# ***Конспект по Python***

## Содержание

1. [Логический тип данных](#ЛТД_1)
2. [Сравнения в Python](#Сравнения_2)
3. [None](#None_3)
4. [Булева логика](#Boolean_4)
5. [Функция id](#Функция_id_5)
6. [Двойные неравенства](#Двойные_неравенства_6)
7. [Условный оператор if-else](#if_else_7)
8. [Ключевое слово elif](#Ключевое_слово_elif_8)
9. [Вложенные условия](#Вложенные_условия)
10. [Приведение типов к логическому](#Приведение_типов_к_логическому_10)
11. [Тернарный\_оператор](#Тернарный_оператор_11)
12. [Цикл while с предусловием](#Цикл_с_предусловием_12)
13. [Цикл for](#Цикл_for_47)
14. [Функция range()](#Функция_range_48)
15. [Инструкция break](#Инструкция_break_13)
16. [Инструкция continue](#Инструкция_continue_14)
17. [Использование else после while](#Использование_else_после_while_15)
18. [Бесконечный цикл](#Бесконечный_цикл_16)
19. [Ключевое слово pass](#Ключевое_слово_pass_17)
20. [Способы замера времени работы программы](#Замер_времени_работы_программы_18)
21. [Модуль random](#Модуль_random_19)
22. [Представление вещественных чисел в памяти компьютера](#ПВЧПК_20)
23. [Алгоритм перевода чисел в формат с плавающей запятой](#Num21)
24. [Формат с фиксированной запятой для вещественных чисел](#num22)
25. [Понятие модуля в Python](#Понятие_модуля_в_Python_23)
26. [Различные варианты import](#Различные_варианты_import_24)
27. [Библиотека math](#Библиотека_math_25)
28. [Библиотека decimal](#Библиотека_decimal_26)
29. [Запись строк. Типы str, int, float, bool](#Строки_27)
30. [Арифметические операции над строками](#Арифметические_операции_над_строками_28)
31. [Функции len(), ord(), chr()](#Функции_len_ord_chr_29)
32. [Сравнение строк](#Сравнение_строк_30)
33. [Метод in](#Метод_in_31)
34. [Доступ к элементу по индексу](#Доступ_к_элементу_по_индексу_32)
35. [Неизменяемость строк](#Неизменяемость_строк_33)
36. [Срез от строки](#Срез_от_строки_34)
37. [Копия строки](#Копия_строки_35)
38. [Поиск подстроки в строке](#Поиск_подстроки_в_строке_36)
39. [Проверка предиката](#Проверка_предиката_37)
40. [Проверка условия относительно строки](#Проверка_условия_относительно_строки_38)
41. [Методы, изменяющие строку](#Методы_изменяющие_строку_39)
42. [Разделение строки на подстроки](#Разделение_строки_на_подстроки_40)
43. [Выравнивание строк](#Выравнивание_строк_41)
44. [Удаление ненужных символов из строки](#Удаление_ненужных_символов_из_строки_42)
45. [Строковые литералы](#Строковые_литералы_43)
46. [Метод format()](#Метод_format_44)
47. [C-style форматирование](#C_style_форматирование_45)
48. [f-строки](#f_строки_46)
49. [Работа с файлами](#Работа_с_файлами_49)
50. [Инструкции With ... as](#Инструкции_With_as_50)

1: "Логический тип данных"

**Описание и объяснение темы:**

* Логический тип данных в Python представлен двумя значениями: True и False. Логические значения используются для выполнения условий в программировании и являются результатом логических операций.
* В Python True и False также интерпретируются как 1 и 0 соответственно, что позволяет использовать их в арифметических операциях.
* Ключевые логические операторы включают and, or и not, которые помогают комбинировать и инвертировать логические выражения.

**Синтаксис и особенности применения:**

* True и False являются встроенными константами, а их значения чувствительны к регистру.
* Логические значения автоматически возвращаются при использовании операторов сравнения, таких как ==, !=, <, >, <=, >=.

**Примеры кода:**

##### Пример 1: Проверка логического типа данных

x = True

y = False

print(type(x)) # <class 'bool'>

print(type(y)) # <class 'bool'>

##### Пример 2: Логические операции и результаты

a = True

b = False

print(a and b) # False

print(a or b) # True

print(not a) # False

##### Пример 3: Логические значения в арифметических операциях

print(True + 2) # 3

print(False + 5) # 5

##### Пример 4: Присвоение логических значений на основе условий

number = 10

is\_positive = number > 0

print(is\_positive) # True

##### Пример 5: Использование логических значений в циклах и условиях

is\_valid = True

if is\_valid:

print("Данные верны") # Данные верны

else:

print("Данные неверны")

2: "Сравнения в Python"

**Описание и объяснение темы:**

* В Python операторы сравнения используются для сравнения значений и возвращают логическое значение True или False. Операторы сравнения позволяют создавать условия, которые проверяют соответствие между переменными и значениями.
* Основные операторы сравнения:
  + ==: проверяет, равны ли два значения.
  + !=: проверяет, не равны ли два значения.
  + <: проверяет, меньше ли одно значение другого.
  + >: проверяет, больше ли одно значение другого.
  + <=: проверяет, меньше или равно одно значение другому.
  + >=: проверяет, больше или равно одно значение другому.

**Синтаксис и особенности применения:**

* Операторы сравнения часто используются в конструкциях if, while, а также в комбинации с логическими операторами and, or и not для создания сложных условий.
* Важно помнить, что результат выражения сравнения — это булево значение, которое может быть использовано в логических выражениях или напрямую для принятия решений.

**Примеры кода:**

##### Пример 1: Проверка равенства и неравенства

a = 5

b = 10

print(a == b) # False

print(a != b) # True

##### Пример 2: Сравнение с использованием <, >

x = 15

y = 20

print(x < y) # True

print(x > y) # False

##### Пример 3: Проверка диапазона с использованием <=, >=

score = 85

print(score >= 50) # True

print(score <= 100) # True

##### Пример 4: Сравнение строк

word1 = "apple"

word2 = "banana"

print(word1 < word2) # True, сравнение по алфавиту

##### Пример 5: Использование сравнения в условиях

age = 18

if age >= 18:

print("Доступ разрешен") # Доступ разрешен

else:

print("Доступ запрещен")

##### Пример 6: Сравнения внутри сложного условия

temperature = 25

is\_raining = False

if temperature > 20 and not is\_raining:

print("Можно гулять") # Можно гулять

else:

print("Оставайтесь дома")

3: "None"

**Описание и объяснение темы:**

* None в Python — это специальный объект, который представляет отсутствие значения. Он используется для обозначения пустого или неопределенного состояния, аналогичного null в других языках программирования.
* None часто используется в функциях, которые не возвращают значений, или для инициализации переменных, значения которых определяются позже.
* None является объектом, и его тип определяется как NoneType.

**Синтаксис и особенности применения:**

* None используется как самостоятельное значение и проверяется с помощью оператора is, а не ==, чтобы избежать случайных ошибок. Например, x is None.
* Тип NoneType является уникальным, и объект None единственный в своем роде, что обеспечивает его удобное использование как универсального значения для отсутствия данных.

**Примеры кода:**

##### Пример 1: Проверка типа None

x = None

print(type(x)) # <class 'NoneType'>

##### Пример 2: Использование None как значения по умолчанию

def greet(name=None):

if name is None:

print("Привет, гость!")

else:

print(f"Привет, {name}!")

greet() # Привет, гость!

greet("Алексей") # Привет, Алексей!

##### Пример 3: Сравнение с None с использованием is

result = None

if result is None:

print("Результат отсутствует") # Результат отсутствует

##### Пример 4: Инициализация переменной с None

data = None

# позднее в программе значение может быть изменено

data = [1, 2, 3]

print(data) # [1, 2, 3]

##### Пример 5: Функция, возвращающая None

def calculate():

pass # функция пока ничего не делает

result = calculate()

print(result) # None

##### Пример 6: Использование None для завершения цикла

values = [10, None, 20, None, 30]

for value in values:

if value is None:

print("Пропущенное значение")

else:

print(f"Значение: {value}")

# Вывод:

# Значение: 10

# Пропущенное значение

# Значение: 20

# Пропущенное значение

# Значение: 30

4: "Булева логика"

**Описание и объяснение темы:**

* Булева логика в Python основывается на использовании булевых значений (True и False) и логических операторов для построения условий и принятия решений.
* Основные логические операторы:
  + and: возвращает True, если оба выражения истинны.
  + or: возвращает True, если хотя бы одно из выражений истинно.
  + not: инвертирует значение, превращая True в False и наоборот.
* Булева логика позволяет создавать сложные условия и проверять их истинность для управления потоком выполнения программы.

**Синтаксис и особенности применения:**

* Логические операторы могут комбинироваться в сложные выражения, а not используется для инверсии условия.
* Операторы and и or имеют приоритет ниже, чем операторы сравнения, что позволяет комбинировать выражения сравнения в условиях.

**Примеры кода:**

##### Пример 1: Использование and и or для сложных условий

x = 5

y = 10

print(x > 0 and y > 0) # True, оба выражения истинны

print(x > 0 or y < 0) # True, хотя бы одно выражение истинно

##### Пример 2: Использование not для инверсии условия

is\_active = True

print(not is\_active) # False, так как `not` инвертирует значение

##### Пример 3: Логические выражения в условиях

temperature = 30

humidity = 70

if temperature > 25 and humidity > 50:

print("Жарко и влажно") # Жарко и влажно

else:

print("Погода в норме")

##### Пример 4: Вложенные условия с логическими операторами

age = 20

has\_permission = True

if age >= 18 and has\_permission:

print("Доступ разрешен") # Доступ разрешен

else:

print("Доступ запрещен")

##### Пример 5: Использование логики с несколькими условиями

score = 75

passed\_exam = score >= 50

perfect\_score = score == 100

if passed\_exam and not perfect\_score:

print("Экзамен сдан, но не на максимальный балл") # Экзамен сдан, но не на максимальный балл

##### Пример 6: Логика or для проверки одного из нескольких условий

day = "Суббота"

is\_holiday = False

if day == "Суббота" or day == "Воскресенье" or is\_holiday:

print("Выходной день") # Выходной день

else:

5: "Функция id"

**Описание и объяснение темы:**

* Функция id() в Python возвращает уникальный идентификатор для объекта, переданного в качестве аргумента. Этот идентификатор представляет адрес объекта в памяти, который уникален для данного объекта на протяжении его существования.
* id() полезна при работе с изменяемыми и неизменяемыми объектами, так как позволяет проверить, являются ли два объекта одним и тем же в памяти.
* Идентификаторы могут быть полезны для отладки и понимания механизма работы Python с переменными.

**Синтаксис и особенности применения:**

* Синтаксис: id(object), где object — это объект, для которого требуется узнать идентификатор.
* id() особенно полезна для проверки ссылочной природы изменяемых объектов (например, списков и словарей) по сравнению с неизменяемыми объектами (такими как строки и числа).

**Примеры кода:**

##### Пример 1: Получение идентификатора для объекта

x = 42

print(id(x)) # Выводит уникальный идентификатор для объекта 42

##### Пример 2: Сравнение идентификаторов для одинаковых значений

a = 10

b = 10

print(id(a)) # Идентификатор a

print(id(b)) # Идентификатор b, совпадает с a, так как Python кеширует маленькие целые числа

##### Пример 3: Идентификаторы для изменяемых объектов

list1 = [1, 2, 3]

list2 = [1, 2, 3]

print(id(list1)) # Идентификатор list1

print(id(list2)) # Идентификатор list2 отличается, так как это разные объекты

##### Пример 4: Проверка, ссылаются ли переменные на один и тот же объект

x = [1, 2, 3]

y = x

print(id(x)) # Идентификатор x

print(id(y)) # Идентификатор y, совпадает с x, так как y ссылается на тот же объект

##### Пример 5: Идентификаторы для строк (неизменяемый объект)

str1 = "hello"

str2 = "hello"

print(id(str1)) # Идентификатор str1

print(id(str2)) # Идентификатор str2 совпадает с str1, так как строки неизменяемы

##### Пример 6: Использование id() для проверки копий объектов

import copy

original\_list = [1, 2, 3]

shallow\_copy = copy.copy(original\_list)

deep\_copy = copy.deepcopy(original\_list)

print(id(original\_list)) # Идентификатор оригинала

print(id(shallow\_copy)) # Идентификатор поверхностной копии, отличается

print(id(deep\_copy)) # Идентификатор глубокой копии, также отличается

6: "Двойные неравенства"

**Описание и объяснение темы:**

* В Python можно использовать двойные и даже множественные неравенства для проверки, находится ли значение в определённом диапазоне. Это позволяет записывать условия более компактно и легко читаемо.
* Двойные неравенства работают по принципу математики: a < b < c будет возвращать True, если b больше a и меньше c.
* Такие выражения удобны для проверки диапазонов значений без необходимости разбивать условие на несколько выражений с использованием and.

**Синтаксис и особенности применения:**

* Синтаксис: a < b < c или a <= b <= c, где a, b, c — это значения, участвующие в проверке.
* Python обрабатывает каждое сравнение слева направо, что позволяет писать сложные условия более кратко.

**Примеры кода:**

##### Пример 1: Простое двойное неравенство

x = 5

print(1 < x < 10) # True, так как x больше 1 и меньше 10

##### Пример 2: Проверка диапазона значений с использованием <= и >=

score = 75

print(50 <= score <= 100) # True, score находится в диапазоне от 50 до 100 включительно

##### Пример 3: Использование двойного неравенства для фильтрации значений

temperature = 20

if 15 <= temperature <= 25:

print("Температура комфортная") # Температура комфортная

else:

print("Температура некомфортная")

##### Пример 4: Множественное неравенство для нескольких значений

a, b, c = 3, 5, 7

print(a < b < c) # True, так как a < b и b < c

##### Пример 5: Сравнение значений с использованием отрицания

age = 17

if not (13 <= age <= 19):

print("Не является подростком")

else:

print("Подросток") # Подросток

##### Пример 6: Использование неравенств в условиях циклов

for num in range(1, 20):

if 5 < num < 15:

print(num, "входит в диапазон 6-14") # Числа от 6 до 14 входят в диапазон

7: "Условный оператор if-else"

**Описание и объяснение темы:**

* Условный оператор if-else в Python позволяет выполнять разные блоки кода в зависимости от выполнения или невыполнения условия. Это основной инструмент для ветвления логики в программе.
* С помощью if программа проверяет, является ли заданное условие истинным (True). Если условие истинно, выполняется блок кода под if, иначе программа переходит к блоку else, если он указан.
* Если необходимо проверить несколько условий, используется конструкция if-elif-else.

**Синтаксис и особенности применения:**

* Синтаксис:

if условие:

# код, выполняемый, если условие истинно

else:

# код, выполняемый, если условие ложно

* Оператор if поддерживает вложенные условия и комбинации с elif для создания более сложных логических проверок.

**Примеры кода:**

##### Пример 1: Простое использование if-else

age = 18

if age >= 18:

print("Вам разрешено голосовать") # Вывод: Вам разрешено голосовать

else:

print("Вам ещё нет 18")

##### Пример 2: Проверка нескольких условий с if-elif-else

score = 85

if score >= 90:

print("Отлично")

elif score >= 75:

print("Хорошо") # Вывод: Хорошо

else:

print("Нужно постараться")

##### Пример 3: Вложенные условия

age = 20

if age >= 18:

if age >= 21:

print("Вам разрешено покупать алкоголь") # Вывод: Вам разрешено покупать алкоголь

else:

print("Вам разрешено голосовать, но не покупать алкоголь")

else:

print("Вам ещё нет 18")

##### Пример 4: Условный оператор с логическими выражениями

temperature = 30

humidity = 60

if temperature > 25 and humidity < 70:

print("Погода комфортная для прогулки") # Погода комфортная для прогулки

else:

print("Лучше остаться дома")

##### Пример 5: Использование if-else для определения чётности числа

number = 4

if number % 2 == 0:

print("Число чётное") # Число чётное

else:

print("Число нечётное")

##### Пример 6: Условное присваивание с if-else

balance = 1000

withdrawal = 1500

if balance >= withdrawal:

balance -= withdrawal

print("Снятие успешно")

else:

print("Недостаточно средств") # Недостаточно средств

8: "Ключевое слово elif"

**Описание и объяснение темы:**

* Ключевое слово elif используется в Python для проверки нескольких условий в одной конструкции if. Оно сокращает синтаксис цепочек условий, чтобы избежать необходимости использования вложенных if.
* elif — это сокращение от "else if". После выполнения условия if, если оно ложно, программа переходит к проверке условия elif. Если условие elif истинно, соответствующий блок кода выполняется. Если нет, выполнение переходит к следующему elif или else, если они указаны.
* Ключевое слово elif делает код более читаемым, так как позволяет компактно проверять несколько условий подряд.

**Синтаксис и особенности применения:**

* Синтаксис:

if условие1:

# код для условия1

elif условие2:

# код для условия2

else:

# код для остальных случаев

* Важно помнить, что если одно из условий в цепочке if-elif-else истинно, остальные не проверяются, и выполнение переходит к соответствующему блоку.

**Примеры кода:**

##### Пример 1: Простая цепочка if-elif-else

grade = 82

if grade >= 90:

print("Отлично")

elif grade >= 75:

print("Хорошо") # Вывод: Хорошо

elif grade >= 60:

print("Удовлетворительно")

else:

print("Неудовлетворительно")

##### Пример 2: Проверка дня недели

day = "Среда"

if day == "Понедельник":

print("Начало недели")

elif day == "Среда":

print("Середина недели") # Вывод: Середина недели

elif day == "Пятница":

print("Почти выходные")

else:

print("Необычный день")

##### Пример 3: Классификация температуры

temperature = 15

if temperature > 30:

print("Очень жарко")

elif temperature > 20:

print("Тепло")

elif temperature > 10:

print("Прохладно") # Вывод: Прохладно

else:

print("Холодно")

##### Пример 4: Оценка численного балла

score = 92

if score == 100:

print("Идеальный результат!")

elif score >= 90:

print("Отличный результат") # Вывод: Отличный результат

elif score >= 75:

print("Хороший результат")

else:

print("Можно лучше")

##### Пример 5: Выбор активности по времени суток

hour = 14

if hour < 12:

print("Доброе утро")

elif hour < 18:

print("Добрый день") # Вывод: Добрый день

else:

print("Добрый вечер")

##### Пример 6: Проверка возрастных категорий

age = 25

if age < 13:

print("Ребёнок")

elif age < 18:

print("Подросток")

elif age < 60:

print("Взрослый") # Вывод: Взрослый

else:

print("Пожилой человек")

9: "Вложенные условия"

**Описание и объяснение темы:**

* Вложенные условия представляют собой конструкции, где один оператор if находится внутри другого. Это позволяет более детально контролировать выполнение программы, проверяя дополнительные условия внутри основного условия.
* Вложенные условия полезны, когда необходимо принять решение, основанное на нескольких уровнях проверки. Например, можно сначала проверить основное условие, а затем, если оно истинно, переходить к более специфическим проверкам.

**Синтаксис и особенности применения:**

* Синтаксис:

if условие1:

if условие2:

# код, выполняемый при истинности обоих условий

else:

# код, выполняемый при истинности только условия1

else:

# код, выполняемый, если условие1 ложно

* Использование вложенных условий увеличивает вложенность кода, что может сделать его сложнее для понимания. Поэтому вложенные условия рекомендуется применять, когда они действительно необходимы.

**Примеры кода:**

##### Пример 1: Простое вложенное условие

age = 20

has\_ticket = True

if age >= 18:

if has\_ticket:

print("Доступ разрешен") # Вывод: Доступ разрешен

else:

print("Нужен билет")

else:

print("Доступ запрещен")

##### Пример 2: Вложенные условия с проверкой двух условий

grade = 85

attendance = 90

if grade >= 80:

if attendance >= 85:

print("Курс завершен успешно") # Курс завершен успешно

else:

print("Необходима высокая посещаемость")

else:

print("Оценка недостаточно высокая")

##### Пример 3: Проверка возраста и статуса работы

age = 30

is\_employed = True

if age > 18:

if is\_employed:

print("Работающий взрослый") # Работающий взрослый

else:

print("Безработный взрослый")

else:

print("Несовершеннолетний")

##### Пример 4: Проверка множества условий для логического решения

temperature = 25

is\_sunny = True

if temperature > 20:

if is\_sunny:

print("Можно идти на пляж") # Можно идти на пляж

else:

print("Лучше остаться дома")

else:

print("Слишком холодно")

##### Пример 5: Вложенные условия с логическими операторами

day = "Пятница"

is\_holiday = False

if day == "Пятница":

if not is\_holiday:

print("Рабочий день") # Рабочий день

else:

print("Выходной день")

else:

print("Не пятница")

##### Пример 6: Сложное вложенное условие с тремя уровнями

time = 14

day = "Суббота"

weather = "солнечно"

if day == "Суббота" or day == "Воскресенье":

if weather == "солнечно":

if time >= 12:

print("Отличный день для пикника") # Отличный день для пикника

else:

print("Лучше подождать до обеда")

else:

print("Погода не подходит для пикника")

else:

print("Сегодня рабочий день")

10: "Приведение типов к логическому"

**Описание и объяснение темы:**

* В Python большинство типов данных можно приводить к булевому типу (bool). Это преобразование позволяет понять, какое значение будет считаться истинным (True) или ложным (False) в логических операциях и условиях.
* Булевое преобразование полезно в условиях, когда значения данных автоматически приводятся к булевому типу, например, в if или while. Тип данных или значение считается False, если оно пустое или нулевое, в остальных случаях оно считается True.
* Основные правила:
  + Числовые значения: 0 или 0.0 считаются False; все остальные числа — True.
  + Строки: пустая строка "" считается False, непустая строка — True.
  + Списки, кортежи, множества, словари: пустые коллекции считаются False, непустые — True.

**Синтаксис и особенности применения:**

* Синтаксис: bool(значение) — функция bool() приводит значение к логическому типу.
* Использование булевого преобразования позволяет лаконично проверять состояние переменных или объектов без явного сравнения, например, if some\_list: вместо if len(some\_list) > 0.

**Примеры кода:**

##### Пример 1: Приведение числовых значений к логическому типу

print(bool(0)) # False

print(bool(10)) # True

print(bool(-5)) # True

##### Пример 2: Приведение строк к логическому типу

print(bool("")) # False, пустая строка

print(bool("Python")) # True, непустая строка

##### Пример 3: Приведение коллекций к логическому типу

print(bool([])) # False, пустой список

print(bool([1, 2, 3])) # True, непустой список

print(bool({})) # False, пустой словарь

print(bool({"key": "value"})) # True, непустой словарь

##### Пример 4: Приведение None к логическому типу

print(bool(None)) # False, так как None означает отсутствие значения

##### Пример 5: Использование приведения типов в условиях

name = ""

if name:

print("Имя указано")

else:

print("Имя отсутствует") # Вывод: Имя отсутствует

##### Пример 6: Булевое значение выражений и операций

x = 5

y = 0

print(bool(x and y)) # False, так как y равно 0

print(bool(x or y)) # True, так как x не равен 0

11: "Тернарный оператор"

**Описание и объяснение темы:**

* Тернарный оператор в Python позволяет записывать условные выражения в одну строку, что делает код более компактным и читабельным.
* Синтаксис тернарного оператора:

выражение\_if\_true if условие else выражение\_if\_false

* + Если условие истинно (True), выполняется выражение\_if\_true, иначе — выражение\_if\_false.
* Тернарный оператор используется для простых условий, когда результат выражения можно записать лаконично. Он заменяет собой конструкцию if-else, но не подходит для более сложных многострочных логик.

**Синтаксис и особенности применения:**

* Синтаксис: value\_if\_true if condition else value\_if\_false
* Тернарный оператор улучшает читаемость кода, когда требуется выбрать одно из двух значений в зависимости от условия, например, для задания значения по умолчанию или выполнения простой проверки.

**Примеры кода:**

##### Пример 1: Простое условие с тернарным оператором

age = 18

status = "Взрослый" if age >= 18 else "Несовершеннолетний"

print(status) # Вывод: Взрослый

##### Пример 2: Выбор значения в зависимости от переменной

is\_raining = True

activity = "Читать книгу дома" if is\_raining else "Идти гулять"

print(activity) # Вывод: Читать книгу дома

##### Пример 3: Тернарный оператор для задания значения по умолчанию

name = ""

display\_name = name if name else "Гость"

print(display\_name) # Вывод: Гость

##### Пример 4: Использование тернарного оператора в выражении для вычисления

x = 10

y = 20

max\_value = x if x > y else y

print(max\_value) # Вывод: 20

##### Пример 5: Вложенные тернарные операторы

score = 85

result = "Отлично" if score >= 90 else ("Хорошо" if score >= 75 else "Удовлетворительно")

print(result) # Вывод: Хорошо

##### Пример 6: Тернарный оператор для отображения знака числа

number = -5

sign = "Положительное" if number > 0 else "Отрицательное" if number < 0 else "Ноль"

print(sign) # Вывод: Отрицательное

12: "Цикл while с предусловием"

**Описание и объяснение темы:**

* Цикл с предусловием в Python реализуется с помощью оператора while. Он выполняется до тех пор, пока условие истинно (True). Как только условие становится ложным (False), выполнение цикла прекращается.
* Цикл while удобен для ситуаций, когда заранее неизвестно количество итераций, и выполнение зависит от внешнего условия, которое изменяется внутри цикла.

**Синтаксис и особенности применения:**

* Синтаксис:

while условие:

# код, выполняемый, пока условие истинно

* Важно контролировать условие в while, чтобы избежать бесконечного цикла, особенно при работе с переменными, изменяемыми внутри цикла.
* Если требуется выйти из цикла до завершения условия, используется инструкция break.

**Примеры кода:**

##### Пример 1: Базовый цикл while

count = 0

while count < 5:

print("Count:", count) # Выводит числа от 0 до 4

count += 1

##### Пример 2: Цикл while с пользовательским вводом

password = ""

while password != "12345":

password = input("Введите пароль: ")

print("Пароль верный!") # Цикл завершится, когда введён пароль "12345"

##### Пример 3: Использование break для выхода из цикла

number = 0

while True:

print("Число:", number)

number += 1

if number >= 5:

break # Цикл прервётся, когда number станет 5

##### Пример 4: Подсчет суммы чисел до заданного значения

total = 0

num = 1

while num <= 10:

total += num

num += 1

print("Сумма от 1 до 10:", total) # Вывод: 55

##### Пример 5: Использование while для итерации по коллекции

items = ["яблоко", "банан", "груша"]

index = 0

while index < len(items):

print(items[index]) # Выводит: яблоко, банан, груша

index += 1

##### Пример 6: Условие для завершения цикла с обработкой исключений

while True:

try:

number = int(input("Введите число: "))

print("Вы ввели:", number)

break # Цикл прерывается при успешном вводе числа

except ValueError:

print("Ошибка: введите корректное число")

13: "Цикл for"

**Описание и объяснение темы:**

* Цикл for в Python позволяет перебрать элементы коллекции (списка, строки, множества, словаря и т.д.) или итерируемого объекта (например, range). Цикл for выполняет блок кода для каждого элемента в последовательности.
* for часто используется для обработки списков, выполнения действий над каждым элементом коллекции или генерации последовательностей значений с помощью range().

**Синтаксис и особенности применения:**

* Синтаксис:

for элемент in коллекция:

# код для выполнения

* Используется с функцией range() для создания числовых последовательностей.

**Примеры кода:**

##### Пример 1: Перебор элементов списка

fruits = ["яблоко", "банан", "вишня"]

for fruit in fruits:

print(fruit)

# Выводит:

# яблоко

# банан

# вишня

##### Пример 2: Использование for с range() для генерации числовой последовательности

for i in range(5):

print(i)

# Выводит числа от 0 до 4

##### Пример 3: Перебор символов в строке

text = "Python"

for char in text:

print(char)

# Выводит:

# P

# y

# t

# h

# o

# n

##### Пример 4: Перебор элементов словаря

person = {"name": "Alice", "age": 25}

for key, value in person.items():

print(f"{key}: {value}")

# Выводит:

# name: Alice

# age: 25

##### Пример 5: Использование for с range() для задания шага

for i in range(0, 10, 2):

print(i)

# Выводит: 0, 2, 4, 6, 8

##### Пример 6: Создание списка квадратов чисел с помощью цикла for

squares = []

for i in range(1, 6):

squares.append(i\*\*2)

print(squares) # Выводит: [1, 4, 9, 16, 25]

14: "Функция range()"

**Описание и объяснение темы:**

* Функция range() используется для генерации последовательности чисел. Она часто применяется в циклах for для создания итераций с заданным количеством шагов.
* range() позволяет задать начальное значение, конечное значение (не включается в последовательность) и шаг.

**Синтаксис и особенности применения:**

* Синтаксис: range(start, stop, step), где:
  + start (по умолчанию 0): начальное значение.
  + stop: конечное значение, не включается в последовательность.
  + step (по умолчанию 1): шаг, которым будет увеличиваться значение.
* range() возвращает объект range, который является итерируемым и не создает в памяти всю последовательность сразу, что делает его эффективным для больших последовательностей.

**Примеры кода:**

##### Пример 1: Создание последовательности от 0 до 4

for i in range(5):

print(i)

# Выводит: 0, 1, 2, 3, 4

##### Пример 2: Указание начального и конечного значения

for i in range(2, 6):

print(i)

# Выводит: 2, 3, 4, 5

##### Пример 3: Использование отрицательного шага для обратного отсчёта

for i in range(10, 0, -2):

print(i)

# Выводит: 10, 8, 6, 4, 2

##### Пример 4: Преобразование range в список

numbers = list(range(5, 15, 3))

print(numbers) # Выводит: [5, 8, 11, 14]

##### Пример 5: Перебор списка с использованием индексов

fruits = ["яблоко", "банан", "вишня"]

for i in range(len(fruits)):

print(f"Индекс {i}: {fruits[i]}")

# Выводит:

# Индекс 0: яблоко

# Индекс 1: банан

# Индекс 2: вишня

##### Пример 6: Использование range() с пустым результатом при некорректных параметрах

for i in range(5, 2):

print(i)

# Ничего не выводит, так как начальное значение больше конечного

15: "Инструкция break"

**Описание и объяснение темы:**

* Инструкция break используется для немедленного завершения цикла (for или while) вне зависимости от того, выполнено ли основное условие цикла.
* break позволяет выйти из цикла при достижении определённого условия, что особенно полезно в циклах с неограниченным числом итераций или при поиске определённого значения в последовательности.
* После выполнения break выполнение кода продолжится сразу за пределами цикла.

**Синтаксис и особенности применения:**

* Синтаксис:

while условие:

# код цикла

if условие\_для\_прерывания:

break

* break удобно использовать, когда нужно прервать выполнение цикла до завершения всех итераций, особенно в сочетании с условиями.

**Примеры кода:**

##### Пример 1: Простой пример break в цикле while

count = 0

while count < 10:

if count == 5:

break

print("Count:", count) # Выводит числа от 0 до 4

count += 1

##### Пример 2: Использование break для выхода из бесконечного цикла

while True:

user\_input = input("Введите 'stop' для остановки: ")

if user\_input == "stop":

print("Цикл завершён")

break

##### Пример 3: Поиск элемента в списке с break

numbers = [1, 3, 5, 7, 9, 11]

target = 7

for number in numbers:

if number == target:

print("Найдено число:", number) # Найдено число: 7

break

##### Пример 4: break в цикле for с условием

for i in range(10):

print(i)

if i == 6:

break # Цикл завершится, когда i станет 6

# Выводит числа от 0 до 6

##### Пример 5: break в комбинации с while для проверки ввода пользователя

while True:

number = int(input("Введите положительное число: "))

if number > 0:

print("Спасибо!")

break

else:

print("Ошибка: введите положительное число")

##### Пример 6: Прерывание вложенного цикла с помощью break и else

for i in range(5):

for j in range(5):

if j == 3:

break # Прерывает внутренний цикл, когда j равно 3

print(f"i={i}, j={j}")

16: "Инструкция continue"

**Описание и объяснение темы:**

* Инструкция continue используется для перехода к следующей итерации цикла, пропуская оставшийся код в текущей итерации. Она полезна, когда требуется пропустить выполнение части кода при выполнении определённого условия, не прерывая цикл полностью.
* continue чаще всего применяется для пропуска определённых значений в цикле, избегая выполнения лишних действий.

**Синтаксис и особенности применения:**

* Синтаксис:

for элемент in коллекция:

if условие:

continue

# код, который будет выполнен, если условие ложно

* continue позволяет лаконично пропустить ненужные значения в цикле, обеспечивая простоту и читаемость кода.

**Примеры кода:**

##### Пример 1: Пропуск чётных чисел с использованием continue

for number in range(10):

if number % 2 == 0:

continue # Пропускает чётные числа

print("Нечётное число:", number)

# Вывод: 1, 3, 5, 7, 9

##### Пример 2: continue в цикле while для пропуска итерации

count = 0

while count < 5:

count += 1

if count == 3:

continue # Пропускает итерацию, когда count равно 3

print("Count:", count)

# Выводит: 1, 2, 4, 5

##### Пример 3: Пропуск конкретного значения в списке

words = ["apple", "banana", "cherry", "date"]

for word in words:

if word == "banana":

continue # Пропускает слово "banana"

print(word)

# Выводит: apple, cherry, date

##### Пример 4: continue для пропуска отрицательных чисел

numbers = [-5, 3, -1, 7, -2]

for number in numbers:

if number < 0:

continue # Пропускает отрицательные числа

print("Положительное число:", number)

# Выводит: 3, 7

##### Пример 5: Применение continue в цикле для фильтрации значений

for i in range(1, 11):

if i % 3 != 0:

continue # Пропускает числа, не кратные 3

print("Число кратное 3:", i)

# Вывод: 3, 6, 9

##### Пример 6: continue для пропуска неполных данных в списке

data = ["John", "", "Alice", "", "Bob"]

for name in data:

if name == "":

continue # Пропускает пустые строки

print("Имя:", name)

# Выводит: John, Alice, Bob

17: "Использование else после while"

**Описание и объяснение темы:**

* В Python после цикла while можно использовать блок else. Этот блок выполняется, если цикл завершился "естественным" образом, то есть без использования break.
* Блок else может быть полезен для выполнения кода, если условие цикла перестало быть истинным. Например, это удобно для сценариев, где необходимо выполнить действие, если не было выполнено принудительное завершение.

**Синтаксис и особенности применения:**

* Синтаксис:

while условие:

# код цикла

else:

# код, который выполнится, если цикл завершится без break

* Если цикл прерван с помощью break, блок else пропускается.

**Примеры кода:**

##### Пример 1: Простое использование else после while

count = 0

while count < 3:

print("Итерация:", count)

count += 1

else:

print("Цикл завершен без прерывания") # Выводится после цикла

##### Пример 2: Пропуск блока else при break

count = 0

while count < 5:

if count == 3:

break # Прерывает цикл, блок else не выполнится

print("Итерация:", count)

count += 1

else:

print("Цикл завершен без прерывания") # Этот блок не будет выполнен

##### Пример 3: Проверка завершения цикла с использованием else

password = "secret"

attempts = 0

while attempts < 3:

user\_input = input("Введите пароль: ")

if user\_input == password:

print("Доступ разрешен")

break

attempts += 1

else:

print("Превышено количество попыток") # Выполняется, если цикл завершится

##### Пример 4: Использование else для проверки выполнения задачи

numbers = [1, 3, 5, 7]

index = 0

while index < len(numbers):

if numbers[index] % 2 == 0:

print("Найдено чётное число")

break

index += 1

else:

print("Чётное число не найдено") # Выполняется, если цикл завершится

##### Пример 5: Игра угадай число с else после while

secret\_number = 7

attempts = 0

while attempts < 3:

guess = int(input("Угадайте число от 1 до 10: "))

if guess == secret\_number:

print("Угадали!")

break

attempts += 1

else:

print("Вы не угадали за 3 попытки") # Выводится, если все попытки исчерпаны

##### Пример 6: Проверка наличия значения в последовательности

data = ["apple", "banana", "cherry"]

item = "orange"

index = 0

while index < len(data):

if data[index] == item:

print("Элемент найден")

break

index += 1

else:

print("Элемент не найден") # Выполняется, если цикл завершается без break

18: "Бесконечный цикл"

**Описание и объяснение темы:**

* Бесконечный цикл — это цикл, который продолжает выполняться без остановки. В Python такой цикл чаще всего создаётся с помощью while True, где условие всегда остаётся истинным.
* Бесконечные циклы могут быть полезны для задач, которые требуют постоянного выполнения, например, для работы с сервисами реального времени, ожидания события или опроса устройства.
* Чтобы остановить бесконечный цикл, используется инструкция break, которая завершает выполнение цикла при достижении определённого условия.

**Синтаксис и особенности применения:**

* Синтаксис:

while True:

# код, выполняющийся в цикле

* При создании бесконечного цикла важно предусмотреть способ его завершения, чтобы избежать зависания программы.

**Примеры кода:**

##### Пример 1: Бесконечный цикл с завершением по команде пользователя

while True:

user\_input = input("Введите 'exit' для завершения: ")

if user\_input == "exit":

print("Цикл завершен")

break # Остановка цикла при вводе 'exit'

##### Пример 2: Бесконечный цикл для счётчика с условием остановки

count = 0

while True:

print("Счётчик:", count)

count += 1

if count >= 5:

break # Прерывает цикл, когда счётчик достигнет 5

##### Пример 3: Бесконечный цикл для проверки состояния (пример системы мониторинга)

while True:

system\_status = "running" # Например, получение статуса системы

if system\_status != "running":

print("Система остановлена")

break

print("Система работает")

# Здесь могла бы быть пауза, например, time.sleep(5)

##### Пример 4: Игра угадай число с бесконечным циклом

import random

secret\_number = random.randint(1, 10)

while True:

guess = int(input("Угадайте число от 1 до 10: "))

if guess == secret\_number:

print("Вы угадали!")

break

else:

print("Неправильно, попробуйте ещё раз")

##### Пример 5: Цикл для запроса корректного ввода

while True:

try:

number = int(input("Введите целое число: "))

print("Вы ввели:", number)

break # Завершение цикла при корректном вводе

except ValueError:

print("Ошибка: введите именно целое число")

##### Пример 6: Использование бесконечного цикла для имитации работы сервера

while True:

print("Ожидание запроса от клиента...")

# Имитация обработки запроса

request = input("Введите 'stop' для завершения или любой другой запрос: ")

if request == "stop":

print("Сервер завершает работу")

break

else:

print("Запрос обработан")

19: "Ключевое слово pass"

**Описание и объяснение темы:**

* Ключевое слово pass в Python используется как заполнитель, который ничего не выполняет, но синтаксически необходим для поддержания структуры кода. Это позволяет создать структуру для кода, который будет добавлен позднее, или намеренно оставить блок пустым.
* pass применяется в ситуациях, когда требуется пустое тело функции, цикла, условия или класса, и код не должен прерываться ошибками из-за отсутствия инструкций.

**Синтаксис и особенности применения:**

* Синтаксис:

if условие:

pass # Пустое тело условия

* pass может быть использован в любых структурах кода, где Python ожидает наличие кода, но разработчик не хочет пока ничего выполнять.

**Примеры кода:**

##### Пример 1: Использование pass в условии if

number = 10

if number > 0:

pass # Ничего не делает, оставлено для будущего кода

else:

print("Число отрицательное")

##### Пример 2: pass в определении функции

def unfinished\_function():

pass # Заготовка для будущего кода

print("Функция определена, но пока не делает ничего.")

##### Пример 3: Использование pass в цикле

for i in range(5):

if i % 2 == 0:

pass # Пропуск обработки для чётных чисел

else:

print("Нечётное число:", i)

# Выводит: Нечётное число: 1, Нечётное число: 3

##### Пример 4: pass в конструкции try-except

try:

print(10 / 0)

except ZeroDivisionError:

pass # Пропускает обработку ошибки деления на ноль

print("Программа продолжается после pass")

##### Пример 5: pass в классе

class MyClass:

pass # Пустой класс-заготовка

obj = MyClass()

print("Экземпляр класса создан, но класс пока пустой.")

##### Пример 6: Применение pass для создания заготовок в структуре программы

def menu():

pass # Заготовка для основного меню

def settings():

pass # Заготовка для настроек

print("Программа запущена. Функции пока не реализованы.")

20: "Способы замера времени работы программы"

**Описание и объяснение темы:**

* В Python замер времени работы программы позволяет измерить производительность кода и выявить участки, нуждающиеся в оптимизации.
* Основные способы замера времени:
  1. Использование модуля time с функцией time().
  2. Использование модуля timeit, предназначенного для точного измерения времени выполнения небольших участков кода.
  3. Применение встроенного профилировщика cProfile для подробного анализа времени выполнения программы.

**Синтаксис и особенности применения:**

* time.time(): возвращает текущее время в секундах с начала эпохи. Можно замерить разницу между начальным и конечным временем для измерения выполнения.
* timeit.timeit(): позволяет выполнить и замерить время выполнения выражения несколько раз, что даёт среднее время работы.
* cProfile.run(): предоставляет детальную информацию о вызовах функций и времени выполнения для всего кода.

**Примеры кода:**

##### Пример 1: Замер времени выполнения кода с использованием time.time()

import time

start\_time = time.time()

# Код для замера времени выполнения

total = sum(range(1000000))

end\_time = time.time()

print("Время выполнения:", end\_time - start\_time, "секунд")

##### Пример 2: Замер времени функции с timeit

import timeit

def sample\_function():

return sum(range(1000000))

execution\_time = timeit.timeit("sample\_function()", globals=globals(), number=10)

print("Среднее время выполнения:", execution\_time / 10, "секунд")

##### Пример 3: Использование timeit для замера короткого выражения

import timeit

execution\_time = timeit.timeit("x = sum([i for i in range(100)])", number=1000)

print("Время выполнения выражения:", execution\_time, "секунд")

##### Пример 4: Профилирование кода с использованием cProfile

import cProfile

def test\_function():

total = 0

for i in range(100000):

total += i

return total

cProfile.run('test\_function()') # Выводит подробную информацию о времени выполнения

##### Пример 5: Замер времени с использованием perf\_counter из модуля time

import time

start = time.perf\_counter()

# Код для измерения

for \_ in range(1000000):

pass

end = time.perf\_counter()

print("Время выполнения:", end - start, "секунд")

##### Пример 6: Использование контекстного менеджера timeit для замера времени

from timeit import default\_timer as timer

start = timer()

# Код для замера

result = [x\*\*2 for x in range(10000)]

end = timer()

print("Время выполнения:", end - start, "секунд")

21: "Модуль random"

**Описание и объяснение темы:**

* Модуль random в Python предоставляет функции для генерации случайных чисел, выбора случайных элементов из последовательностей и выполнения других задач, связанных со случайностью. Он широко используется в задачах моделирования, тестирования, игр и везде, где требуется случайность.
* Основные функции модуля random:
  + random(): генерирует случайное число с плавающей точкой в диапазоне [0.0, 1.0).
  + randint(a, b): генерирует случайное целое число в диапазоне [a, b].
  + choice(seq): возвращает случайный элемент из последовательности seq.
  + shuffle(seq): перемешивает элементы в последовательности seq на месте.
  + sample(population, k): возвращает список из k уникальных элементов, выбранных случайным образом из population.

**Синтаксис и особенности применения:**

* Модуль random позволяет контролировать генерацию случайных чисел с помощью функции seed(n), которая задаёт начальную точку генерации случайных чисел. Это полезно для воспроизводимости результатов.

**Примеры кода:**

##### Пример 1: Генерация случайного числа с плавающей точкой

import random

print(random.random()) # Генерирует число от 0.0 до 1.0, например, 0.756234

##### Пример 2: Генерация случайного целого числа

print(random.randint(1, 10)) # Генерирует случайное число от 1 до 10 включительно

##### Пример 3: Выбор случайного элемента из списка

colors = ["красный", "зелёный", "синий", "жёлтый"]

print(random.choice(colors)) # Выбирает случайный цвет, например, "зелёный"

##### Пример 4: Перемешивание списка

cards = [1, 2, 3, 4, 5]

random.shuffle(cards)

print(cards) # Выводит перемешанный список, например, [3, 5, 1, 4, 2]

##### Пример 5: Выбор случайной выборки без повторений

numbers = range(1, 50)

lottery\_numbers = random.sample(numbers, 6)

print(lottery\_numbers) # Например, [7, 21, 34, 16, 2, 47]

##### Пример 6: Использование seed() для воспроизводимости случайности

random.seed(42)

print(random.randint(1, 10)) # Генерирует одно и то же число при каждом запуске, например, 7

print(random.randint(1, 10)) # Последовательность остаётся той же, например, 1

22: "Представление вещественных чисел в памяти компьютера"

**Описание и объяснение темы:**

* Вещественные числа, или числа с плавающей точкой, представляются в памяти компьютера по стандарту IEEE 754. Этот стандарт обеспечивает точное представление и операции с вещественными числами, хотя он имеет ограничения по точности из-за дискретного представления чисел.
* В Python тип float используется для представления чисел с плавающей точкой, и каждый float занимает 64 бита в памяти. Это позволяет хранить числа в диапазоне приблизительно от 1.7 × 10^-308 до 1.7 × 10^308 с точностью около 15 десятичных цифр.
* Память для вещественного числа делится на три части:
  + **Знак** (1 бит): определяет, положительное или отрицательное число.
  + **Экспонента** (11 битов для float): указывает, насколько нужно сдвинуть запятую.
  + **Мантисса** (52 бита): хранит значимые цифры числа.

**Синтаксис и особенности применения:**

* Числа с плавающей точкой в Python могут вызывать проблемы с точностью, особенно при выполнении операций сложения и вычитания.
* Для точных вычислений с вещественными числами можно использовать модуль decimal, который позволяет управлять точностью.

**Примеры кода:**

##### Пример 1: Представление числа с плавающей точкой

x = 0.1 + 0.2

print(x) # Ожидается 0.3, но может выводить 0.30000000000000004 из-за ограничений точности

##### Пример 2: Проверка пределов типа float

import sys

print(sys.float\_info) # Выводит максимальное и минимальное значение float

##### Пример 3: Ошибка округления при операциях с float

a = 0.1

b = 0.2

c = a + b

print(c == 0.3) # False, из-за ошибки округления, результат 0.30000000000000004

##### Пример 4: Преодоление ошибки округления с использованием модуля decimal

from decimal import Decimal

a = Decimal('0.1')

b = Decimal('0.2')

c = a + b

print(c) # Точное значение: 0.3

print(c == Decimal('0.3')) # True

##### Пример 5: Округление чисел с плавающей точкой

number = 3.141592653589793

print(round(number, 2)) # Округляет до двух знаков после запятой: 3.14

##### Пример 6: Представление отрицательного вещественного числа

negative\_float = -0.5

print(negative\_float) # Выводит: -0.5

23: "Алгоритм перевода чисел в формат с плавающей запятой"

**Описание и объяснение темы:**

* Алгоритм перевода чисел в формат с плавающей запятой реализован по стандарту IEEE 754, который используется для представления вещественных чисел в памяти компьютера. Этот стандарт делит число на три части: знак, экспоненту и мантиссу.
* Формат с плавающей запятой позволяет представлять числа, как большие, так и маленькие, с высокой степенью точности, но также вводит ограниченные точности, что приводит к ошибкам округления.

**Основные шаги алгоритма перевода числа в формат с плавающей запятой:**

1. **Определение знака числа**: если число положительное, бит знака равен 0; если отрицательное — 1.
2. **Преобразование числа в нормализованный двоичный формат**: мантисса представляется в двоичной системе как число от 1 до 2 (1.xxxxx). Десятичное число переводится в двоичное, и мантисса отделяется от экспоненты.
3. **Определение экспоненты**: настраивается с использованием смещения (в формате float используется 127 для 32-битного числа и 1023 для 64-битного), чтобы избежать отрицательных значений экспоненты.
4. **Формирование итогового бита**: результатом является число, которое хранит знак, экспоненту и мантиссу для хранения в памяти.

**Примеры кода:**

##### Пример 1: Преобразование десятичного числа в двоичное

def float\_to\_binary(num):

# Проверка знака числа

sign = 0 if num >= 0 else 1

num = abs(num)

# Определение целой и дробной частей

integer\_part = int(num)

fractional\_part = num - integer\_part

# Преобразование целой части

binary\_integer = bin(integer\_part).replace("0b", "")

# Преобразование дробной части

binary\_fractional = []

while fractional\_part and len(binary\_fractional) < 23:

fractional\_part \*= 2

bit = int(fractional\_part)

binary\_fractional.append(str(bit))

fractional\_part -= bit

binary\_result = f"{sign} {binary\_integer}.{''.join(binary\_fractional)}"

return binary\_result

print(float\_to\_binary(0.75)) # Пример для числа 0.75

##### Пример 2: Преобразование числа с плавающей точкой в представление IEEE 754

import struct

def float\_to\_ieee754(num):

binary\_representation = struct.unpack('>I', struct.pack('>f', num))[0]

return f"{binary\_representation:032b}"

print(float\_to\_ieee754(0.75)) # Представление IEEE 754 для 0.75

##### Пример 3: Извлечение мантиссы и экспоненты из числа с плавающей точкой

import math

num = 0.15625

mantissa, exponent = math.frexp(num)

print("Мантисса:", mantissa) # 0.625

print("Экспонента:", exponent) # -2 (в двоичной системе)

##### Пример 4: Переход к формату IEEE 754 для числа с отрицательным значением

num = -5.5

binary\_representation = float\_to\_ieee754(num)

print("Представление IEEE 754 для -5.5:", binary\_representation)

##### Пример 5: Вывод информации о знаке, мантиссе и экспоненте для числа

def decompose\_float(num):

sign = 0 if num >= 0 else 1

mantissa, exponent = math.frexp(abs(num))

print("Знак:", sign)

print("Мантисса:", mantissa)

print("Экспонента:", exponent)

decompose\_float(-0.15625)

##### Пример 6: Округление чисел до ближайшего представимого значения

import decimal

# Число с плавающей точкой, требующее округления

num = decimal.Decimal('0.10000000000000000555')

rounded\_num = round(num, 15)

print("Округленное значение:", rounded\_num) # Округленное до 0.1

24: "Формат с фиксированной запятой для вещественных чисел"

**Описание и объяснение темы:**

* Формат с фиксированной запятой — это метод представления вещественных чисел, при котором позиция десятичной запятой остаётся неизменной, независимо от числового значения.
* В отличие от формата с плавающей запятой, формат с фиксированной запятой сохраняет точное количество десятичных разрядов, что может быть полезно для финансовых и бухгалтерских расчётов, где точность и фиксированное количество десятичных знаков играют важную роль.
* В Python для работы с фиксированной точностью обычно используют модуль decimal, который позволяет точно представлять числа с заданным количеством знаков после запятой.

**Синтаксис и особенности применения:**

* Модуль decimal предоставляет класс Decimal, который позволяет управлять количеством знаков после запятой.
* Настройка точности в decimal позволяет работать с вещественными числами, избегая ошибок округления, свойственных стандартному типу float.

**Примеры кода:**

##### Пример 1: Создание числа с фиксированной точностью с помощью decimal.Decimal

from decimal import Decimal

price = Decimal("19.99")

quantity = Decimal("3")

total = price \* quantity

print("Итоговая стоимость:", total) # Выводит: 59.97 с точной фиксированной запятой

##### Пример 2: Настройка количества знаков после запятой с помощью decimal.getcontext()

from decimal import Decimal, getcontext

getcontext().prec = 4 # Установка точности до 4 значащих цифр

result = Decimal("1.2345") + Decimal("2.3456")

print("Результат:", result) # Выводит: 3.580

##### Пример 3: Сравнение точности между float и decimal.Decimal

from decimal import Decimal

float\_value = 0.1 + 0.2

decimal\_value = Decimal("0.1") + Decimal("0.2")

print("Результат с float:", float\_value) # Выводит: 0.30000000000000004

print("Результат с Decimal:", decimal\_value) # Выводит: 0.3

##### Пример 4: Округление числа до заданного количества знаков

from decimal import Decimal, ROUND\_DOWN

value = Decimal("3.14159")

rounded\_value = value.quantize(Decimal("0.01"), rounding=ROUND\_DOWN)

print("Округленное значение:", rounded\_value) # Выводит: 3.14

##### Пример 5: Установка фиксированной точки для финансовых расчётов

from decimal import Decimal

salary = Decimal("1500.50")

tax\_rate = Decimal("0.15")

tax\_amount = salary \* tax\_rate

print("Налог:", tax\_amount.quantize(Decimal("0.01"))) # Округлённый до 0.01

##### Пример 6: Использование decimal в формате с фиксированной запятой в цикле

from decimal import Decimal

total = Decimal("0.00")

for i in range(1, 11):

total += Decimal("0.10")

print("Сумма с фиксированной запятой:", total) # Выводит точное значение 1.00

25: "Понятие модуля в Python"

**Описание и объяснение темы:**

* Модуль в Python — это файл с расширением .py, содержащий функции, классы и переменные, которые можно импортировать и использовать в других программах. Модули позволяют структурировать и организовывать код, делая его более читаемым и переиспользуемым.
* Python включает обширную стандартную библиотеку, содержащую модули для работы с файловой системой, обработкой данных, математическими операциями и многим другим. Пользователь также может создавать свои модули или устанавливать сторонние, расширяя функционал Python.
* Импортировать модуль можно с помощью команды import, а для более точного доступа к конкретным элементам модуля — с помощью from ... import ....

**Синтаксис и особенности применения:**

* Синтаксис для импорта модуля:

import module\_name

* Чтобы импортировать определённую функцию или переменную из модуля, используется:

from module\_name import function\_name

* Для использования псевдонимов применяется ключевое слово as.

**Примеры кода:**

##### Пример 1: Импорт стандартного модуля math и использование его функций

import math

print(math.sqrt(16)) # Выводит квадратный корень из 16: 4.0

print(math.pi) # Выводит значение числа пи: 3.141592653589793

##### Пример 2: Импорт отдельных функций из модуля math

from math import sqrt, pi

print(sqrt(25)) # Выводит: 5.0

print(pi) # Выводит: 3.141592653589793

##### Пример 3: Использование псевдонима для модуля

import math as m

print(m.sin(m.pi / 2)) # Выводит: 1.0, синус 90 градусов

##### Пример 4: Создание и импорт пользовательского модуля

# В файле my\_module.py:

def greet(name):

return f"Привет, {name}!"

# В основном файле:

import my\_module

print(my\_module.greet("Алексей")) # Выводит: Привет, Алексей!

##### Пример 5: Импорт всех элементов из модуля (использовать с осторожностью)

from math import \*

print(cos(0)) # Выводит: 1.0, косинус 0 градусов

print(log(10)) # Выводит логарифм по основанию e от 10

##### Пример 6: Проверка всех доступных функций и атрибутов модуля с помощью dir()

import math

print(dir(math)) # Выводит список всех доступных функций и атрибутов в модуле math

26: "Различные варианты import"

**Описание и объяснение темы:**

* Python предоставляет несколько вариантов для импорта модулей, которые позволяют управлять доступом к функциям, переменным и классам из модулей.
* Основные варианты import:
  1. **Стандартный импорт всего модуля**: позволяет импортировать модуль целиком, после чего его содержимое доступно через имя модуля.
  2. **Избирательный импорт**: позволяет импортировать только необходимые функции, классы или переменные из модуля.
  3. **Импорт с псевдонимом**: используется для сокращения имен или устранения конфликтов имен модулей.
  4. **Импорт всех элементов (\*)**: импортирует все содержимое модуля, делая его доступным напрямую (не рекомендуется для крупных модулей).

**Синтаксис и особенности применения:**

* import module\_name: стандартный импорт всего модуля.
* from module\_name import function\_name: избирательный импорт.
* import module\_name as alias: импорт с псевдонимом.
* from module\_name import \*: импорт всех элементов (не рекомендуется из-за возможности конфликтов имён).

**Примеры кода:**

##### Пример 1: Стандартный импорт всего модуля

import math

print(math.sqrt(16)) # Доступ к sqrt через имя модуля, выводит: 4.0

##### Пример 2: Избирательный импорт конкретных функций

from math import sqrt, pi

print(sqrt(25)) # Доступ к sqrt напрямую, выводит: 5.0

print(pi) # Выводит: 3.141592653589793

##### Пример 3: Импорт с псевдонимом для сокращения имени модуля

import math as m

print(m.sin(m.pi / 2)) # Доступ через псевдоним, выводит: 1.0

##### Пример 4: Импорт всех элементов из модуля (не рекомендуется)

from math import \*

print(sin(pi / 2)) # Выводит: 1.0, доступ к sin без указания модуля

##### Пример 5: Использование псевдонимов для функций

from math import sqrt as square\_root

print(square\_root(49)) # Выводит: 7.0, используем псевдоним для функции sqrt

##### Пример 6: Импорт внутри функции для локального использования

def calculate\_square\_root(x):

from math import sqrt # Локальный импорт только для этой функции

return sqrt(x)

print(calculate\_square\_root(64)) # Выводит: 8.0

27: "Библиотека math"

**Описание и объяснение темы:**

* Библиотека math в Python предоставляет функции для выполнения математических операций, таких как вычисление квадратного корня, тригонометрические вычисления, логарифмы и константы. Эти функции оптимизированы для работы с числами с плавающей точкой и позволяют выполнять различные математические вычисления с высокой точностью.
* Основные функции и константы math:
  + math.sqrt(x): вычисляет квадратный корень числа x.
  + math.pi и math.e: константы для числа π (пи) и e (основание натурального логарифма).
  + Тригонометрические функции: sin, cos, tan и их обратные функции.
  + Логарифмы: log, log10, log2.
  + Функции округления: floor, ceil.

**Синтаксис и особенности применения:**

* import math: импортирует всю библиотеку math.
* Функции и константы из модуля math работают только с числами, поэтому при необходимости другие типы данных нужно преобразовать в float или int.

**Примеры кода:**

##### Пример 1: Вычисление квадратного корня с помощью math.sqrt

import math

print(math.sqrt(16)) # Выводит: 4.0

##### Пример 2: Использование математических констант pi и e

print("Число пи:", math.pi) # Выводит: 3.141592653589793

print("Число e:", math.e) # Выводит: 2.718281828459045

##### Пример 3: Вычисление синуса и косинуса угла (в радианах)

angle = math.pi / 4 # Угол 45 градусов в радианах

print(math.sin(angle)) # Выводит: 0.7071067811865475

print(math.cos(angle)) # Выводит: 0.7071067811865476

##### Пример 4: Логарифмы с основанием e и 10

value = 10

print(math.log(value)) # Натуральный логарифм, выводит: 2.302585092994046

print(math.log10(value)) # Логарифм по основанию 10, выводит: 1.0

##### Пример 5: Округление с использованием floor и ceil

number = 3.7

print(math.floor(number)) # Округление вниз, выводит: 3

print(math.ceil(number)) # Округление вверх, выводит: 4

##### Пример 6: Вычисление факториала числа

print(math.factorial(5)) # 5! = 120

28: "Библиотека decimal"

**Описание и объяснение темы:**

* Библиотека decimal в Python предназначена для работы с числами с фиксированной точностью, что позволяет избежать ошибок округления, характерных для float. Она особенно полезна в финансовых расчётах и в других случаях, когда необходима точность.
* Класс Decimal из этой библиотеки обеспечивает точное представление вещественных чисел и позволяет задавать точность арифметических операций, настраиваемую с помощью контекста.

**Синтаксис и особенности применения:**

* Основные компоненты библиотеки:
  + Decimal(): создаёт число с фиксированной точностью.
  + getcontext().prec: позволяет задать точность.
  + quantize(): используется для округления числа до заданного количества знаков.
* Важно использовать строку при создании объекта Decimal, чтобы избежать ошибок округления, возникающих при преобразовании float.

**Примеры кода:**

##### Пример 1: Создание чисел Decimal и простая операция сложения

from decimal import Decimal

a = Decimal("0.1")

b = Decimal("0.2")

result = a + b

print(result) # Выводит: 0.3, точное значение

##### Пример 2: Настройка точности с помощью getcontext().prec

from decimal import Decimal, getcontext

getcontext().prec = 4 # Установка точности до 4 значащих цифр

result = Decimal("1.2345") + Decimal("2.3456")

print(result) # Выводит: 3.580

##### Пример 3: Округление числа с использованием quantize()

value = Decimal("3.14159")

rounded\_value = value.quantize(Decimal("0.01")) # Округление до двух знаков

print("Округленное значение:", rounded\_value) # Выводит: 3.14

##### Пример 4: Сравнение float и Decimal для точности вычислений

float\_value = 0.1 + 0.2

decimal\_value = Decimal("0.1") + Decimal("0.2")

print("Результат с float:", float\_value) # Выводит: 0.30000000000000004

print("Результат с Decimal:", decimal\_value) # Выводит: 0.3

##### Пример 5: Использование Decimal для финансовых расчётов

price = Decimal("19.99")

quantity = Decimal("3")

total = price \* quantity

print("Итоговая стоимость:", total.quantize(Decimal("0.01"))) # Выводит: 59.97

##### Пример 6: Преобразование строки в Decimal для точного представления

decimal\_value = Decimal("123.456789")

print(decimal\_value) # Выводит точное значение: 123.456789

29: "Запись строк. Типы str, int, float, bool"

**Описание и объяснение темы:**

* В Python строки (str), целые числа (int), числа с плавающей точкой (float) и булевы значения (bool) — это базовые типы данных, которые позволяют работать с текстом, числами и логическими значениями.
* Каждый из этих типов имеет свои уникальные свойства и методы:
  + **str**: представляет текстовые данные. Строки неизменяемы, и каждый символ имеет индекс.
  + **int**: целочисленные значения без дробной части. int может представлять как положительные, так и отрицательные числа.
  + **float**: числа с плавающей точкой, представляющие вещественные значения.
  + **bool**: логические значения, которые могут быть только True или False.

**Синтаксис и особенности применения:**

* Создание объектов этих типов можно выполнить с помощью присваивания или использования встроенных функций str(), int(), float() и bool() для приведения типов.
* Типы int, float и bool поддерживают арифметические и логические операции, в то время как str имеет методы для работы с текстом.

**Примеры кода:**

##### Пример 1: Создание и использование строки (str)

text = "Привет, мир!"

print(text) # Выводит: Привет, мир!

print(text[0]) # Доступ к первому символу, выводит: П

print(len(text)) # Длина строки, выводит: 12

##### Пример 2: Операции с целыми числами (int)

a = 10

b = 3

print(a + b) # Сложение, выводит: 13

print(a \* b) # Умножение, выводит: 30

print(a // b) # Целочисленное деление, выводит: 3

##### Пример 3: Операции с числами с плавающей точкой (float)

x = 5.7

y = 3.2

print(x - y) # Вычитание, выводит: 2.5

print(x / y) # Деление, выводит: 1.78125

##### Пример 4: Булевы значения (bool) и логические операции

is\_active = True

is\_logged\_in = False

print(is\_active and is\_logged\_in) # Логическое И, выводит: False

print(is\_active or is\_logged\_in) # Логическое ИЛИ, выводит: True

print(not is\_active) # Логическое НЕ, выводит: False

##### Пример 5: Приведение типов (str, int, float, bool)

number = "123"

print(int(number) + 5) # Приведение строки к int и сложение, выводит: 128

decimal\_number = "12.34"

print(float(decimal\_number) \* 2) # Приведение строки к float, выводит: 24.68

print(bool(0)) # Преобразование 0 в bool, выводит: False

print(bool("Hello")) # Непустая строка в bool, выводит: True

##### Пример 6: Преобразование числа в строку и объединение с текстом

age = 25

message = "Ваш возраст: " + str(age)

print(message) # Выводит: Ваш возраст: 25

30: "Арифметические операции над строками"

**Описание и объяснение темы:**

* В Python строки не поддерживают арифметические операции в традиционном смысле, как числа, но есть два оператора, которые применимы к строкам:
  1. **Конкатенация (+)**: позволяет объединять две строки.
  2. **Повторение (\*)**: позволяет повторить строку заданное количество раз.
* Конкатенация объединяет строки в одну, тогда как оператор \* создаёт новую строку, повторяя исходную строку несколько раз.

**Синтаксис и особенности применения:**

* Синтаксис для объединения: str1 + str2.
* Синтаксис для повторения: str \* n, где n — целое число.
* Попытка выполнения других арифметических операций с str (например, -, /) вызовет ошибку.

**Примеры кода:**

##### Пример 1: Конкатенация строк с помощью +

greeting = "Hello"

name = "Alice"

message = greeting + ", " + name + "!"

print(message) # Выводит: Hello, Alice!

##### Пример 2: Повторение строки с помощью \*

word = "ha"

laugh = word \* 3

print(laugh) # Выводит: hahaha

##### Пример 3: Объединение чисел и строк через приведение типа

age = 30

message = "Возраст: " + str(age)

print(message) # Выводит: Возраст: 30

##### Пример 4: Использование \* для создания строки из символов

line = "-" \* 10

print(line) # Выводит: ----------

##### Пример 5: Ошибка при использовании неподдерживаемых арифметических операций

text = "Python"

# print(text - "P") # Ошибка: unsupported operand type(s) for -: 'str' and 'str'

##### Пример 6: Комбинирование строки с повторением и конкатенацией

base = "Na"

chorus = base \* 4 + " " + "Batman!"

print(chorus) # Выводит: NaNaNaNa Batman!

31: "Функции len(), ord(), chr()"

**Описание и объяснение темы:**

* Функции len(), ord(), и chr() в Python являются встроенными и используются для работы со строками и символами:
  + **len()**: возвращает количество символов в строке или элементов в последовательности (например, список, кортеж).
  + **ord()**: принимает одиночный символ и возвращает его числовое представление в таблице Unicode.
  + **chr()**: выполняет обратное преобразование, принимая числовой код и возвращая соответствующий символ из таблицы Unicode.

**Синтаксис и особенности применения:**

* Синтаксис:
  + len(obj): возвращает длину строки или количество элементов в объекте obj.
  + ord(char): возвращает числовой код символа char.
  + chr(code): возвращает символ, соответствующий числу code в таблице Unicode.

**Примеры кода:**

##### Пример 1: Использование len() для подсчёта символов в строке

text = "Hello, World!"

print(len(text)) # Выводит: 13

##### Пример 2: Использование ord() для получения числового кода символа

char = "A"

print(ord(char)) # Выводит: 65, код символа "A" в Unicode

##### Пример 3: Использование chr() для получения символа по коду

code = 97

print(chr(code)) # Выводит: "a", символ с кодом 97 в Unicode

##### Пример 4: Преобразование строки в список кодов Unicode

text = "Hi"

codes = [ord(char) for char in text]

print(codes) # Выводит: [72, 105]

##### Пример 5: Преобразование списка кодов Unicode обратно в строку

codes = [72, 105]

text = "".join(chr(code) for code in codes)

print(text) # Выводит: Hi

##### Пример 6: Подсчёт количества символов в пустой строке

empty\_string = ""

print(len(empty\_string)) # Выводит: 0

32: "Сравнение строк"

**Описание и объяснение темы:**

* В Python строки можно сравнивать с использованием операторов ==, !=, <, >, <=, >=. Сравнение строк происходит в лексикографическом порядке, то есть по порядку символов в таблице Unicode.
* Сравнение строк полезно для задач сортировки, проверки совпадений и других операций с текстом. При сравнении строки рассматриваются посимвольно слева направо, и если встречается первое несовпадение, результат сравнения определяется по нему.

**Синтаксис и особенности применения:**

* == проверяет, равны ли строки.
* != проверяет, не равны ли строки.
* < и > проверяют, какая из строк "меньше" или "больше" в алфавитном порядке.
* Строки регистрозависимы: apple и Apple считаются разными.

**Примеры кода:**

##### Пример 1: Проверка равенства строк

str1 = "hello"

str2 = "hello"

print(str1 == str2) # Выводит: True, строки совпадают

##### Пример 2: Сравнение строк с учетом регистра

str1 = "Python"

str2 = "python"

print(str1 == str2) # Выводит: False, регистр букв различается

##### Пример 3: Лексикографическое сравнение строк

str1 = "apple"

str2 = "banana"

print(str1 < str2) # Выводит: True, "apple" идёт перед "banana" по алфавиту

##### Пример 4: Сравнение строк разной длины

str1 = "app"

str2 = "apple"

print(str1 < str2) # Выводит: True, так как "app" короче "apple" и считается меньше

##### Пример 5: Проверка неравенства строк

str1 = "hello"

str2 = "world"

print(str1 != str2) # Выводит: True, строки различны

##### Пример 6: Сравнение строк с одинаковыми символами, но разной длиной

str1 = "abc"

str2 = "abcd"

print(str1 < str2) # Выводит: True, "abc" короче и меньше по порядку

33: "Метод in"

**Описание и объяснение темы:**

* Оператор in в Python используется для проверки наличия подстроки в строке или элемента в коллекции (списке, кортеже, словаре, множестве). В контексте строк in проверяет, содержится ли подстрока в строке, возвращая True или False.
* Оператор in полезен для быстрого выполнения проверок без необходимости использования циклов или сложных условий.

**Синтаксис и особенности применения:**

* Синтаксис: подстрока in строка возвращает True, если подстрока найдена в строка.
* Сочетание not in позволяет проверить, что подстрока отсутствует.

**Примеры кода:**

##### Пример 1: Проверка наличия подстроки в строке

text = "Hello, world!"

print("world" in text) # Выводит: True, так как "world" содержится в строке

##### Пример 2: Проверка отсутствия подстроки

text = "Python programming"

print("Java" not in text) # Выводит: True, так как "Java" отсутствует

##### Пример 3: Условное выполнение кода при наличии подстроки

phrase = "Learning Python is fun"

if "Python" in phrase:

print("Нашли 'Python' в строке!") # Выводит: Нашли 'Python' в строке!

##### Пример 4: Поиск подстроки с учетом регистра

text = "Data Science"

print("data" in text) # Выводит: False, так как регистр букв различается

##### Пример 5: Использование in для проверки символа

word = "apple"

print("a" in word) # Выводит: True, так как "a" есть в "apple"

print("b" in word) # Выводит: False, так как "b" нет в "apple"

##### Пример 6: Использование not in для фильтрации строк

words = ["apple", "banana", "cherry"]

filtered\_words = [word for word in words if "a" not in word]

print(filtered\_words) # Выводит: ['cherry'], так как "a" отсутствует

34: "Доступ к элементу по индексу"

**Описание и объяснение темы:**

* В Python доступ к элементам строки можно получить по индексу. Индекс — это позиция символа в строке, начиная с 0 для первого символа, 1 для второго и так далее.
* Использование индекса позволяет получить конкретный символ строки или изменить его. Также поддерживается доступ к символам с конца строки с помощью отрицательных индексов: -1 указывает на последний символ, -2 — на предпоследний.
* Индексация важна для работы с подстроками, создания циклов и итераций, а также для поиска и замены символов.

**Синтаксис и особенности применения:**

* Синтаксис: строка[индекс] возвращает символ строки по заданному индексу.
* Попытка обратиться к индексу, который выходит за границы строки, вызовет ошибку IndexError.

**Примеры кода:**

##### Пример 1: Доступ к символу по положительному индексу

text = "Hello, World!"

print(text[0]) # Выводит: H, первый символ строки

print(text[7]) # Выводит: W, восьмой символ строки

##### Пример 2: Доступ к символу по отрицательному индексу

text = "Python"

print(text[-1]) # Выводит: n, последний символ строки

print(text[-3]) # Выводит: h, третий символ с конца

##### Пример 3: Извлечение подстроки с использованием индексации

text = "Data Science"

print(text[5:11]) # Выводит: Scien, символы с индекса 5 по 10

##### Пример 4: Использование цикла для доступа к каждому символу

text = "AI"

for i in range(len(text)):

print(f"Символ на позиции {i}: {text[i]}")

# Выводит:

# Символ на позиции 0: A

# Символ на позиции 1: I

##### Пример 5: Проверка наличия символа перед доступом по индексу

text = "Hello"

index = 3

if index < len(text):

print(text[index]) # Выводит: l, если индекс не превышает длину строки

else:

print("Индекс вне диапазона")

##### Пример 6: Ошибка при выходе за пределы диапазона индексов

text = "Python"

# print(text[10]) # IndexError, так как индекс 10 выходит за границы строки

35: "Неизменяемость строк"

**Описание и объяснение темы:**

* В Python строки являются неизменяемыми объектами. Это означает, что после создания строки изменить отдельные символы или подстроки напрямую нельзя.
* Если требуется изменить строку, необходимо создать новую строку с нужными изменениями. Неизменяемость строк позволяет оптимизировать память и избежать непредвиденных изменений данных, к которым могут обращаться несколько частей программы.

**Синтаксис и особенности применения:**

* Попытка присвоить новое значение отдельному символу строки вызовет ошибку TypeError.
* Для изменения строки используются такие методы, как конкатенация (+) или срезы, которые создают новую строку.

**Примеры кода:**

##### Пример 1: Попытка изменить символ строки

text = "Hello"

# text[0] = "Y" # TypeError, так как строки неизменяемы

##### Пример 2: Изменение строки с использованием среза и конкатенации

text = "Hello"

new\_text = "Y" + text[1:] # Замена первого символа

print(new\_text) # Выводит: Yello

##### Пример 3: Создание новой строки с использованием метода replace()

text = "Hello, World!"

new\_text = text.replace("World", "Python")

print(new\_text) # Выводит: Hello, Python!

##### Пример 4: Конкатенация строк для изменения исходной строки

text = "apple"

text = text + "s" # Добавление символа "s" в конце

print(text) # Выводит: apples

##### Пример 5: Обход ограничений неизменяемости с помощью списка

text = "immutable"

text\_list = list(text) # Преобразуем строку в список символов

text\_list[0] = "I" # Изменяем первый символ

new\_text = "".join(text\_list)

print(new\_text) # Выводит: Immutable

##### Пример 6: Проверка неизменяемости путём сравнения идентификаторов строк

text = "hello"

new\_text = text + "!"

print(id(text)) # Идентификатор исходной строки

print(id(new\_text)) # Идентификатор новой строки, они разные

36: "Срез от строки"

**Описание и объяснение темы:**

* Срезы позволяют извлечь часть строки, указав начальный и конечный индексы. Этот метод работает как на отдельных символах, так и на подстроках.
* Срезы полезны, когда нужно получить часть строки, изменить порядок символов или создать новую строку на основе существующей.

**Синтаксис и особенности применения:**

* Синтаксис: строка[начало:конец:шаг], где:
  + начало: начальный индекс (по умолчанию 0).
  + конец: конечный индекс (не включается в результат).
  + шаг: количество символов, которые будут пропущены (по умолчанию 1).
* Если указать отрицательный шаг, строка будет извлечена в обратном порядке.

**Примеры кода:**

##### Пример 1: Извлечение подстроки с использованием начала и конца

text = "Hello, World!"

print(text[0:5]) # Выводит: Hello

##### Пример 2: Срез с использованием шага

text = "abcdef"

print(text[::2]) # Выводит: ace, каждый второй символ

##### Пример 3: Обратный порядок символов с помощью отрицательного шага

text = "Python"

print(text[::-1]) # Выводит: nohtyP

##### Пример 4: Пропуск символов с помощью среза

text = "abcdefgh"

print(text[1:7:2]) # Выводит: bdf, начиная с индекса 1 и до 7 с шагом 2

##### Пример 5: Обрезка строки до определённого символа

text = "Hello, World!"

print(text[:5]) # Выводит: Hello, от начала до индекса 5

##### Пример 6: Извлечение символов с конца строки

text = "abcdef"

print(text[-3:]) # Выводит: def, последние три символа

37: "Копия строки"

**Описание и объяснение темы:**

* В Python строки являются неизменяемыми, и копия строки — это отдельный объект с тем же содержимым, но с другим идентификатором. Поскольку строки неизменяемы, копирование строки можно рассматривать как создание нового объекта с теми же символами.
* Копия строки может понадобиться при необходимости передать строку в функцию или изменить её, создав новый объект, не затрагивая исходную строку.

**Синтаксис и особенности применения:**

* Самый простой способ создать копию строки — использовать срез [:] или присвоить строку другой переменной.
* Поскольку строки неизменяемы, копии можно создать с помощью стандартных операций, при этом новые строки будут иметь разные идентификаторы.

**Примеры кода:**

##### Пример 1: Создание копии строки с использованием среза

text = "Hello"

copy\_text = text[:]

print(copy\_text) # Выводит: Hello

print(id(text), id(copy\_text)) # Разные идентификаторы для исходной и копии

##### Пример 2: Копия строки через прямое присваивание

original = "Python"

duplicate = original

print(duplicate) # Выводит: Python

print(id(original), id(duplicate)) # Идентификаторы совпадают, так как строки неизменяемы

##### Пример 3: Копирование строки с конкатенацией

text = "Data"

copy\_text = "" + text

print(copy\_text) # Выводит: Data

print(id(text), id(copy\_text)) # Идентификаторы совпадают

##### Пример 4: Проверка на совпадение идентификаторов исходной и копированной строки

text = "Immutable"

copy\_text = text[:]

print(text == copy\_text) # True, строки идентичны

print(id(text) == id(copy\_text)) # True, так как строки неизменяемы

##### Пример 5: Копия строки при передаче в функцию

def modify\_string(s):

s = s + " modified"

return s

original = "Test"

new\_string = modify\_string(original)

print(original) # Выводит: Test, исходная строка не изменилась

print(new\_string) # Выводит: Test modified

##### Пример 6: Проверка уникальности объекта при копировании пустой строки

empty\_string = ""

copy\_empty = empty\_string[:]

print(id(empty\_string) == id(copy\_empty)) # True, так как обе пустые строки указывают на один объект

38: "Поиск подстроки в строке"

**Описание и объяснение темы:**

* В Python поиск подстроки в строке можно выполнить с помощью методов find(), index(), in, startswith() и endswith(). Эти методы позволяют определить, присутствует ли подстрока в строке, и, при необходимости, узнать её начальный индекс.
* Методы поиска полезны для задач анализа текста, где требуется выявить наличие определённых слов или символов в строках.

**Синтаксис и особенности применения:**

* str.find(substring): возвращает индекс начала первого вхождения подстроки или -1, если подстрока не найдена.
* str.index(substring): аналогичен find(), но вызывает ошибку ValueError, если подстрока не найдена.
* in: проверяет, находится ли подстрока в строке, и возвращает True или False.
* str.startswith(substring) и str.endswith(substring): проверяют, начинается или заканчивается ли строка определённой подстрокой.

**Примеры кода:**

##### Пример 1: Поиск подстроки с использованием find()

text = "Hello, world!"

index = text.find("world")

print(index) # Выводит: 7, позиция начала подстроки "world"

##### Пример 2: Поиск подстроки с использованием index()

text = "Hello, world!"

try:

index = text.index("Python")

except ValueError:

print("Подстрока не найдена") # Выводит: Подстрока не найдена

##### Пример 3: Проверка наличия подстроки с помощью in

text = "Data science is interesting"

print("science" in text) # Выводит: True

print("math" in text) # Выводит: False

##### Пример 4: Использование startswith() для проверки начала строки

text = "Python programming"

print(text.startswith("Python")) # Выводит: True

print(text.startswith("Java")) # Выводит: False

##### Пример 5: Использование endswith() для проверки окончания строки

filename = "document.txt"

print(filename.endswith(".txt")) # Выводит: True

print(filename.endswith(".pdf")) # Выводит: False

##### Пример 6: Поиск всех вхождений подстроки

text = "banana"

positions = []

start = 0

while (index := text.find("a", start)) != -1:

positions.append(index)

start = index + 1

print(positions) # Выводит: [1, 3, 5], все позиции символа "a"

39: "Проверка предиката"

**Описание и объяснение темы:**

* В Python предикат — это функция или выражение, возвращающее логическое значение True или False. Предикаты используются для проверки условий, фильтрации данных и выполнения различных операций, зависящих от истинности или ложности условия.
* Проверка предиката удобна для сокращённого написания условий в циклах и конструкциях if, а также для создания фильтров с использованием функций all(), any(), filter().

**Синтаксис и особенности применения:**

* all(iterable): возвращает True, если все элементы в iterable являются истинными.
* any(iterable): возвращает True, если хотя бы один элемент в iterable является истинным.
* filter(function, iterable): возвращает итератор с элементами, для которых function возвращает True.

**Примеры кода:**

##### Пример 1: Проверка всех значений с помощью all()

numbers = [2, 4, 6, 8]

print(all(num % 2 == 0 for num in numbers)) # Выводит: True, все числа чётные

##### Пример 2: Проверка наличия хотя бы одного положительного значения с any()

values = [-1, 0, -3, 4]

print(any(value > 0 for value in values)) # Выводит: True, так как 4 > 0

##### Пример 3: Использование filter() для фильтрации положительных чисел

numbers = [-5, 3, -1, 7, -2]

positive\_numbers = list(filter(lambda x: x > 0, numbers))

print(positive\_numbers) # Выводит: [3, 7]

##### Пример 4: Проверка предиката внутри if

text = "hello123"

if any(char.isdigit() for char in text):

print("Текст содержит цифры") # Выводит: Текст содержит цифры

##### Пример 5: Использование all() для проверки выполнения условия

values = [10, 20, 30]

print(all(x > 5 for x in values)) # Выводит: True, все элементы больше 5

##### Пример 6: Фильтрация списка на основе предиката

words = ["apple", "banana", "cherry", "avocado"]

words\_with\_a = list(filter(lambda x: 'a' in x, words))

print(words\_with\_a) # Выводит: ['apple', 'banana', 'avocado']

40: "Проверка условия относительно строки"

**Описание и объяснение темы:**

* В Python строки можно проверять на выполнение различных условий, например, является ли строка числом, состоит ли она только из букв или только из цифр. Такие проверки выполняются с помощью встроенных методов строковых объектов.
* Основные методы для проверки условий:
  + str.isalpha(): проверяет, состоит ли строка только из букв.
  + str.isdigit(): проверяет, состоит ли строка только из цифр.
  + str.isalnum(): проверяет, состоит ли строка только из букв и цифр.
  + str.isspace(): проверяет, состоит ли строка только из пробелов.
  + str.islower(): проверяет, все ли буквы в строке в нижнем регистре.
  + str.isupper(): проверяет, все ли буквы в строке в верхнем регистре.

**Синтаксис и особенности применения:**

* Применяются как str.method(), где str — это строка, к которой применяется метод.
* Эти методы возвращают логическое значение True или False, в зависимости от выполнения условия.

**Примеры кода:**

##### Пример 1: Проверка, состоит ли строка только из букв с isalpha()

text = "HelloWorld"

print(text.isalpha()) # Выводит: True, так как строка состоит только из букв

##### Пример 2: Проверка, состоит ли строка только из цифр с isdigit()

number = "123456"

print(number.isdigit()) # Выводит: True, так как строка состоит только из цифр

##### Пример 3: Проверка, состоит ли строка только из букв и цифр с isalnum()

text = "Python3"

print(text.isalnum()) # Выводит: True, так как строка состоит из букв и цифр

##### Пример 4: Проверка на наличие только пробелов с isspace()

space\_str = " "

print(space\_str.isspace()) # Выводит: True, так как строка состоит только из пробелов

##### Пример 5: Проверка на нижний и верхний регистры

text\_lower = "python"

text\_upper = "PYTHON"

print(text\_lower.islower()) # Выводит: True

print(text\_upper.isupper()) # Выводит: True

##### Пример 6: Комбинированная проверка строки

password = "Pass123"

if password.isalnum() and any(char.isdigit() for char in password):

print("Пароль допустим") # Выводит: Пароль допустим, строка содержит буквы и цифры

41: "Методы, изменяющие строку"

**Описание и объяснение темы:**

* В Python строки являются неизменяемыми объектами, поэтому методы, которые как бы "изменяют" строку, на самом деле создают и возвращают её новую версию с учётом внесённых изменений. Эти методы включают replace(), upper(), lower(), strip(), capitalize(), и другие.
* Методы изменения строк позволяют выполнять манипуляции, такие как удаление пробелов, замена символов, изменение регистра и форматирование.

**Синтаксис и особенности применения:**

* Синтаксис для вызова: str.method(), где str — исходная строка, и method() — вызываемый метод.
* Поскольку строки неизменяемы, исходная строка остаётся без изменений, а метод возвращает новую строку.

**Примеры кода:**

##### Пример 1: Замена символов с replace()

text = "Hello, World!"

new\_text = text.replace("World", "Python")

print(new\_text) # Выводит: Hello, Python!

##### Пример 2: Приведение к верхнему и нижнему регистру

text = "Python"

print(text.upper()) # Выводит: PYTHON

print(text.lower()) # Выводит: python

##### Пример 3: Удаление пробелов с помощью strip()

text = " Hello "

print(text.strip()) # Выводит: Hello, удаляются пробелы с обеих сторон

print(text.lstrip()) # Выводит: Hello , удаляются пробелы слева

print(text.rstrip()) # Выводит: Hello, удаляются пробелы справа

##### Пример 4: Преобразование строки с помощью capitalize()

text = "python programming"

print(text.capitalize()) # Выводит: Python programming, первая буква становится заглавной

##### Пример 5: Центрирование строки с center()

text = "Hello"

centered\_text = text.center(10, "\*")

print(centered\_text) # Выводит: \*\*Hello\*\*\*

##### Пример 6: Замена нескольких символов в строке

text = "apple, banana, cherry"

new\_text = text.replace(",", ";")

print(new\_text) # Выводит: apple; banana; cherry

42: "Разделение строки на подстроки"

**Описание и объяснение темы:**

* В Python метод split() позволяет разделить строку на подстроки по определённому разделителю и возвращает список с результатами разделения.
* split() полезен для обработки текстов, где строки содержат данные, разделённые пробелами, запятыми или другими символами (например, CSV-файлы).

**Синтаксис и особенности применения:**

* Синтаксис: str.split(separator, maxsplit), где:
  + separator — разделитель, по умолчанию пробел.
  + maxsplit — максимальное количество разделений; если не указано, строка будет разделена полностью.
* Метод rsplit() выполняет ту же операцию, но начинает с конца строки.

**Примеры кода:**

##### Пример 1: Разделение строки по пробелам (по умолчанию)

text = "apple banana cherry"

words = text.split()

print(words) # Выводит: ['apple', 'banana', 'cherry']

##### Пример 2: Разделение строки по запятой

text = "apple,banana,cherry"

fruits = text.split(",")

print(fruits) # Выводит: ['apple', 'banana', 'cherry']

##### Пример 3: Использование параметра maxsplit для ограничения количества разделений

text = "one, two, three, four"

limited\_split = text.split(", ", 2)

print(limited\_split) # Выводит: ['one', 'two', 'three, four']

##### Пример 4: Разделение строки с пробелами в начале и конце

text = " hello world "

words = text.split()

print(words) # Выводит: ['hello', 'world']

##### Пример 5: Использование rsplit() для разделения строки с конца

text = "apple,banana,cherry"

last\_two = text.rsplit(",", 1)

print(last\_two) # Выводит: ['apple,banana', 'cherry']

##### Пример 6: Преобразование строки в список слов с использованием split()

sentence = "This is a test sentence"

words = sentence.split()

print(words) # Выводит: ['This', 'is', 'a', 'test', 'sentence']

43: "Выравнивание строк"

**Описание и объяснение темы:**

* В Python строки можно выравнивать относительно ширины, используя методы ljust(), rjust() и center(). Эти методы позволяют добавлять пробелы или другие символы слева, справа или по обеим сторонам строки, чтобы она соответствовала указанной ширине.
* Выравнивание строк удобно при создании отчетов, таблиц и форматировании вывода для повышения читаемости.

**Синтаксис и особенности применения:**

* str.ljust(width, fillchar): выравнивает строку по левому краю, заполняя свободное пространство символом fillchar (по умолчанию — пробел).
* str.rjust(width, fillchar): выравнивает строку по правому краю.
* str.center(width, fillchar): центрирует строку.
* Все методы возвращают новую строку, исходная строка остаётся неизменной.

**Примеры кода:**

##### Пример 1: Левое выравнивание строки с ljust()

text = "Apple"

aligned\_text = text.ljust(10)

print(aligned\_text) # Выводит: 'Apple '

##### Пример 2: Правое выравнивание строки с rjust()

text = "Banana"

aligned\_text = text.rjust(10)

print(aligned\_text) # Выводит: ' Banana'

##### Пример 3: Центрирование строки с center()

text = "Cherry"

aligned\_text = text.center(10)

print(aligned\_text) # Выводит: ' Cherry '

##### Пример 4: Использование символа \* для заполнения свободного пространства

text = "Date"

aligned\_text = text.center(10, "\*")

print(aligned\_text) # Выводит: '\*\*\*Date\*\*\*'

##### Пример 5: Левое выравнивание с заполнением символом -

text = "Orange"

aligned\_text = text.ljust(12, "-")

print(aligned\_text) # Выводит: 'Orange------'

##### Пример 6: Пример табличного выравнивания текста

header = "Product"

price = "Price"

print(header.ljust(15) + price.rjust(10))

print("Apple".ljust(15) + "$1.50".rjust(10))

print("Banana".ljust(15) + "$0.80".rjust(10))

44: "Удаление ненужных символов из строки"

**Описание и объяснение темы:**

* В Python удаление ненужных символов в начале, конце или по краям строки можно выполнить с помощью методов strip(), lstrip() и rstrip(). Эти методы часто используются для очистки строк от пробелов, переносов строки, специальных символов или лишних символов, таких как запятые или точки.
* Основные методы:
  + strip(): удаляет символы с обоих концов строки.
  + lstrip(): удаляет символы только с левой стороны строки.
  + rstrip(): удаляет символы только с правой стороны строки.

**Синтаксис и особенности применения:**

* Синтаксис: str.strip(chars), где chars — строка символов для удаления (по умолчанию удаляются пробелы).
* Все методы возвращают новую строку, исходная строка остаётся неизменной.

**Примеры кода:**

##### Пример 1: Удаление пробелов по краям строки с помощью strip()

text = " Hello, World! "

cleaned\_text = text.strip()

print(cleaned\_text) # Выводит: 'Hello, World!'

##### Пример 2: Удаление символов в начале строки с помощью lstrip()

text = "---Python"

cleaned\_text = text.lstrip("-")

print(cleaned\_text) # Выводит: 'Python'

##### Пример 3: Удаление символов в конце строки с помощью rstrip()

text = "Data Science!!"

cleaned\_text = text.rstrip("!")

print(cleaned\_text) # Выводит: 'Data Science'

##### Пример 4: Удаление нескольких символов с помощью strip()

text = "#@!Python Programming@!#"

cleaned\_text = text.strip("#@!")

print(cleaned\_text) # Выводит: 'Python Programming'

##### Пример 5: Удаление переносов строки и пробелов

text = "\n\n Hello, World! \n"

cleaned\_text = text.strip()

print(cleaned\_text) # Выводит: 'Hello, World!'

##### Пример 6: Удаление лишних символов в процессе обработки данных

data = [" apple ", " banana ", "cherry "]

cleaned\_data = [item.strip() for item in data]

print(cleaned\_data) # Выводит: ['apple', 'banana', 'cherry']

45: "Строковые литералы"

**Описание и объяснение темы:**

* Строковые литералы — это текстовые данные, заключённые в кавычки ('...' или "...") и представляющие строковые значения. Python поддерживает различные типы строковых литералов для удобного представления текстовых данных:
  + Обычные строки: записываются в одинарных ('...') или двойных ("...") кавычках.
  + Многострочные строки: заключены в тройные кавычки ('''...''' или """..."""), позволяют записывать текст в несколько строк.
  + Сырые строки: записываются с префиксом r или R перед кавычками (r"..."), чтобы избежать экранирования.
  + Юникодные строки: поддерживают символы Unicode для различных языков и символов, используются в обычных строках, где можно вставлять коды Unicode (\uXXXX).

**Синтаксис и особенности применения:**

* Строки можно создать с помощью одинарных, двойных или тройных кавычек, в зависимости от потребностей. Для многострочных данных тройные кавычки предпочтительны, так как они поддерживают переносы строки.

**Примеры кода:**

##### Пример 1: Обычные строки в одинарных и двойных кавычках

text1 = 'Hello, World!'

text2 = "Hello, World!"

print(text1) # Выводит: Hello, World!

print(text2) # Выводит: Hello, World!

##### Пример 2: Многострочные строки с тройными кавычками

text = """Это многострочная

строка, которая занимает

несколько строк."""

print(text)

# Выводит:

# Это многострочная

# строка, которая занимает

# несколько строк.

##### Пример 3: Сырые строки для использования специальных символов без экранирования

path = r"C:\new\_folder\file.txt"

print(path) # Выводит: C:\new\_folder\file.txt

##### Пример 4: Использование символов Unicode

text = "Привет, мир! \u263A" # \u263A — это код символа ☺

print(text) # Выводит: Привет, мир! ☺

##### Пример 5: Многострочная строка с одинарными тройными кавычками

text = '''Эта строка

тоже многострочная и

записана в одинарных кавычках.'''

print(text)

# Выводит:

# Эта строка

# тоже многострочная и

# записана в одинарных кавычках.

##### Пример 6: Строки с кавычками внутри текста

quote = 'She said, "Hello!"'

print(quote) # Выводит: She said, "Hello!"

quote2 = "It's a beautiful day!"

print(quote2) # Выводит: It's a beautiful day!

46: "Метод format()"

**Описание и объяснение темы:**

* Метод format() используется для форматирования строк и позволяет подставлять значения в текст, используя специальные маркеры {}. Этот метод делает строки более читаемыми и гибкими, особенно при работе с параметрами, которые нужно вставить в текст.
* С помощью format() можно управлять порядком вставляемых значений, задавать выравнивание и формат чисел, а также именовать параметры.

**Синтаксис и особенности применения:**

* Синтаксис: "строка с {} для вставки".format(значение1, значение2).
* Маркеры {} могут содержать позиционные аргументы, именованные аргументы и параметры форматирования.
* Поддерживаются выравнивание, округление, форматирование чисел и дат.

**Примеры кода:**

##### Пример 1: Простое форматирование с позиционными аргументами

text = "Привет, {}! Сегодня {}.".format("Алексей", "понедельник")

print(text) # Выводит: Привет, Алексей! Сегодня понедельник.

##### Пример 2: Именованные аргументы в format()

text = "Имя: {name}, возраст: {age}".format(name="Мария", age=30)

print(text) # Выводит: Имя: Мария, возраст: 30

##### Пример 3: Форматирование с указанием индексов для позиционных аргументов

text = "Число {0} в квадрате равно {1}, а его корень равен {2}".format(4, 16, 2)

print(text) # Выводит: Число 4 в квадрате равно 16, а его корень равен 2

##### Пример 4: Выравнивание текста с помощью format()

text = "|{:<10}|{:^10}|{:>10}|".format("лево", "центр", "право")

print(text) # Выводит: |лево | центр | право|

##### Пример 5: Форматирование чисел с плавающей точкой

value = 3.14159

formatted\_value = "Число пи приблизительно равно {:.2f}".format(value)

print(formatted\_value) # Выводит: Число пи приблизительно равно 3.14

##### Пример 6: Использование format() для форматирования процентов

accuracy = 0.9345

formatted\_accuracy = "Точность: {:.1%}".format(accuracy)

print(formatted\_accuracy) # Выводит: Точность: 93.4%

47: "C-style форматирование с помощью спецификаторов"

**Описание и объяснение темы:**

* C-style форматирование в Python основано на использовании оператора % и спецификаторов формата, подобных тем, которые применяются в языке C. Этот способ форматирования считается устаревшим, но всё ещё используется для простых задач.
* Основные спецификаторы:
  + %s: строка.
  + %d: целое число.
  + %f: число с плавающей точкой.
  + %x / %X: шестнадцатеричное представление числа.
* С помощью %.nf можно задавать точность для чисел с плавающей точкой, где n — число знаков после запятой.

**Синтаксис и особенности применения:**

* Синтаксис: "шаблон % спецификатор" % (значения).
* Для нескольких значений требуется передать кортеж.

**Примеры кода:**

##### Пример 1: Форматирование строки с использованием %s

name = "Анна"

text = "Привет, %s!" % name

print(text) # Выводит: Привет, Анна!

##### Пример 2: Форматирование целого числа с %d

age = 25

text = "Мне %d лет." % age

print(text) # Выводит: Мне 25 лет.

##### Пример 3: Форматирование числа с плавающей точкой с указанием точности

pi = 3.14159

text = "Число пи: %.2f" % pi

print(text) # Выводит: Число пи: 3.14

##### Пример 4: Форматирование нескольких значений с использованием кортежа

name = "Иван"

age = 30

text = "Имя: %s, Возраст: %d" % (name, age)

print(text) # Выводит: Имя: Иван, Возраст: 30

##### Пример 5: Форматирование в шестнадцатеричном формате

number = 255

text = "Число %d в шестнадцатеричном формате: %x" % (number, number)

print(text) # Выводит: Число 255 в шестнадцатеричном формате: ff

##### Пример 6: Выравнивание и точность для чисел с плавающей точкой

value = 5.6789

text = "Значение: %10.2f" % value

print(text) # Выводит: Значение: 5.68 (выравнивание на 10 символов)

48: "f-строки"

**Описание и объяснение темы:**

* f-строки (форматированные строки) — это метод форматирования строк, введённый в Python 3.6. f-строки позволяют вставлять значения переменных прямо в строку, обрамляя их фигурными скобками {} и добавляя префикс f перед строкой.
* f-строки поддерживают выражения, форматирование чисел, округление и выравнивание, что делает их более гибкими и удобными по сравнению с другими методами форматирования.

**Синтаксис и особенности применения:**

* Синтаксис: f"текст {переменная} текст".
* Внутри {} можно использовать выражения и вызовы функций.
* Для форматирования чисел можно использовать :, например, :.2f для указания точности с двумя знаками после запятой.

**Примеры кода:**

##### Пример 1: Простая вставка переменной в f-строку

name = "Ольга"

text = f"Здравствуйте, {name}!"

print(text) # Выводит: Здравствуйте, Ольга!

##### Пример 2: Вставка числового значения с форматированием

pi = 3.14159

text = f"Число пи: {pi:.2f}"

print(text) # Выводит: Число пи: 3.14

##### Пример 3: Встроенные выражения в f-строке

a = 5

b = 10

text = f"Сумма {a} и {b} равна {a + b}"

print(text) # Выводит: Сумма 5 и 10 равна 15

##### Пример 4: Использование функций внутри f-строки

name = "анна"

text = f"Имя с заглавной буквы: {name.capitalize()}"

print(text) # Выводит: Имя с заглавной буквы: Анна

##### Пример 5: Выравнивание и добавление символов в f-строках

value = 42

text = f"Число: {value:>10}" # Выравнивание вправо на 10 символов

print(text) # Выводит: Число: 42

##### Пример 6: Форматирование процентов в f-строках

accuracy = 0.875

text = f"Точность: {accuracy:.1%}"

print(text) # Выводит: Точность: 87.5%

49: "Работа с файлами"

**Описание и объяснение темы:**

* В Python работа с файлами осуществляется с помощью встроенных функций open(), read(), write(), close(), а также конструкции with, которая автоматически закрывает файл после работы с ним.
* Файлы можно открывать в различных режимах:
  + r — чтение (файл должен существовать).
  + w — запись (создаёт новый файл или перезаписывает существующий).
  + a — добавление (добавляет данные в конец файла).
  + rb, wb — чтение и запись в двоичном режиме (используется для файлов, отличных от текста).

**Синтаксис и особенности применения:**

* open("filename", "mode") возвращает объект файла, который можно использовать для чтения или записи.
* Конструкция with open(...) as file автоматически закрывает файл после завершения блока, что предотвращает утечки памяти.

**Примеры кода:**

##### Пример 1: Чтение файла построчно

with open("example.txt", "r") as file:

for line in file:

print(line.strip()) # Читает и выводит строки без переносов

##### Пример 2: Запись в файл

with open("output.txt", "w") as file:

file.write("Это строка записана в файл.\n")

file.write("Ещё одна строка.")

# Файл output.txt теперь содержит две строки

##### Пример 3: Добавление текста в существующий файл

with open("output.txt", "a") as file:

file.write("\nНовая строка добавлена.")

# Текст добавляется в конец файла

##### Пример 4: Чтение всего содержимого файла с помощью read()

with open("example.txt", "r") as file:

content = file.read()

print(content) # Выводит весь текст файла

##### Пример 5: Чтение файла в двоичном режиме

with open("image.png", "rb") as file:

binary\_data = file.read()

print(binary\_data[:10]) # Выводит первые 10 байт изображения

##### Пример 6: Запись списка строк в файл с помощью writelines()

lines = ["Первая строка\n", "Вторая строка\n", "Третья строка\n"]

with open("output.txt", "w") as file:

file.writelines(lines)

# В файл записаны три строки из списка

50: "Инструкции, With .. as"

**Описание и объяснение темы:**

* Конструкция with .. as в Python используется для автоматического управления ресурсами, такими как файлы, сетевые соединения, блокировки и др. Эта конструкция гарантирует, что ресурс будет корректно освобождён после завершения работы с ним, даже если возникнут исключения.
* Наиболее часто with .. as применяется для работы с файлами, обеспечивая их автоматическое закрытие.

**Синтаксис и особенности применения:**

* Синтаксис:

with выражение as имя\_переменной:

# блок кода, в котором используется имя\_переменной

* При выходе из блока with метод \_\_exit\_\_ освобождает ресурс, даже если в блоке возникло исключение.

**Примеры кода:**

##### Пример 1: Чтение файла с использованием with

with open("example.txt", "r") as file:

content = file.read()

print(content) # Выводит содержимое файла

# Файл автоматически закрывается после выхода из блока with

##### Пример 2: Запись в файл с with

with open("output.txt", "w") as file:

file.write("Эта строка записана в файл.")

# Файл закрыт после завершения блока with

##### Пример 3: Использование with .. as для работы с несколькими файлами

with open("input.txt", "r") as infile, open("output.txt", "w") as outfile:

for line in infile:

outfile.write(line)

# Файлы автоматически закрываются после окончания работы

##### Пример 4: Обработка исключений внутри with .. as

try:

with open("nonexistent.txt", "r") as file:

content = file.read()

except FileNotFoundError:

print("Файл не найден.")

# Если файл не существует, выводится сообщение об ошибке

##### Пример 5: Использование with для блокировки с threading.Lock

import threading

lock = threading.Lock()

with lock:

# код, выполняемый в безопасной блокировке

print("Код выполняется в защищённом блоке")

# Блокировка освобождается после завершения блока

##### Пример 6: Использование with .. as для временного изменения конфигурации

from decimal import localcontext, Decimal

with localcontext() as ctx:

ctx.prec = 4 # Установка точности в пределах блока

result = Decimal("1.234567") + Decimal("1.234567")

print(result) # Выводит: 2.469

# После блока точность возвращается к исходному значению