно добавить параметр `usecols`, указав числовой индекс нужной колонки.

```
9.3.8 cols = [1]
      players = pd.read_excel('./ players.xlsx', usecols=cols)
      players.head()
```



	FC
0	Флуминенсе
1	Ювентус
2	Атлетико Мадрид

Рисунок 9.3.5 – Вывод метода `head()` листа с информации только столбца FC

`DataFrame` обладает широким спектром встроенных функций, которые упрощают операции с данными, такие как изменение, добавление, агрегирование и построение сводных таблиц. Подробнее о методах работы с данными пакета `Pandas` можно узнать по ссылке [7].

## 9.4. Работа с файловой системой

Для работы с файловой системой в Python используют модули `os`, `os.path` и `shutil`, а для операций с файлами.

Важный момент при работе с файловой системой – корректный формат пути к файлам и каталогам. Стандартный для Windows путь `'C:\Users\User\Python\test.py'` считается некорректным для Python.

Варианты исправления пути к файлу `test.py`:

```
9.4.1 'C:/Users/User/Python/test.py' #обратные символы \
      'C:\\Users\\User\\Python\\test.py' #экранирование символа \
      r'C:\Users\User\Python\test.py' #r-строка
```

## Модуль os

Таблица 9.4.1 – Методы модуля os

Метод	Описание	Комментарий
<code>getcwd()</code>	Возвращает путь к текущей рабочей директории в виде строки	
<code>listdir()</code>	Возвращает список всех поддиректорий и файлов текущего рабочего каталога	Содержимое вложенных папок не отображается
<code>makedirs(path)</code>	Создать новую директорию	В качестве параметра принимает полный путь <code>path</code> , включающий название нового каталога. Путь можно получить с помощью <code>getcwd()</code> . См. пример 9.4.2
<code>chdir(path)</code>	Сменить текущую рабочую директорию на <code>path</code>	В качестве параметра принимает новый путь к директории. Может привести к ошибке, если путь к переданной директории указан неправильно или не существует. Метод стоит использовать только с обработкой ошибок. См. пример 9.4.3
<code>makedirs(path)</code>	Создать целевую папку <code>path</code> и все промежуточные директории, если они не существуют	
<code>remove(path)</code>	Удаление файлов	В качестве параметра принимает путь <code>path</code> к директории, из которой будут удалены файлы
<code>walk(path)</code>	Возвращает генератор, в котором содержится вся информация о рабочем каталоге <code>path</code> , включая содержимое всех поддиректорий	См. пример 9.4.4

## Модуль os.path

Таблица 9.4.2 – Методы модуля os.path

Метод	Описание	Комментарий
<code>basename (path)</code>	Извлечь имя файла из полного пути (path)	
<code>dirname (path)</code>	Получить путь (path) к директории, исключая название поддиректории или файла	См. пример 9.4.5
<code>isabs (path)</code>	Возвращает True, если путь (path) до файла абсолютный, False - если путь относительный	Абсолютный путь - всегда начинается с корневого каталога Относительный путь - указывает местоположение файла относительно папки модели См. пример 9.4.6
<code>isdir (path)</code>	Возвращает True, если переданная в метод директория (path) существует, и False – в противном случае	
<code>isfile (path)</code>	Возвращает True, если файл с указанным путем (path) существует, и False – в противном случае	
<code>exists (path)</code>	Возвращает True, если файл или каталог (path) существует, и False – в противном случае	
<code>normcase (path)</code>	Используется для преобразования пути (path) в нижний регистр и нормализации символа \	См. пример 9.4.7
<code>join (path1, path2)</code>	Используется вместо конкатенации строк для получения пути к файлу из названий директории и файла	См. пример 9.4.8

```

9.4.2 import os
      my_cwd = os.getcwd()
      new_dir = 'Tests'
      path = os.path.join(my_cwd, new_dir) #путь + директория
      os.mkdir(path) #создание директории
      print(os.listdir()) #вывод директорий текущего каталога

```

```
9.4.3 import os

my_cwd = os.getcwd() #Текущая директория
new_cwd = r'C:\Users\User\Python\MyData' #Новая директория

try:
    os.chdir(new_cwd)
    print('Смена рабочей директории на ', os.getcwd())
except:
    print('Произошла ошибка')
finally:
    print('Текущая рабочая директория - ', os.getcwd())
```

```
9.4.4 import os
my_cwd = os.getcwd()
result = os.walk(my_cwd)
for i, j, k in result:
    for file in k:
        print(file)
```

```
9.4.5 import os
path = os.path.dirname(r'C:\Users\User\Python\Data')
print(path)
```

OUT: C:\\Users\\User\\Python

```
9.4.6 import os
path1 = os.path.isabs(r'C:\Users\User\Python\Data')
path2 = os.path.isabs(r'.\User\Python\Data')
print(path1, path2)
```

OUT: True False

```
9.4.7 import os
path = os.path.normcase('C:/Users/User/Python')
print(path)
```

OUT: c:\\users\\user\\python

```
9.4.8 import os
my_cwd = os.getcwd()
file_name = 'mytest.py'
path = os.path.join(my_cwd, file_name)
```

---

OUT: C:\Users\User\Python\mytest.py

---

## Модуль shutil

Таблица 9.4.3 – Методы модуля shutil

Метод	Описание	Комментарий
<code>copy(path1, path2)</code>	Копирование файлов	В отличие от <code>copy2()</code> копирует только содержимое файла, но не метаданные
<code>copy2(path1, path2)</code>	Копирование файлов	<code>path_1</code> - путь, с которого файлы будут скопированы <code>path_2</code> - путь, в который файлы будут скопированы
<code>copytree(path1, path2)</code>	Копирование всего содержимое каталога <code>path1</code> в каталог <code>path2</code>	
<code>rmtree(path)</code>	Удаление директории <code>path</code> вместе со всеми файлами	
<code>move(path_1, path_2)</code>	Перемещение файлов с каталога <code>path_1</code> в каталог <code>path_2</code>	

## Модуль sys

Модуль `sys` включает в себя функции и переменные, которые позволяют взаимодействовать с интерпретатором Python и операционной системой.

Этот модуль также предоставляет переменные и функции для управления процессом Python, включая завершение программы, вывод сообщений об ошибках, доступ к переменным окружения и другие.

Модуль `sys` широко используется в Python для управления конфигурацией и окружением выполнения программ, обработки аргументов командной строки, доступа к стандартным потокам ввода/вывода/ошибок, и других задач.

Таблица 9.4.3 – Методы модуля sys

Метод	Описание	Комментарий
-------	----------	-------------

<code>argv</code>		
<code>platform</code>	Вывод информации о платформе	win32, linux или др.
<code>setrecursionlimit(n)</code>	Изменение максимальной глубины рекурсии вызовов на число <code>n</code>	Ограничение на максимальную глубину рекурсии предусмотрено, чтобы предотвратить переполнение стека и последующий сбой программы. Это ограничение обычно установлено на достаточно высоком уровне (порядка 1000)
<code>path</code>	Список директорий, в которых интерпретатор Python ищет модули для импорта	
<code>stdin</code>	Стандартный поток ввода	Поток ввода — это объект в программе, куда попадает весь текст, который ввёл пользователь. Данные хранятся в нем до тех пор, пока программа их не прочитала. Данные поступают в программу и временно сохраняются в потоке ввода, а программа может забрать их оттуда, например, с помощью функции <code>input()</code> . После прочтения, данные пропадают из потока ввода. Для завершения ввода необходимо ввести <code>Ctrl + D</code> См. пример 9.4.9
<code>stdout</code>	Стандартный поток вывода	По умолчанию функция <code>print()</code> перенаправляет вывод данных именно в <code>sys.stdout</code> Отличие от <code>print()</code> : 1. Строка не переводится на новую после вывода; 2. Данные в выводе сохраняю свой тип данных, не переводятся в строковый;

		См. пример 9.4.10
stderr	Стандартный поток вывода ошибок	Стандартные исключения направлены в стандартный вывод stdout. Направить вывод ошибок в отдельный поток stderr можно способами, показанными в примере 9.4.11.

```
9.4.9 import sys
      for line in sys.stdin:      #Для каждой введенной строки
          print(line.strip('\n')) #Вывод с доп. переводом строки
```

---

OUT:   Первая строка  
       Первая строка  
       Вторая строка  
       Вторая строка

---

```
9.4.10 import sys

      print('Hello')
      sys.stdout.write('world!')
      print('I like ')
      sys.stdout.write('python\n')
```

---

OUT:   Hello  
       world!I like  
       python

---

```
9.4.11 import sys
      sys.stderr.write("Произошла ошибка №1\n")

      print("Произошла ошибка №2", file=sys.stderr)
```

---

OUT:   Произошла ошибка №1  
       Произошла ошибка №2

---

## Контрольные вопросы

1. В чем отличие открытие файла с помощью функции `open()` от конструкции `with open ... as ...`?
2. Какие модули вы знаете для работы с разными типами файлов?
3. Как можно переименовать файл с помощью модуля `os`?



## 10. Основы работы с популярными библиотеками

### 10.1. Библиотека Random

Модуль `random` включает в себя функции для генерации случайных чисел, букв, случайного выбора элементов последовательности.

Таблица 10.1.1 – Основные функции библиотеки `random`

Функция	Описание
<code>random()</code>	Генерация случайного числа от 0 до 1
<code>randint(a, b)</code>	Генерация случайного целого числа $n$ , $a \leq n \leq b$
<code>randrange(a, b, step)</code>	Генерация случайно выбранного число из последовательности от $a$ до $b$ с шагом $step$
<code>uniform(a, b)</code>	Генерация случайного числа $n$ с плавающей точкой, $a \leq n \leq b$
<code>choice(L)</code>	Выбор случайного элемента непустой последовательности $L$
<code>shuffle(L)</code>	Перемешивание последовательности $L$ . Последовательность изменяется, поэтому функция не работает для неизменяемых объектов. См. пример 10.1.1

```
10.1.1 from random import *
a = [1, 2, 3, 4, 5]
shuffle(a)
print(a)
```

```
OUT: [5, 3, 1, 2, 4]
```

`random` – псевдослучайный генератор случайных чисел, который использует алгоритм для создания последовательности чисел. Для запуска генерации необходим начальный “seed” (начальное состояние), который определяет всю последующую последовательность. Если задать одинаковый  $x$  в методе `seed(x)` – будет генерироваться одна и та же последовательность.

```
10.1.2 import random
print([random.randint(0, 30) for i in range(5)])
print([random.randint(0, 30) for i in range(5)])
random.seed(5)
print([random.randint(0, 30) for i in range(5)])
random.seed(5)
```

```
print([random.randint(0, 30) for i in range(5)])
```

```
OUT:  [12, 24, 7, 0, 22]
      [13, 10, 29, 4, 17]
      [19, 8, 23, 11, 25]
      [19, 8, 23, 11, 25]
```

## 10.2. Библиотека Math

Для работы с математическими функциями нужно импортировать библиотеку `math`:

```
import math
```

После этого к функциям из этой библиотеки можно обращаться следующим образом:

```
math.имя_функции(...)
```

Таблица 10.2.1 – Функции библиотеки `math`

Функция	Описание
<code>exp(x)</code>	Возвращает экспоненциальное значение $x$ с основой натурального логарифма
<code>fabs(x)</code>	Возвращает абсолютное значение $x$ (математический модуль)
<code>log(x[, base])</code>	Функция возвращает натуральный логарифм от первого аргумента $x$ . Если передается второй аргумент <code>base</code> , он интерпретируется как основание логарифма.
<code>log10(x)</code>	Возвращает десятичный логарифм $x$ .
<code>pow(x, y)</code>	Возвращает $x$ в степени $y$ . В отличие от операции <code>**</code> приводит оба аргумента к типу <code>float</code> .
<code>sqrt(x)</code>	Квадратный корень (square root) из $x$ .

Таблица 10.2.2 – Тригонометрические функции библиотеки math

Функция	Описание
<code>acos(x)</code>	Возвращает арккосинус $x$ , в радианах.
<code>asin(x)</code>	Возвращает арксинус $x$ , в радианах.
<code>atan(x)</code>	Возвращает арктангенс $x$ , в радианах.
<code>atan2(y, x)</code>	Вычисляет арктангенс от отношения $y/x$ и возвращает результат в радианах. Полученное значение находится в интервале $[-\pi; \pi]$ . Угол определяется вектором, который соединяет начало координат с точкой $(x, y)$ , и измеряется относительно положительного направления оси $X$ .
<code>cos(x)</code>	Возвращает косинус $x$ , где $x$ выражен в радианах.
<code>hyp(x, y)</code>	Возвращает $\sqrt{x^2+y^2}$ . Удобно для вычисления гипотенузы ( <code>hyp</code> ) и длины вектора.
<code>sin(x)</code>	Возвращает синус $x$ , где $x$ выражен в радианах.
<code>tan(x)</code>	Возвращает тангенс $x$ , где $x$ выражен в радианах.
<code>degrees(x)</code>	Конвертирует значение угла $x$ из радиан в градусы.
<code>radians(x)</code>	Конвертирует значение угла $x$ из градусов в радианы.

Для округления чисел в Python можно использовать функции `int()`, которая отбрасывает дробную часть, округляя к нулю, и `round()`, которая округляет до ближайшего целого числа. В случае, когда дробная часть равна 0.5, `round()` округляет к ближайшему четному числу.

Библиотека `math` предоставляет дополнительные функции для округления, которые представлены в таблице ниже:

Таблица 10.2.3 – Округление чисел функциями библиотеки math

Функция	Описание
<code>ceil(x)</code>	Округляет $x$ в большую сторону
<code>floor(x)</code>	Округляет $x$ в меньшую сторону
<code>trunc(x)</code>	Работает аналогично <code>int(x)</code>

Таблица 10.2.4 – Примеры округления для различных чисел

Функция	3.5	4.5	-2.5
<code>int(x)</code>	3	4	-2
<code>round(x)</code>	4	4	-2
<code>ceil(x)</code>	4	5	-2
<code>floor(x)</code>	3	4	-3
<code>trunc(x)</code>	3	4	-2

Пример программы:

```
10.2.1 from math import *      # Импорт библиотеки math

def my_function(x):
    x = fabs(x) # Значение будет положительным
    y = sqrt(x) # Извлечение квадратного корня
    y = exp(sin(y) + 1) # Синус плюс 1 и это выражение в
    показатель экспоненты
    return y

print(my_function(-2))

OUT: 7.299208707109954
```

Запись программы примера 10.2.1 без переменных (в функциональном стиле):

```
10.2.2 from math import *

def my_function(x):
    return exp(sin(sqrt(fabs(x))) + 1)

print(my_function(2))

OUT: 7.299208707109954
```

### 10.3. Библиотека Numpy

NumPy — это библиотека Python, которая расширяет возможности языка, предоставляя поддержку для работы с многомерными массивами и матрицами. Библиотека также включает обширный набор высокоуровневых математических функций, оптимизированных для эффективной работы с этими массивами.

Ключевым объектом NumPy является многомерный массив `ndarray`. Он представляет собой структуру данных, которая хранит элементы (чаще всего числа) одного типа, упорядоченные в многомерном формате.

Таблица 10.3.1 – Наиболее важные атрибуты объектов `ndarray`

Атрибут	Описание
<code>ndarray.ndim</code>	Число измерений, так называемые оси массива. См. пример 10.3.4
<code>ndarray.shape</code>	Размер массива или его форма. Описывается кортежем натуральных чисел, где каждый элемент указывает количество элементов вдоль соответствующей оси. Для матрицы из $n$ строк и $m$ столбцов форма будет представлена кортежем $(n, m)$ . Количество элементов в этом кортеже соответствует размерности массива ( <code>ndim</code> ). См. пример 10.3.4
<code>ndarray.size</code>	Количество элементов массива. Равно произведению всех элементов атрибута <code>shape</code>
<code>ndarray.dtype</code>	Тип элементов массива. Для элементов массива могут быть использованы стандартные типы данных: <code>bool_</code> , <code>character</code> , <code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code> , <code>float8</code> , <code>float16</code> , <code>float32</code> , <code>float64</code> , <code>complex64</code> , <code>object_</code> ; возможно определить собственные типы данных. См. пример 10.3.3
<code>ndarray.itemsize</code>	Размер каждого элемента массива в байтах
<code>ndarray.data</code>	Буфер, содержащий фактические элементы массива.

Есть несколько путей импорта. Стандартный метод это — использовать простое выражение:

```
10.3.1 import numpy
```

При использовании большого количества функций из библиотеки NumPy, постоянное написание `numpy.имя_функции` может становиться утомительным. Для удобства, можно использовать сокращенную запись `np.имя_функции` вместо `numpy.имя_функции`.

```
10.3.2 import numpy as np
```

## Создание массивов

Один из самых простых способов создать массив в NumPy — это использовать `numpy.array()`. Эта функция принимает на вход обычный список или кортеж и создает из него объект типа `ndarray`, который является основным типом массива в NumPy.

```
10.3.3 import numpy as np
      a = np.array([1, 2, 3, 7, 9])
      print(a)
      print(a.dtype)
```

```
OUT:  [1 2 3 7 9]
      int64
```

Для трансформации вложенных последовательностей в многомерные массивы следует использовать функцию `array()`. Тип элементов массива останется такой же, как и в исходной последовательности, но в момент создания его можно переопределить.

```
10.3.4 import numpy as np
      b = np.array([[2.6, 4, 7], [1, 7, 9]])
      print(b)
      print('ndim:', b.ndim)
      print('shape: ', b.shape)
```

```
OUT:  ndim: 2
      shape: (2, 3)
```

Переопределение типа элементов в момент создания последовательности:

```
10.3.5 import numpy as np
      b = np.array([[2.6, 4, 7], [1, 7, 9]], dtype=np.complex)
      print(b)
```

```
OUT:  [[2.6+0.j 4. +0.j 7. +0.j]
      [1. +0.j 7. +0.j 9. +0.j]]
```

Задача: элементы массива неизвестны, но массив нужно объявить. Для этого в NumPy предусмотрены функции `zeros()` и `ones()`, которые создают массивы со значениями, заполненными нулями и единицами соответственно. Обе функции принимают кортеж с размерами будущего массива.

```
10.3.6 import numpy as np
      a = np.zeros((2, 4))
      print(a)
      b = np.ones((1, 2, 3))
      print(b)
```

```
OUT:  [[0. 0. 0. 0.]
      [0. 0. 0. 0.]]
```

```
[[[1. 1. 1.]
```

---

```
[1. 1. 1.]])
```

---

Функция `eye()` создает единичную матрицу (двумерный массив) :

```
10.3.7 import numpy as np
a = np.eye(4)
print(a)
```

```
OUT:  [[1. 0. 0. 0.]
       [0. 1. 0. 0.]
       [0. 0. 1. 0.]
       [0. 0. 0. 1.]
```

Функция `empty()` позволяет создать массив, который не заполнен данными. Его начальное содержимое является случайным и определяется содержимым памяти на момент создания массива.

```
10.3.8 import numpy as np
a = np.empty((3, 3))
b = np.empty((3, 3))
print(a)
print(b)
```

```
OUT:  [[1.14253075e-313  0.00000000e+000  4.67980596e-310]
       [4.67980596e-310  4.67980596e-310  4.67980596e-310]
       [4.67980596e-310  4.67980596e-310  4.67980596e-310]]
       [[6.92042957e-310  6.92042957e-310  1.31609791e-070]
       [6.92042813e-310  6.92042814e-310  4.78568450e-225]
       [6.92042813e-310  6.92042814e-310  2.40499582e+087]]
```

Функция `arange()` похожа на встроенную в Python `range()`. Однако, вместо списков она возвращает массивы и может принимать не только целые значения:

```
10.3.9 import numpy as np
a = np.arange(10, 30, 4)
b = np.arange(0, 1, 0.2)
print(a, '\n')
print(b)
```

```
OUT:  [10 14 18 22 26]
```

```
[0.  0.2 0.4 0.6 0.8]
```

---

Использование функции `arange()` с аргументами типа `float` может привести к неопределённому количеству элементов из-за ограничений точности чисел с плавающей запятой.

В таких случаях рекомендуется использовать функцию `linspace()`, которая позволяет указать точное количество элементов в качестве одного из аргументов, что обеспечивает предсказуемость результата.

```
10.3.10 import numpy as np
        print(np.linspace(0, 2, 9)) # 9 чисел от 0 до 2 включит-
        но
OUT:      [0.    0.25 0.5   0.75 1.    1.25 1.5   1.75 2.    ]
```

Функция `fromfunction()`: применяет функцию, переданную первым аргументом, ко всем комбинациям индексов:

```
10.3.11 import numpy as np
        def f1(i, j):
            return 2 * i + j
        print(np.fromfunction(f1, (3, 4)))
OUT:      [[0. 1. 2. 3.]
           [2. 3. 4. 5.]
           [4. 5. 6. 7.]]
```

### Печать (вывод) массива

Если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только начало и конец:

```
10.3.12 import numpy as np
        print(np.arange(0, 1000, 1))
OUT:      [ 0    1    2 ..., 997 998 999]
```

Для полного вывода массива в NumPy используется функция `numpy.set_printoptions()`. Она позволяет настроить параметры печати массивов в соответствии с требованиями.

Функция `numpy.set_printoptions()` принимает аргументы:

`precision`: количество цифр после запятой (по умолчанию 8).

`threshold`: максимальное количество элементов массива, которое выводится полностью (по умолчанию 1000).

`edgeitems`: количество элементов в начале и конце каждой размерности массива, которые выводятся, если количество элементов превышает `threshold` (по умолчанию 3).

`linewidth`: максимальное количество символов в строке (по умолчанию 75).

`suppress`: если `True`, не выводит маленькие значения в экспоненциальной форме (по умолчанию `False`).

`nanstr`: строковое представление NaN (по умолчанию 'nan').

`infstr`: строковое представление inf (по умолчанию 'inf').

`formatter`: позволяет более тонко настроить форматирование печати массивов.



```

10.3.13 import numpy as np
        np.set_printoptions(threshold=30)
        print(np.arange(0, 30, 1), '\n')

        np.set_printoptions(threshold=20, linewidth=20,
                             precision=2)
        print(np.linspace(0, 2, 10))
OUT:    [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
        19 20 21 22 23 24 25 26 27 28 29]

        [0.    0.22 0.44
         0.67 0.89 1.11
         1.33 1.56 1.78
         2.   ]

```

---

### Модификация массива

Метод `reshape()` позволяет изменить форму существующего массива, создавая новый многомерный массив с заданными размерами. В примере ниже одномерный массив из пятнадцати элементов расформируется в двумерный массив, состоящий из трёх строк и пяти столбцов:

```

10.3.14 import numpy as np
        a = np.array(range(15))
        b = a.reshape((3, 5))
        print(a, b, sep='\n')
        print(a.shape, '->', b.shape)
OUT:    [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
        [[ 0  1  2  3  4]
         [ 5  6  7  8  9]
         [10 11 12 13 14]]
        (15,) -> (3, 5)

```

---

Метод `copy()` используется для создания копии существующего массива в памяти:

```

10.3.15 import numpy as np
        a = np.array([1, 2, 3], float)
        b = a
        c = a.copy()
        a[0] = 0
        print(a, b, c, sep='\n')
OUT:    [0. 2. 3.]
        [0. 2. 3.]
        [1. 2. 3.]

```

---

Метод `fill()` используется для заполнения массива одинаковым значением:

```
10.3.16 import numpy as np
        a = np.array([1, 2, 3], float)
        print(a)
        a.fill(0)
        print(a)
```

```
OUT:      [1. 2. 3.]
          [0. 0. 0.]
```

Метод `transpose()` используется для транспонирования, в таком случае создается новый массив:

```
10.3.17 import numpy as np
        a = np.array(range(6), float).reshape((2, 3))
        print(a)
        b = a.transpose()
        print(b)
```

```
OUT:      [[0. 1. 2.]
          [3. 4. 5.]]
          [[0. 3.]
          [1. 4.]
          [2. 5.]]
```

Метод `flatten()` используется для конвертирования массива в одномерный формат:

```
10.3.18 import numpy as np
        a = np.array([[1, 2, 3], [4, 5, 6]], float)
        print(a)
        print(a.flatten())
```

```
OUT:      [[1. 2. 3.]
          [4. 5. 6.]]
          [1. 2. 3. 4. 5. 6.]
```

Метод `concatenate()` используется для конкатенации двух или более массивов (в результирующий массив включаются элементы из всех сконкатенированных массивов). В многомерных массивах можно указать ось (`axis`), вдоль которой будет выполняться соединение элементов. Если ось не задана, соединение происходит по первому измерению массива:

```
10.3.19 import numpy as np
        arr1 = np.array([[10, 20], [40, 60]], float)
        arr2 = np.array([[11, 33], [55, 77]], float)
        print(np.concatenate((arr1, arr2)), end='\n\n')
        print(np.concatenate((arr1, arr2), axis=0), end='\n\n')
        print(np.concatenate((arr1, arr2), axis=1), end='\n\n')
```

```
OUT:      [[10. 20.]
          [40. 60.]
          [11. 33.]
          [55. 77.]]
```

---

```
[[10. 20.]
 [40. 60.]
 [11. 33.]
 [55. 77.]]

[[10. 20. 11. 33.]
 [40. 60. 55. 77.]]
```

---

Размерность массива можно изменить при использовании объекта `newaxis` в квадратных скобках:

```
10.3.20 import numpy as np
        a = np.array([1, 2, 3], float)
        print(a)
        print(a[:,np.newaxis])
        print(a[:,np.newaxis].shape)
        print(a[np.newaxis,:])
        print(a[np.newaxis,:].shape)
```

```
OUT:    [1.  2.  3.]
        [[1.]
         [2.]
         [3.]]
        (3, 1)
        [[1.  2.  3.]]
        (1, 3)
```

---

В этом случае каждый двумерный массив, созданный с помощью `newaxis`, имеет размерность (3, 1), затем тот же массив приводится к размерности (1,3). Объект `newaxis` удобен для создания массивов с нужными размерностями в векторной и матричной математике.

## 10.4. Библиотека Matplotlib

Matplotlib — это широко востребованная библиотека на Python, предназначенная для визуализации данных. В зависимости от целей её можно использовать для построения различных графиков, таких как линейные графики, круговые диаграммы, столбчатые диаграммы и т. д. Импортировать библиотеку принято следующим образом:

```
10.4.1 import matplotlib.pyplot as plt
```

### Базовые приёмы

Построение графика в отдельном окне осуществляется с помощью функции `plot()`. Передав список функция примет это за координаты по оси `y`, в таком случае координаты по оси `x` будут выглядеть как список,

начинающийся 0, количество элементов по оси  $x$  будет равно количеству элементов по оси  $y$ . Цвет линии по умолчанию синий. Для визуализации графика необходимо закончить программу функцией `show()`.

```
10.4.2 import matplotlib.pyplot as plt
      plt.plot([11, 22, 33, 44, 55, 66, 77])
      plt.show()
```

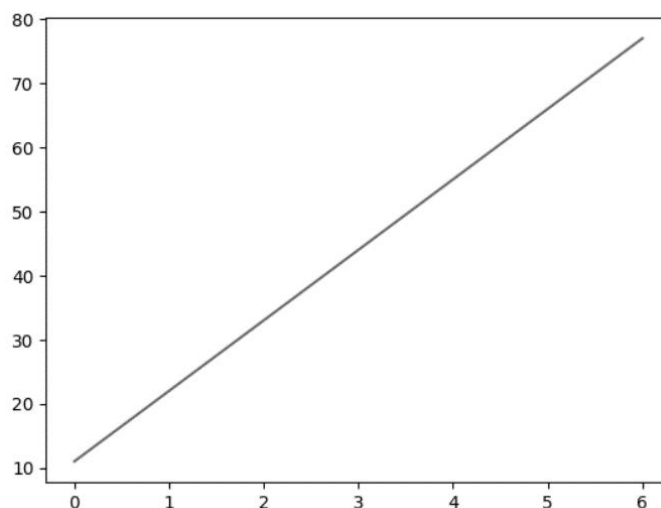


Рисунок 10.4.1 – Результат построения простого графика

Для построения графика по заданным координатам  $x$  и  $y$  следует передать списки со значениями по оси  $x$  и оси  $y$ :

```
10.4.3 import matplotlib.pyplot as plt
      import random as r
      x = [i ** 2 for i in range(5)]
      y = [i ** 5 for i in range(5)]
      print('X: ', x, ' Y:', y)
      plt.plot(x, y)
      plt.show()
```

OUT: [0, 1, 4, 9, 16] Y: [0, 1, 32, 243, 1024]

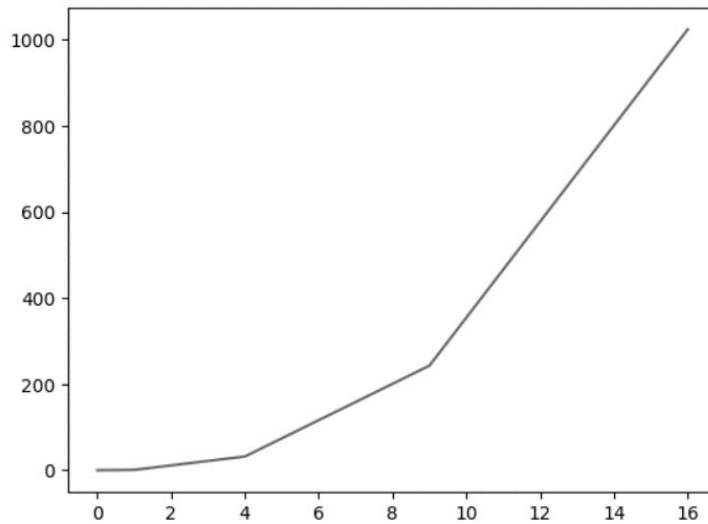


Рисунок 10.4.2 – Результат построения графика по заданным x и y

Функция `plot()` может принимать дополнительный аргумент, который определяет внешний вид точек на графике. Можно указать цвет линии графика или символ, который будет использоваться для отображения точек. Например, `'mo'` установит фиолетовый цвет и круглый символ для точек.

```
10.4.4 import matplotlib.pyplot as plt
      plt.plot([1, 2, 3, 4, 5], [5, 4, 3, 2, 1], 'mo')
      plt.show()
```

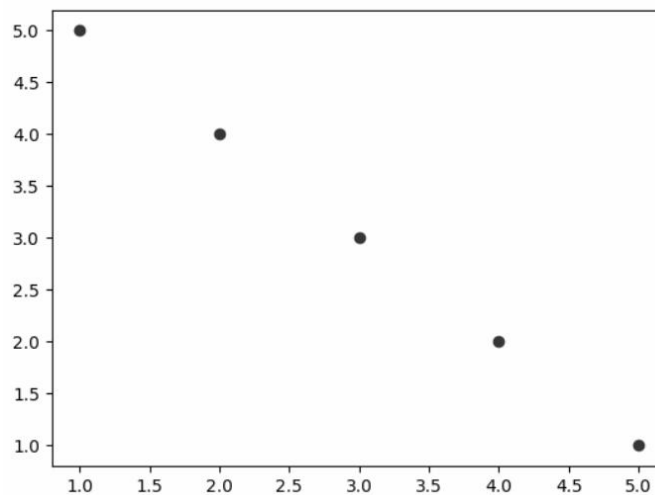


Рисунок 10.4.3 – Результат построения графика заданного цвета и формы

Поменять цвет графика можно записав строку с названием цвета в параметры `plot()`:

```
10.4.5 import matplotlib.pyplot as plt
      plt.plot([1, 2, 3, 4, 5], [5, 4, 3, 2, 1], 'green')
      plt.show()
```

Таблица 10.4.1 – Маркеры отрисовки графиков

Символ	Описание
'.'	Точечный маркер
','	Пиксельный маркер
'o'	Маркер круга
'v'	Маркер треугольника, направленного вниз
'^'	Маркер треугольника, направленного вверх
'<'	Маркер треугольника, направленного влево
'>'	Маркер треугольника, направленного вправо
's'	Квадратный маркер
'p'	Пятиугольный маркер
'P'	Маркет жирного символа +
'+'	Маркер обычного символа +
'*'	Маркер символа звездочка
'x', 'X'	Маркеры символа x и жирного символа <b>x</b> соответственно
'd', 'D'	Маркеры ромба и жирного ромба соответственно
' ', '_'	Маркеры вертикальной и горизонтальной линии соответственно

Таблица 10.4.2 – Стили линий

Символ	Описание
'_'	Сплошная линия
'--'	Пунктирная линия
'-.'	Линия пунктира с точкой
':'	Линия точечного пунктира

Цвет линии в функции `plot()` можно записать как полным названием, так и просто символом. Символы и полные названия цветов представлены в таблице 10.4.3.

Таблица 10.4.3 – Цвета линий

Символ	Цвет	Описание
'b'	'blue'	Синий
'g'	'green'	Зелёный
'r'	'red'	Красный
'c'	'cyan'	Бирюзовый
'm'	'magenta'	Фиолетовый
'y'	'yellow'	Жёлтый
'k'	'black'	Чёрный
'w'	'white'	Белый

### matplotlib и NumPy

Графическая библиотека Matplotlib основана на NumPy. При работе с Matplotlib, передавать списки в качестве аргументов нужно как для представления данных, так и для определения границ осей графика. Внутри Matplotlib эти списки преобразуются в массивы NumPy.

Изобразить график конкретной функции можно следующим образом: задать список значений по оси  $x$  (чем больше значений в списке - тем плавнее график), отдельно задать список значений по оси  $y$ , используя зависимость от  $x$ , в параметрах `plot()` передать значения списков  $x$  и  $y$ . Дополнительно можно указать цвет, а также отрисовать сетку с помощью функции `grid()`:

```
10.4.6 import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 30)
y = [i**2 for i in x]
plt.grid()
plt.plot(x, y, 'black')
```

Чтобы отобразить несколько графиков на одном поле необходимо передавать значения по осям  $x$  и  $y$  в `plot()` последовательно через запятую (см. пример 10.4.7).

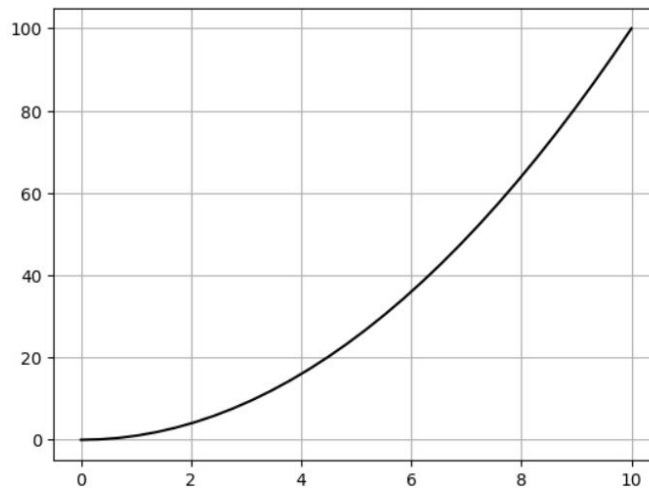


Рисунок 10.4.4 – Результат выполнения программы 10.4.6

```
10.4.7 x = np.linspace(0, 10, 50)
      y1 = x
      y2 = [i**2 for i in x]
      plt.grid()
      plt.plot(x, y1, '--', x, y2, '.')
```

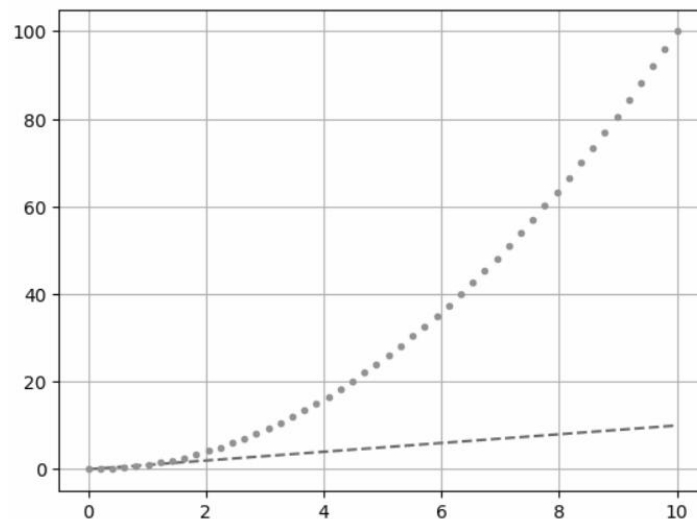


Рисунок 10.4.5 – Результат выполнения программы 10.4.7

Таким образом можно добавлять массивы NumPy в качестве входных. В качестве примера изобразим на одном графике две косинусоиды. Набор точек по оси  $x$  генерируется с помощью функции `arange()`, а для оси  $y$  применяется `cos()` ко всем элементам массива (без цикла `for`).

```
10.4.8 from numpy import *
      x = arange(0,5,0.01)
      y1 = cos(math.pi*x)
      y2 = cos(math.pi*x+math.pi/2)
      plt.plot(x,y1,'--',x,y2,'.')
      plt.show()
```



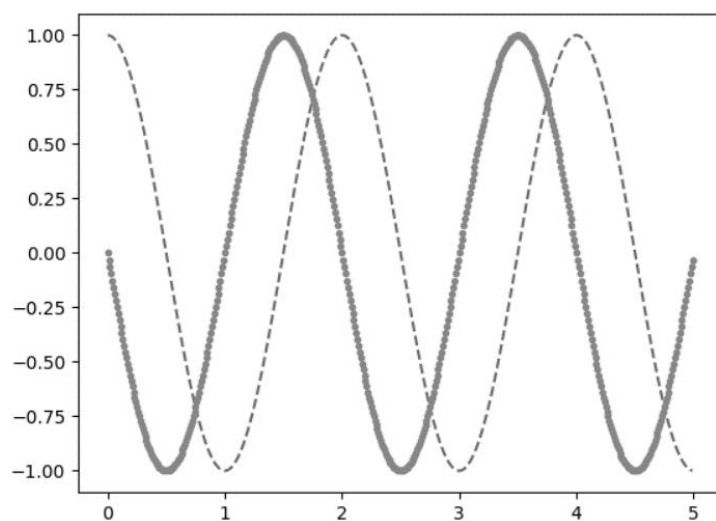


Рисунок 10.4.6 – Результат выполнения программы 10.4.8

## Работа с несколькими объектами и осями

Библиотека Matplotlib позволяет работать с несколькими графиками одновременно, а также создавать подграфики внутри одного объекта. Функция `subplot()` используется как для деления объекта на разные зоны для рисования, так и для указания конкретного подграфика, с которым будет работать последующий код.

```
10.4.9 import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0, 15, 0.05)
x = t * np.sin(t)
y1 = t * np.cos(t)
y2 = t ** 3
plt.subplot(121)
plt.plot(x, y1, 'm-.')
plt.subplot(122)
plt.plot(x, y2, 'c--')
plt.show()
```

Размер общего поля задаётся с помощью функции `figure()`

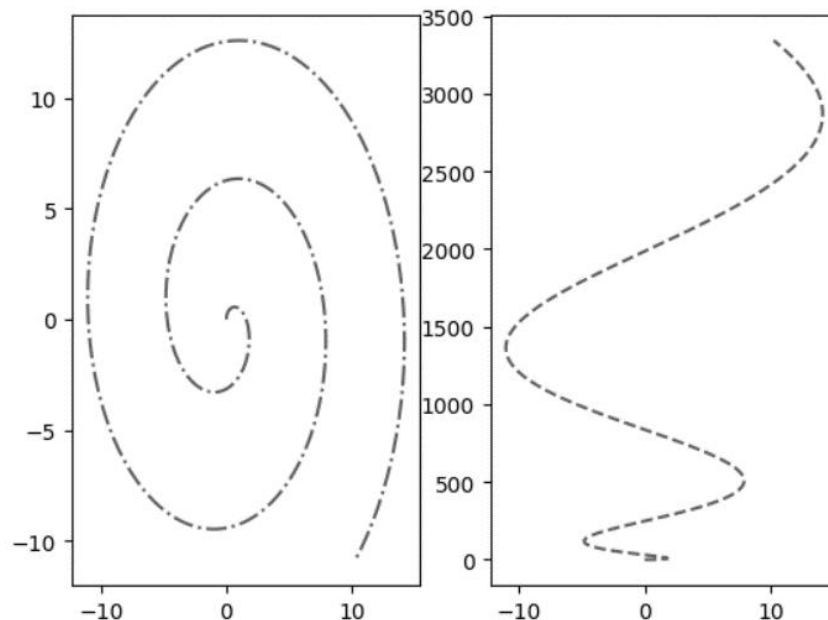


Рисунок 10.4.7 – Результат выполнения программы 10.4.9

Функция `subplot()` используется для определения местоположения подграфика в объекте. Есть несколько способов задать зоны для вывода подграфиков с помощью `subplot()`. В одном из способов первый аргумент указывает количество строк в сетке подграфиков, второй аргумент - количество

столбцов. Третий аргумент - номер подграфика в сетке, который нужно активировать (считается сверху вниз, слева направо).

Аргументы `subplot()` определяют разбиение объекта на зоны и устанавливают текущий подграфик для дальнейшей работы. Аргумент функции `subplot()` состоит из трех целых чисел:

1. Количество частей, на которые нужно разбить объект по вертикали.
2. Количество частей по горизонтали.
3. Номер подграфика, который будет активен для последующих команд.

### **Добавление элементов на график**

Для того, чтобы сделать график более информативным существует множество элементов, которые можно добавить на график и наполнить его дополнительной информацией, например, тестовыми блоками, легендой и т.п.

### **Добавление текста**

Функция `title()` добавляет заголовок к графику. Для добавления меток к осям используются функции `xlabel()` и `ylabel()`. В качестве аргумента они принимают строку, которая будет отображена в качестве метки. В примере 10.4.10 добавлены две метки на график, которые описывают тип значений на

```
10.4.10 plt.title('График')
        plt.axis([0, 10, 0, 100])
        plt.title('График')
        plt.xlabel('Значения x')
        plt.ylabel('Значения x^2')
        plt.plot([3, 5, 7, 9], [9, 25, 49, 81], 'g*')
        plt.show()
```

каждой из осей.

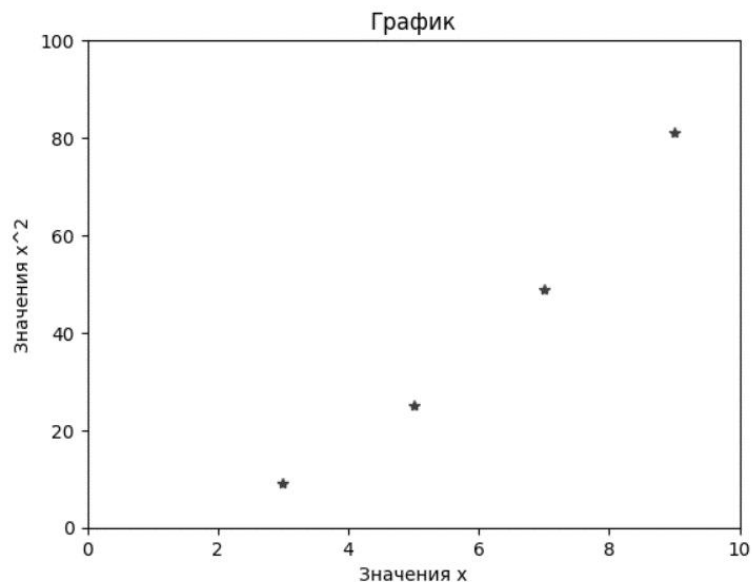


Рисунок 10.4.8 – Результат выполнения программы 10.4.10

Используя ключевые слова, можно изменить свойства текста на графике. Например, можно изменить шрифт и размер заголовка, а также изменить цвет меток осей для выделения заголовка графика. Важно помнить, что не все шрифты поддерживают кириллицу.

```
10.4.11 plt.axis([0, 10, 0, 100])
plt.title('График', fontsize=30, fontname='Consolas')
plt.xlabel('Значения x', color='b')
plt.ylabel('значения x^2', color='b')
plt.plot([3, 5, 7, 9], [9, 25, 49, 81], 'g*')
plt.show()
```

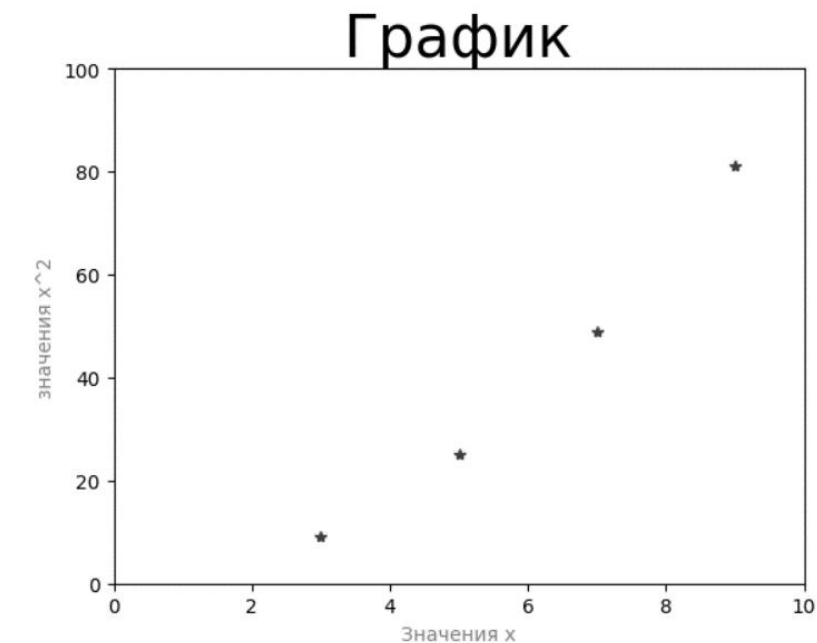


Рисунок 10.4.9 – Результат выполнения программы 10.4.11

В `matplotlib` возможно добавлять текст в любом месте графика с помощью функции `text(x, y, s, fontdict=None)`. Первые два аргумента — это координаты текста, `s` — это строка с текстом, а `fontdict` — шрифт.

```
10.4.12 plt.axis([0, 10, 0, 100])
plt.title('График', fontsize=30, fontname='Consolas')
plt.xlabel('Значения x', color='b')
plt.ylabel('значения x^2', color='b')
plt.plot([3, 5, 7, 9], [9, 25, 49, 81], 'g*')
plt.text(3, 3.5, 'Меркурий')
plt.text(5, 27, 'Венера')
plt.text(7, 52, 'Земля')
plt.text(9, 83, 'Марс')
plt.show()
```

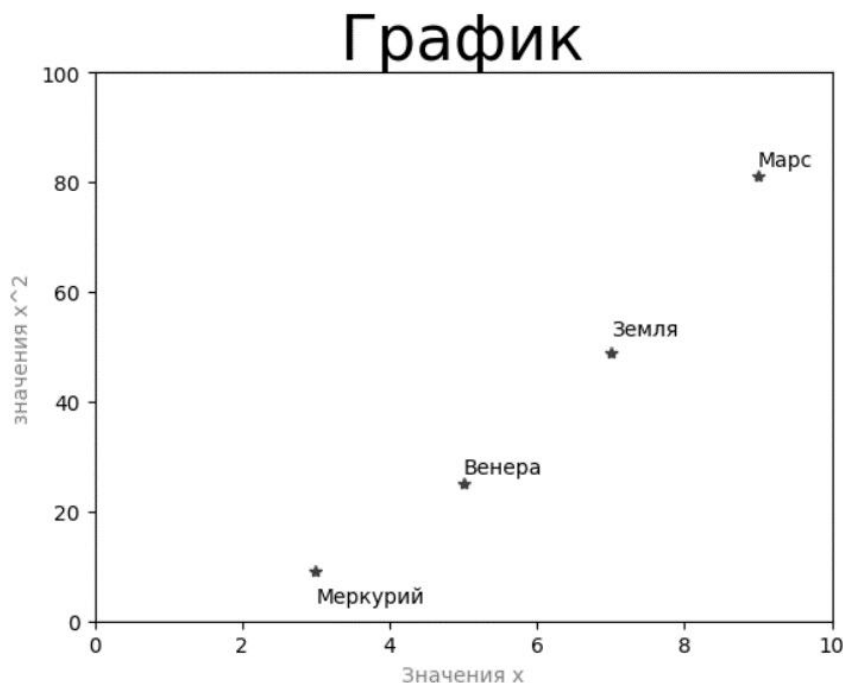


Рисунок 10.4.10 – Результат выполнения программы 10.4.12

### Добавление легенды

На графике зачастую должна присутствовать легенда. Функция `legend()` добавит этот элемент. В функцию нужно передать строку, которая будет отображаться в легенде. В примере 10.4.13 текст "Порядок планет" охарактеризует входящий массив данных.

```
10.4.13 plt.axis([0, 10, 0, 100])
plt.title('График', fontsize=30, fontname='Consolas')
plt.xlabel('Значения x', color='b')
plt.ylabel('значения x^2', color='b')
```

```
plt.plot([3,5,7,9],[9,25,49,81],'g*')
plt.text(3,3.5,'Меркурий')
plt.text(5,27,'Венера')
plt.text(7,52,'Земля')
plt.text(9,83,'Марс')
plt.legend(['Порядок планет'])
plt.show()
```

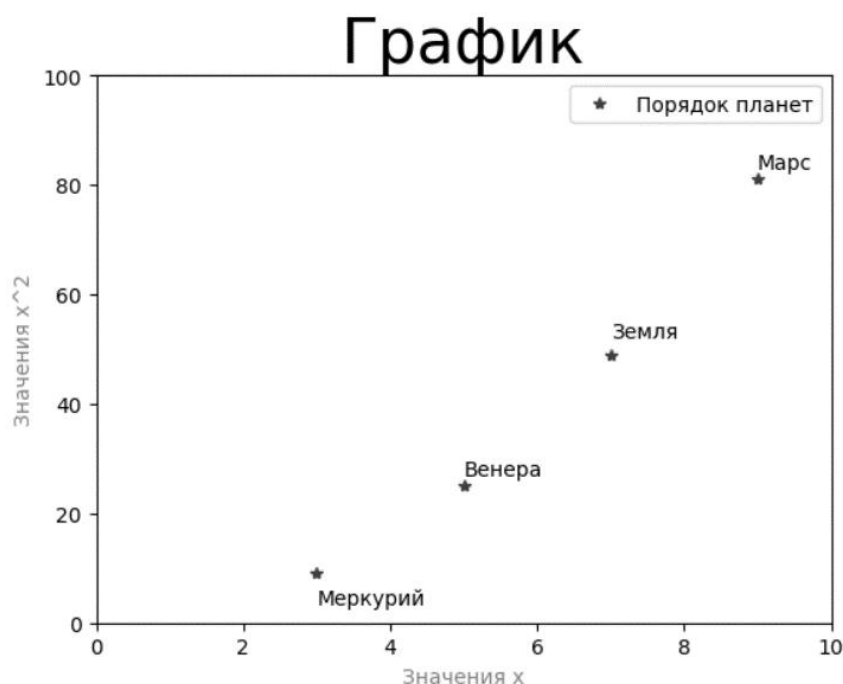


Рисунок 10.4.11 – Результат выполнения программы 10.4.13

По умолчанию, легенда на графике располагается в правом верхнем углу. Для изменения расположения легенды можно использовать ключевое слово `loc` с числовым значением от 0 до 10, где каждое число означает угловое положение. Значение 1 по умолчанию означает правый верхний угол. Все значения положения и кодов описаны в таблице 10.4.4.

Таблица 10.4.4 – Положения легенды

Код	Положение
0	Лучшее
1	Верхний правый угол
2	Верхний левый угол
3	Нижний левый угол
4	Нижний правый угол
5	Справа
6	Слева по центру
7	Справа по центру

8	Снизу по центру
9	Сверху по центру
10	По центру

```

10.4.14 import matplotlib.pyplot as plt
    plt.axis([0,5,0,20])
    plt.axis([0, 10, 0, 100])
    plt.title('График',fontsize=30, fontname='Consolas')
    plt.xlabel('Значения x', color='b')
    plt.ylabel('Значения x^2', color='b')
    plt.plot([3,5,7,9],[9,25,49,81],'g*')
    plt.text(3,3.5,'Меркурий')
    plt.text(5,27,'Венера')
    plt.text(7,52,'Земля')
    plt.text(9,83,'Марс')
    plt.legend(['Порядок планет'], loc=2)
    plt.show()

```

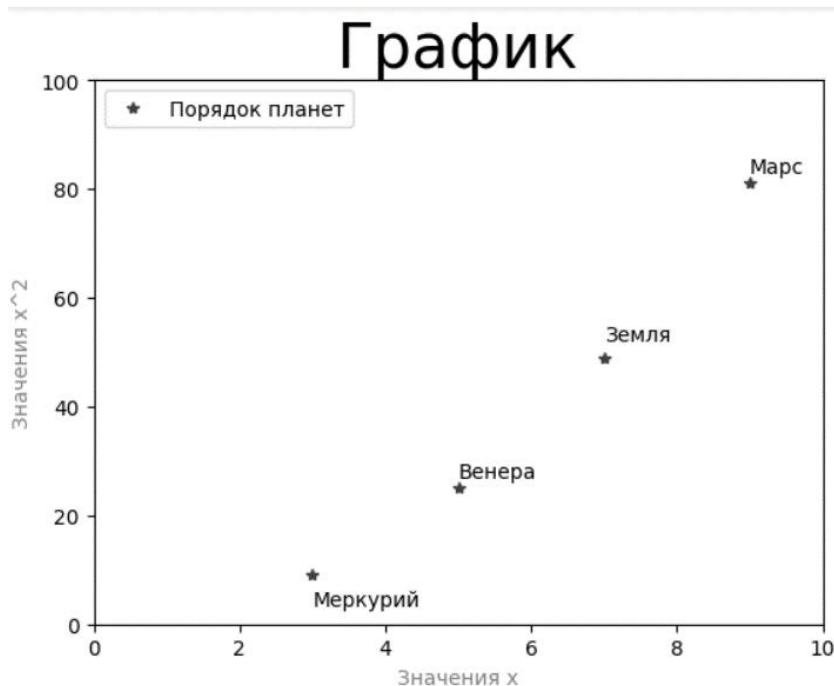


Рисунок 10.4.12 – Результат выполнения программы 10.4.14

График можно сохранить в виде файла-изображения. Для сохранения графика в файл используется функция `savefig()`. В качестве аргумента необходимо указать имя будущего файла. Важно выполнить `savefig()` после всех других команд, иначе будет сохранен пустой PNG-файл.

## Контрольные вопросы

1. Как создать график с использованием двух списков  $x$  и  $y$ ?
2. Как из имеющегося списка псевдослучайно выбрать элемент?
3. Какие существуют функции создания массива с помощью библиотеки NumPy?



## Список литературы

1. OnlineGDBCompiler. [Электронный ресурс]. - Режим доступа — URL: <https://www.onlinegdb.com/>.
2. Google Colab. Google Colaboratory. [Электронный ресурс]. - Режим доступа — URL: <https://colab.research.google.com>.
3. JupyterLite. [Электронный ресурс]. - Режим доступа — URL: <https://jupyter.org/try-jupyter/>.
4. Spyder. [Электронный ресурс]. - Режим доступа — URL: <https://www.spyder-ide.org/>.
5. The Jupyter Notebook. [Электронный ресурс]. - Режим доступа — URL: <https://jupyter.org/>.
6. PyCharm [Электронный ресурс]. - Режим доступа — URL: <https://www.jetbrains.com/pycharm/>.
7. Pandas documentation. Дата: 10.04.2024. Версия: 2.2.2. [Электронный ресурс]. - Режим доступа — URL: <https://pandas.pydata.org/docs/pandas.zip>.
8. Самойленко, Н. Python для сетевых инженеров / Н. Самойленко. — 2023. — 735 с.

Учебное издание

Агалаков Антон Александрович  
Дементьева Кристина Игоревна

## **Программирование на языке Python. Базовый уровень**

Редактор  
Корректор

---

Электронное издание.  
Рекомендовано к публикации  
Редакционно-издательским советом СибГУТИ,  
протокол № \_\_\_\_ от «\_\_\_\_» \_\_\_\_\_ 20\_\_ г