**Методичка по основам Python.**

**1. Типы данных**

**Числа**

В Python есть два основных типа чисел: целые (int) и вещественные (float).

a = 5 # int

b = 3.14 # float

Примеры операций:

sum = a + b # Сложение

difference = a - b # Вычитание

product = a \* b # Умножение

division = a / b # Деление

div = a // b # Нахождение целой части

mod = a % b # Нахождение остатка от деления

**Строки**

Строки в Python заключаются в кавычки: одинарные или двойные.

text = "Hello, World!"

Операции со строками:

# Конкатенация (склеивание)

greeting = "Hello" + " " + "World" # "Hello World"

# Длина строки

length = len(text) # 13

**Списки**

Список — это коллекция элементов, которые можно изменять.

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

Операции со списками:

# Доступ по индексу

print(fruits[0]) # "apple"

# Добавление элемента

fruits.append("orange")

# Удаление элемента

fruits.remove("banana")

**2. Синтаксис и отступы**

В Python отступы играют важную роль. Они определяют блоки кода. Обычно используется 4 пробела на отступ.

if a > b:

print("a больше, чем b")

else:

print("a не больше, чем b")

**3. Линейные программы**

Линейные программы — это простые последовательные операции.

Пример:

x = 5

y = 10

z = x + y

print("Сумма:", z)

**4. Операции со строками**

Строки можно разбивать на части, находить подстроки и изменять регистр:

text = "Hello, World!"

# Преобразование в верхний и нижний регистр

print(text.upper()) # "HELLO, WORLD!"

print(text.lower()) # "hello, world!"

# Проверка на вхождение подстроки

if "World" in text:

print("Слово 'World' найдено!")

**Операции со строками: Замена подстроки (replace)**

Метод replace позволяет заменить в строке все вхождения одной подстроки на другую. Этот метод не изменяет исходную строку, а возвращает новую с произведенными заменами.

**Синтаксис**

новая\_строка = исходная\_строка.replace("что\_заменить", "на\_что\_заменить")

**Пример 1: Простая замена**

Заменим одно слово на другое.

text = "Hello, World!"

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

print(new\_text) # "Hello, Python!"

**Пример 2: Замена символов**

Можно использовать replace для замены символов, например, для удаления пробелов.

text = "1 2 3 4 5"

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

print(new\_text) # "12345"

**Пример 3: Замена с указанием количества замен**

Метод replace позволяет ограничить количество замен, указав дополнительный аргумент.

text = "apple, apple, apple"

new\_text = text.replace("apple", "orange", 1)

print(new\_text) # "orange, apple, apple"

Метод replace — полезный инструмент для работы с текстом, позволяющий легко и быстро заменить фрагменты строки.

**5. Операции с массивами (списками)**

Работа со списками — это важная часть программирования:

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

# Сумма всех элементов

print(sum(numbers))

# Среднее значение

average = sum(numbers) / len(numbers)

print("Среднее значение:", average)

# Сортировка

numbers.sort()

print("Отсортированные числа:", numbers)

**6. Условия (if, else)**

Условия позволяют выполнять код в зависимости от определенных условий:

x = 10

y = 20

if x > y:

print("x больше, чем y")

else:

print("x меньше или равно y")

**Пример 1: Проверка на четность**

Проверим, является ли число четным или нечетным.

number = 7

if number % 2 == 0:

print("Число четное")

else:

print("Число нечетное")

**Пример 2: Категории по возрасту**

Определим категорию человека в зависимости от его возраста.

age = 15

if age < 13:

print("Ребенок")

elif 13 <= age < 18:

print("Подросток")

else:

print("Взрослый")

**Пример 3: Максимум из трех чисел**

Найдем самое большое число из трех заданных.

a = 5

b = 10

c = 3

if a > b and a > c:

print("a - наибольшее число")

elif b > a and b > c:

print("b - наибольшее число")

else:

print("c - наибольшее число")

**Пример 4: Проверка на положительное, отрицательное или ноль**

Определим, является ли число положительным, отрицательным или равно нулю.

number = -5

if number > 0:

print("Число положительное")

elif number < 0:

print("Число отрицательное")

else:

print("Число равно нулю")

**7. Циклы for и while**

**Цикл for**

Этот цикл полезен для повторения операций по коллекции:

for fruit in fruits:

print(fruit)

**Цикл while**

Цикл while работает, пока выполняется условие.

count = 5

while count > 0:

print("Осталось:", count)

count -= 1

**Пример 1: Сумма чисел от 1 до n (цикл for)**

Выведем сумму всех чисел от 1 до n.

n = 10

total = 0

for i in range(1, n + 1):

total += i

print("Сумма чисел от 1 до", n, "равна", total)

**Пример 2: Факториал числа (цикл for)**

Найдем факториал числа n.

n = 5

factorial = 1

for i in range(1, n + 1):

factorial \*= i

print("Факториал числа", n, "равен", factorial)

**Пример 3: Четные числа от 1 до 20 (цикл while)**

Выведем все четные числа от 1 до 20.

i = 1

while i <= 20:

if i % 2 == 0:

print(i)

i += 1

**Пример 4: Таблица умножения (вложенный цикл for)**

Составим таблицу умножения для чисел от 1 до 5.

for i in range(1, 6):

for j in range(1, 6):

print(i, "\*", j, "=", i \* j)

print("------")

**Пример 5: Обратный отсчет (цикл while)**

Запустим обратный отсчет от 10 до 1.

count = 10

while count > 0:

print("Обратный отсчет:", count)

count -= 1

print("Время вышло!")

**Пример 6: Найти первое число, кратное 7 (цикл while)**

Ищем первое число, которое делится на 7, начиная с определенного значения.

num = 10

while num % 7 != 0:

num += 1

print("Первое число, кратное 7:", num)

**Пример использования break и continue**

**Пример 8: Прерывание цикла (break)**

Остановим цикл при нахождении заданного числа.

for i in range(1, 11):

if i == 5:

print("Найдено число 5, прерывание цикла.")

break

print(i)

**Пример 9: Пропуск итерации (continue)**

Пропустим итерацию, если число делится на 3.

for i in range(1, 11):

if i % 3 == 0:

continue

print(i)

**8. Функции (def)**

Функции позволяют объединять несколько операций в один блок кода, который можно вызывать несколько раз. Это полезно для структурирования кода и упрощения его чтения.

**Синтаксис функции**

def имя\_функции(аргументы):

# тело функции

return значение

**Пример 1: Простая функция без параметров**

Создадим функцию, которая просто печатает приветствие.

def say\_hello():

print("Привет, мир!")

# Вызов функции

say\_hello()

**Пример 2: Функция с параметром**

Функция, которая принимает один параметр и выводит его на экран.

def greet(name):

print("Привет,", name)

# Вызов функции

greet("Камиль")

**Пример 3: Функция с возвратом значения**

Функция, которая принимает два числа и возвращает их сумму.

def add(a, b):

return a + b

# Вызов функции

result = add(5, 3)

print("Сумма:", result)

**Пример 4: Функция с несколькими параметрами и логикой внутри**

Функция, которая определяет, является ли число четным или нечетным.

def is\_even(number):

if number % 2 == 0:

return True

else:

return False

# Вызов функции

print(is\_even(4)) # True

print(is\_even(7)) # False

**Пример 5: Рекурсивная функция**

Рекурсивная функция вызывает сама себя. Примером является функция для вычисления факториала числа.

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n - 1)

# Вызов функции

print(factorial(5)) # 120

**9. Полезные библиотеки Python**

**1. Библиотека itertools**

itertools — это стандартная библиотека Python, которая предоставляет функции для создания итераторов для эффективной работы с последовательностями данных.

**itertools.product**

Функция product создает декартово произведение (все возможные комбинации) элементов из нескольких последовательностей.

from itertools import product

a = [1, 2]

b = ['a', 'b']

result = list(product(a, b))

print(result) # [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]

**itertools.permutations**

Функция permutations возвращает все возможные перестановки элементов из последовательности заданной длины (по умолчанию длина равна длине последовательности).

from itertools import permutations

items = [1, 2, 3]

result = list(permutations(items, 2))

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

**itertools.combinations**

Функция combinations возвращает все возможные комбинации элементов из последовательности заданной длины, где порядок элементов не имеет значения.

from itertools import combinations

items = [1, 2, 3]

result = list(combinations(items, 2))

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

**2. Библиотека functools (декоратор @lru\_cache)**

Библиотека functools предоставляет инструменты для оптимизации и работы с функциями. Декоратор @lru\_cache используется для кэширования результатов функции, чтобы избежать повторных вычислений. Это полезно, например, для рекурсивных функций.

**Пример: Кэширование функции с @lru\_cache**

from functools import lru\_cache

@lru\_cache

def fibonacci(n):

if n < 2:

return n

return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(50)) # Быстрее, чем без кэширования

**3. Библиотека sys (установка лимита рекурсии setrecursionlimit)**

**sys** — стандартная библиотека Python, которая позволяет управлять параметрами системы и интерпретатора. Функция setrecursionlimit позволяет изменять максимальную глубину рекурсии. По умолчанию этот лимит установлен на 1000.

**Пример: Увеличение лимита рекурсии**

import sys

sys.setrecursionlimit(2000) # Устанавливаем новый лимит

# Пример рекурсивной функции

def recursive\_function(n):

if n == 0:

return

recursive\_function(n - 1)

recursive\_function(1500) # Не приведет к ошибке при увеличенном лимите

!! Будьте осторожны с увеличением лимита, так как это может привести к переполнению памяти и сбою программы.

**4. Библиотека fnmatch (функция fnmatch для работы с шаблонами)**

fnmatch — это библиотека для работы с шаблонами файлов. Функция fnmatch позволяет проверять, соответствует ли строка определенному шаблону (например, проверка расширения файлов или для масок чисел в задании 25).

Пример: Использование fnmatch для фильтрации файлов

from fnmatch import fnmatch

files = ["data1.txt", "data2.csv", "image.png", "notes.txt"]

filtered\_files = [file for file in files if fnmatch(file, "\*.txt")]

print(filtered\_files) # ['data1.txt', 'notes.txt']

Шаблоны:

- \* — соответствует любому количеству символов.

- ? — соответствует одному любому символу.

Эти библиотеки добавляют полезные функции для работы с последовательностями, оптимизации рекурсивных функций, управления лимитом рекурсии и фильтрации строк по шаблонам.

**10. Генераторы списков**

Генераторы списков (или list comprehensions) позволяют создавать списки компактным и удобным способом. Они часто используются для обработки элементов последовательности и создания новых списков с изменёнными или отфильтрованными значениями. Генераторы списков делают код более кратким и читаемым.

**Синтаксис**

новый\_список = [выражение for элемент in последовательность if условие]

- *выражение* — это значение, которое будет добавлено в новый список (может быть функция или преобразование над `элемент`).

- *условие* (необязательно) — фильтр, по которому добавляются только элементы, удовлетворяющие этому условию.

**Примеры**

**Пример 1: Создание списка квадратов чисел**

Создадим список квадратов чисел от 1 до 10.

squares = [x\*\*2 for x in range(1, 11)]

print(squares) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

**Пример 2: Фильтрация с условием**

Создадим список из четных чисел от 1 до 20.

evens = [x for x in range(1, 21) if x % 2 == 0]

print(evens) # [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

**Пример 3: Преобразование строк в верхний регистр**

Используем генератор списков для преобразования всех строк списка в верхний регистр.

words = ["hello", "world", "python", "list"]

uppercase\_words = [word.upper() for word in words]

print(uppercase\_words) # ['HELLO', 'WORLD', 'PYTHON', 'LIST']

**Пример 4: Вложенные генераторы списков**

Создадим таблицу умножения с помощью вложенного генератора списков.

table = [[i \* j for j in range(1, 6)] for i in range(1, 6)]

print(table) # [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]

**Пример 5: Замена элементов списка**

Создадим новый список, заменяя все отрицательные числа на ноль.

numbers = [5, -2, 9, -8, 0, 3]

replaced = [x if x >= 0 else 0 for x in numbers]

print(replaced) # [5, 0, 9, 0, 0, 3]

Генераторы списков позволяют удобно и быстро создавать списки, сокращая код и улучшая его читабельность. Они особенно полезны для простых преобразований и фильтраций.

**Заключение**

Эта методичка охватывает базовые темы Python. Рекомендуется попробовать каждый пример самостоятельно, чтобы лучше освоить язык!