Введение

Программирование - это искусство создания компьютерных программ, которые выполняют различные задачи и решают проблемы. Это ключевой инструмент в современном мире, позволяющий автоматизировать процессы, обрабатывать данные, создавать веб-сайты, мобильные приложения и многое другое. В этом введении мы рассмотрим основы программирования и ключевые концепции, которые вы будете изучать в ходе этой дисциплины.

Что такое программирование?

Программирование - это процесс написания кода, который выполняет определенные инструкции и действия на компьютере. Код может быть написан на различных языках программирования, таких как Python, Java, C++, JavaScript и многие другие. Каждый язык имеет свои синтаксис и особенности, но основные концепции программирования применимы ко всему спектру языков.

В изучении этой дисциплины мы познакомимся с основными концепциями программирования на языке Python. Мы начнем с изучения основ функционального программирования, таких как определение функций, параметры функций, и возвращаемые значения. Затем мы перейдем к изучению области видимости и переменных, включая локальные и глобальные переменные. Далее мы углубимся в модули и библиотеки, и рассмотрим способы их импорта и использования в наших программах.

Далее идет работа с данными, в рамках работы с данными мы изучим методы работы с файлами, включая открытие, чтение и запись файлов. Также рассмотрим форматированный вывод данных и методы обработки исключений, что позволит нам более гибко управлять ошибками в программах.

Глава - Объектно-ориентированное программирование (ООП):

Мы погрузимся в основы объектно-ориентированного программирования (ООП) и изучим основные концепции, такие как классы, объекты, атрибуты и методы классов. Мы также рассмотрим наследование и полиморфизм, которые позволяют создавать более гибкие и масштабируемые программы. В дополнение к этому, мы изучим инкапсуляцию и модификаторы доступа, которые обеспечивают контроль доступа к данным и методам в наших классах.

Глава №4: Графический интерфейс Tkinter:

Наконец, мы ознакомимся с библиотекой Tkinter для создания графических интерфейсов в Python. Мы изучим основные виджеты, меню,

диалоговые окна и организацию элементов интерфейса. После этого мы создадим простое графическое приложение, чтобы показать, как применять полученные знания на практике.

Изучение программирования дает мощный инструмент для решения задач, создания собственных проектов и участия в инновационных областях. Оно развивает логическое мышление, улучшает аналитические навыки и способствует креативному решению проблем. Программирование также предоставляет широкие возможности для карьерного роста и трудоустройства в информационных технологиях.

азработка лекций по дисциплине (Краткий курс лекций, презентации)

Глава №1.Функции и Модули

Тема №1. Определение и вызов функций

Параметры функций, возвращаемые значения.

Цель лекции: Познакомить студентов с основами функций и модулей в Python, а также научить их создавать собственные функции и использовать стандартные модули.

Что такое функция?

В программировании функция - это набор инструкций, объединенных вместе и предназначенных для выполнения конкретной задачи или выполнения конкретной операции. Функции являются основными строительными блоками программы и служат для организации кода в более читаемую и управляемую структуру.

Основные характеристики функций в программировании:

1)Имя функции: Функция имеет имя, по которому к ней можно обратиться при вызове.

2)Параметры (аргументы): Функция может принимать аргументы или параметры, которые представляют значения, передаваемые в функцию при вызове.

3)Тело функции: Тело функции содержит инструкции, которые выполняются при вызове функции. Это набор команд и выражений, реализующих конкретную логику.

4)Возвращаемое значение: Функция может возвращать результат выполнения в виде значения, и функция может не возвращать ничего.

Пример определения и вызова функции на Python:

```
# Определение функции с именем "приветствие"
def приветствие(имя):
    сообщение = f"Салам, {имя}!"
    return сообщение

# Вызов функции и передача аргумента "Чынара"
результат = приветствие("Чынара")
print(результат)  # Вывод
```

результат:

В этом примере приветствие - это функция, она принимает аргумент

имя, создает приветственное сообщение с использованием этого аргумента и возвращает его как результат. При вызове функции с аргументом "Чынара" выводится "Салам, Чынара!".

Зачем используются функции?

Функции используются в программировании по нескольким важным причинам:

1)Структурирование кода: Функции позволяют организовать код программы на более мелкие, легко управляемые блоки. Каждая функция выполняет конкретную задачу или операцию, что делает код более читаемым и понятным.

2)Повторное использование кода: Однажды написанные функции могут быть повторно использованы в различных частях программы или в других программах. Это сокращает дублирование кода и упрощает сопровождение.

3)Абстракция: Функции позволяют абстрагировать (сокращать) сложность. Вместо деталей реализации, пользователь функции может сосредотачиваться на её интерфейсе и том, что функция делает. Это облегчает понимание и использование функций.

4)Упрощение отладки: Когда ошибка возникает в коде, функции помогают ограничить область поиска ошибки. Вы можете тестировать и отлаживать функции независимо друг от друга.

5)Модульность: Функции могут быть организованы в модули, что упрощает структурирование и организацию кода программы.

6)Реализация алгоритмов: Функции позволяют разбить сложные задачи на более мелкие, более управляемые шаги. Это помогает при реализации алгоритмов и логики программы.

7)Улучшение читаемости: Использование функций делает код более читаемым и понятным. Имена функций могут служить документацией для того, что делает код.

8)Поддерживаемость и расширяемость: Путем разбиения программы на функции, вы делаете её более легко поддерживаемой и расширяемой. Новые функции могут быть добавлены без изменения существующего кода.

В общем, функции - это мощный инструмент в программировании, который помогает сделать код более структурированным, эффективным и легко поддерживаемым.

Ключевое слово **def** используется в Python для определения (создания) функций. Синтаксис определения функции выглядит следующим образом:

```
def имя_функции(параметры):
    # Тело функции
    # Выполняемые инструкции
    return результат
```

где:

имя_функции - это имя, которое вы придумываете для функции.

параметры - это список аргументов, которые функция принимает

(они могут быть пустыми, если функция не принимает аргументов).

Тело функции - это блок кода, который выполняется при вызове функции. Он должен быть с отступом (обычно 4 пробела или 1 табуляция) от начала строки.

return - это ключевое слово, которое используется для возвращения значения из функции (это опционально).

Передача аргументов в функцию:

Параметры функций - это способ передать данные или аргументы внутрь функции, чтобы она могла их использовать. Аргументы могут быть переданы функции при её вызове.

Пример передачи аргументов в функцию:

```
def приветствие(имя):
    print(f"Привет, {имя}!")

приветствие("Чынара")  # Вызываем функцию с аргументом "Чынара"
```

результат:

```
Привет, Чынара!
```

В этом примере *имя* - это параметр функции *приветствие*, и мы передаем аргумент "Чынара" при вызове функции.

Позиционные и именованные аргументы:

1)Позиционные аргументы: При передаче аргументов в функцию важен порядок. Аргументы, переданные первыми, соответствуют первым параметрам функции, аргументы, переданные вторыми, соответствуют вторым параметрам и так далее. Это называется "позиционными аргументами".

Пример:

```
def сложение(a, b):
    сумма = a + b
    print(сумма)

сложение(3, 5)  # 3 и 5 - позиционные аргументы
```

результат:

```
8
```

2)Именованные аргументы: Вы можете передавать аргументы в функцию с указанием имени параметра, к которому они относятся. Это называется "именованными аргументами". Это позволяет вам передавать аргументы в произвольном порядке, даже если у функции много параметров.

Пример:

```
def приветствие(имя, возраст):
    print(f"Привет, {имя}! Тебе {возраст} лет.")

приветствие(возраст=30, имя="Чынара")  # именованные аргументы
```

результат:

```
Привет, Чынара! Тебе 30 лет.
```

Значения по умолчанию для аргументов:

В Python вы можете установить значения по умолчанию для параметров функции. Если аргумент не передается при вызове функции, используется значение по умолчанию.

Пример:

```
def приветствие(имя, возраст=30):
    print(f"Привет, {имя}! Тебе {возраст} лет.")

приветствие("Чынара")  # Возраст не передан, используется значение по умолчанию 30
приветствие("Тимур", 25)  # Передано значение 25, оно заменяет значение по умолчанию
```

результат:

```
Привет, Чынара! Тебе 30 лет.
Привет, Тимур! Тебе 25 лет.
```

В этом примере возраст имеет значение по умолчанию 30, но при необходимости его можно переопределить, передав явное значение в вызове функции.

Знание о передаче аргументов и значениях по умолчанию позволяет создавать гибкие и мощные функции, которые могут работать с различными данными.

Тема №2. Область видимости и переменные. Локальные и глобальные переменные.

Область видимости и переменные - это важные концепции в программировании, которые определяют, где и как можно использовать переменные в коде. Они помогают организовать данные в программе и контролировать доступ к ним.

1)Локальные переменные:

- Локальные переменные объявляются внутри определенной области видимости, обычно внутри функции или блока кода.
- Они видимы только внутри той функции или блока, в котором были объявлены.
- Попытка доступа к локальной переменной за пределами ее области видимости вызовет ошибку.
- Локальные переменные обычно используются для временного хранения данных внутри функции.

Пример на Python:

```
def example_function():
    x = 10  # x - локальная переменная
    print(x)

example_function()  # Вернет 10
print(x)  # Ошибка: x не определена вне функции
```

результат:

```
10
    print(x)  # Ошибка: x не определена вне функции
NameError: name 'x' is not defined
```

2)Глобальные переменные:

- Глобальные переменные объявляются за пределами всех функций и блоков кода.
- Они видимы во всем коде программы после своего объявления.
- Глобальные переменные можно использовать как для чтения, так и для записи в любой части программы.
- Глобальные переменные могут быть удобными, но также могут сделать код менее читаемым и подверженным ошибкам, если их использовать неосторожно.

Пример на Python:

```
global_variable = 20  # global_variable - глобальная переменная

def another_function():
    print(global_variable)  # Можно использовать глобальную переменную

another_function()  # Вернет 20
print(global_variable)  # Вернет 20, так как она доступна глобально
```

результат:

```
20
20
```

3)Область видимости:

- Область видимости определяет, где можно обратиться к переменной и где она видима.
- В языках программирования существуют разные уровни областей видимости, такие как локальная, глобальная и вложенная области видимости.
- Если переменная имеет одно имя в разных областях видимости, то будет использоваться значение из наиболее близкой к месту обращения области видимости.

Пример на Python:

```
x = 10  # Глобальная переменная

def example_function():
    x = 5  # Локальная переменная с тем же именем, затеняющая глобальную
    print(x)  # Выводит 5, так как обращение идет к локальной переменной

example_function()
print(x)  # Выводит 10, так как обращение идет к глобальной переменной
```

результат:

```
5
10
```

Важно понимать, как работают локальные и глобальные переменные, чтобы правильно структурировать код и избежать ошибок в программе.

Вот пример задания, которое демонстрирует использование как локальных, так и глобальных переменных в Python:

Задание: Калькулятор(Способ 1)

Напишите программу для выполнения математических операций. Программа должна иметь следующие функции:

1)**add(a, b):** Функция принимает два аргумента a и b, выполняет сложение и выводит результат на экран.

2)subtract(a, b): Функция принимает два аргумента a и b, выполняет вычитание и выводит результат на экран.

3)multiply(a, b): Функция принимает два аргумента a и b, выполняет умножение и выводит результат на экран.

4)divide(a, b): Функция принимает два аргумента a и b, выполняет деление и выводит результат на экран. Проверьте, что b не равно нулю, чтобы избежать ошибки деления на ноль.

Глобальная переменная operation будет использоваться для хранения текущей выполняемой операции, такой как "сложение", "вычитание", "умножение" или "деление". Эта глобальная переменная будет установлена в соответствии с выбором пользователя, и локальные переменные будут использоваться для хранения аргументов a и b для каждой операции.

```python
# Глобальная переменная для хранения текущей операции
operation = ""

def add(a, b):
    global operation  # Используем глобальную переменную
    operation = "сложение"
    result = a + b
    print(f"Результат {operation}: {result}")

def subtract(a, b):
    global operation
    operation = "вычитание"
    result = a - b
    print(f"Результат {operation}: {result}")

def multiply(a, b):
    global operation
    operation = "умножение"
    result = a * b
    print(f"Результат {operation}: {result}")

def divide(a, b):
    global operation
    operation = "деление"
    if b != 0:
        result = a / b
        print(f"Результат {operation}: {result}")
    else:
        print("Ошибка: Деление на ноль!")

# Пример использования
add(5, 3)
subtract(10, 4)
multiply(7, 2)
divide(8, 0)  # Произойдет деление на ноль
```

результат:

```
Результат сложение: 8
Результат вычитание: 6
Результат умножение: 14
Ошибка: Деление на ноль!
```

В этом примере operation - глобальная переменная, а аргументы a и b - локальные переменные для каждой функции. Программа выполняет математические операции в зависимости от выбора пользователя и выводит результаты на экран.

Задание: Калькулятор(Способ 2)

Для создания простого калькулятора, использующего локальные и глобальные переменные для выполнения математических операций, вы

можете написать программу на языке Python. Вот пример такой программы:

```
# Глобальная переменная для хранения результата
результат = 0
def сложить(число):
    global результат
    результат += число
def вычесть(число):
    global результат
    результат -= число

def умножить(число):
    global результат
    результат *= число

def разделить(число):
    global результат
    if число != 0:
        результат /= число
    else:
        print("Ошибка: деление на ноль!")
def главное_меню():
    global результат
    print("Текущий результат:", результат)
    print("1. Сложить")
    print("2. Вычесть")
    print("3. Умножить")
    print("4. Разделить")
    print("5. Выйти")
    выбор = input("Выберите операцию (1/2/3/4/5): ")
    if выбор == '1':
        число = float(input("Введите число для сложения: "))
        сложить(число)
    elif выбор == '2':
        число = float(input("Введите число для вычитания: "))
        вычесть(число)
    elif выбор == '3':
        число = float(input("Введите число для умножения: "))
        умножить(число)
    elif выбор == '4':
        число = float(input("Введите число для деления: "))
        разделить(число)
    elif выбор == '5':
        print("До свидания!")
        exit()
    else:
        print("Некорректный выбор операции.")
while True:
    главное_меню()
```

```
Текущий результат: 0
1. Сложить
2. Вычесть
3. Умножить
4. Разделить
5. Выйти
Выберите операцию (1/2/3/4/5): 1
Введите число для сложения: 56
Текущий результат: 56.0
1. Сложить
2. Вычесть
3. Умножить
4. Разделить
5. Выйти
Выберите операцию (1/2/3/4/5): 2
Введите число для вычитания: 45
Текущий результат: 11.0
1. Сложить
2. Вычесть
3. Умножить
4. Разделить
5. Выйти
Выберите операцию (1/2/3/4/5): 3
Введите число для умножения: 2
Текущий результат: 22.0
1. Сложить
2. Вычесть
3. Умножить
4. Разделить
5. Выйти
Выберите операцию (1/2/3/4/5): 4
Введите число для деления: 2
Текущий результат: 11.0
1. Сложить
2. Вычесть
3. Умножить
4. Разделить
5. Выйти
Выберите операцию (1/2/3/4/5): 5
```

В этой программе:

“результат” является глобальной переменной, в которой хранится текущий результат операций.

Функции сложить, вычесть, умножить и разделить принимают число и выполняют соответствующие математические операции, обновляя глобальную переменную результат.

Функция главное_меню отображает главное меню, предлагая выбрать операцию, и вызывает соответствующую функцию в зависимости от выбора пользователя.

Этот калькулятор позволяет пользователю выполнять операции сложения, вычитания, умножения и деления с текущим результатом.

Цикл **while True**, чтобы программа оставалась активной до тех пор, пока пользователь не решит выйти. Это распространенный подход для создания интерактивных консольных приложений. При желании пользователь может выйти из программы, выбрав соответствующую опцию.

Когда пользователь выбирает опцию "Выйти" в главном меню, программа вызывает exit(), что приводит к завершению программы. Это обычно используется в консольных приложениях для завершения работы программы.

Тема №3.Модули и библиотеки
Импорт модулей, стандартные библиотеки.

В программировании, модуль — это файл, который содержит Python-код, включая функции изменения, классы и функции. Модули предназначены для организации кода, чтобы обеспечить его чистоту, читаемость, повторное использование и структурированность.

Создание модулей - это один из фундаментальных принципов модульного программирования, который позволяет разбить большую программу на более мелкие и независимые части (модули). Это упрощает разработку, отладку и поддержку программного обеспечения. В зависимости от языка программирования, который вы используете, процесс создания модулей может немного различаться. Вот общие шаги по созданию модулей:

1)Определение функциональности модуля: Сначала определите, какая функциональность будет реализована в вашем модуле. Модуль должен выполнять конкретную задачу или предоставлять определенный набор функций.

2)Выбор имени модуля: Подумайте над именем для вашего модуля. Хорошие имена должны быть описательными и уникальными, чтобы избежать конфликтов с именами других модулей.

3)Создание файла модуля: В большинстве языков программирования модуль представляется файлом. Создайте файл с именем вашего модуля и добавьте расширение, соответствующее языку программирования (например, .py для Python или .js для JavaScript).

4)Определение интерфейса модуля: Определите интерфейс модуля, то есть список функций, классов или переменных, которые будут доступны извне модуля. Это включает в себя ключевые функции и данные, которые другие части программы смогут использовать.

5)Реализация функциональности: Напишите код, который реализует функциональность вашего модуля. Убедитесь, что код модуля хорошо структурирован и легко читаем.

6)Импортирование модуля: В других частях вашей программы импортируйте созданный модуль, чтобы использовать его функциональность. Способ импорта зависит от языка программирования.

7)Тестирование модуля: Протестируйте ваш модуль, чтобы убедиться, что он работает правильно. Создайте тестовые случаи для проверки разных аспектов функциональности модуля.

8)Документация: Добавьте документацию к вашему модулю. Опишите, как использовать интерфейс модуля, какие параметры принимают функции, и что они возвращают. Хорошая документация помогает другим разработчикам быстро разобраться в вашем модуле.

9)Обработка ошибок и исключений: Предусмотрите обработку ошибок и исключений в вашем модуле, чтобы улучшить его надежность.

10)Распространение модуля: Если ваш модуль предназначен для использования другими разработчиками, убедитесь, что он легко доступен

для установки и использования. В случае Python, например, вы можете опубликовать модуль на Python Package Index (PyPI).

11)Обновление и поддержка: Поддерживайте ваш модуль, обновляя его при необходимости, и реагируйте на обратную связь от пользователей.

Это общие шаги, которые следует выполнять при создании модулей в программировании. Конкретные детали могут зависеть от языка программирования, платформы и целей вашего проекта.

Давайте рассмотрим пример создания модуля на Python. Создадим простой модуль, который содержит функцию для вычисления суммы двух чисел. Этот модуль можно использовать в других Python-программах.

1.Создайте файл с именем **my_module.py.** В этом файле мы определим наш модуль.

2.В файле my_module.py добавьте следующий код:

```
# my_module.py

def add_numbers(a, b):
    """Эта функция складывает два числа и возвращает результат."""
    return a + b
```

Этот код определяет модуль my_module с одной функцией add_numbers, которая принимает два аргумента a и b, складывает их и возвращает результат.

1.Теперь мы можем создать другую Python-программу и использовать наш модуль. Для этого создайте новый файл (например, main.py) и добавьте следующий код:

```
# main.py
import my_module

result = my_module.add_numbers(5, 7)
print("Результат сложения:", result)
```

Здесь мы импортируем модуль my_module с помощью оператора import и используем функцию add_numbers из этого модуля для сложения чисел 5 и 7. Результат будет выведен на экран.

Запустите файл main.py. Вы должны увидеть следующий вывод:

```
Результат сложения: 12
```

Библиотеки - это наборы функций, классов и ресурсов, предназначенные для решения определенных задач в программировании. Они представляют собой уже написанный и протестированный код, который можно использовать в ваших собственных программных проектах для упрощения разработки и расширения функциональности.

Библиотеки могут быть написаны другими разработчиками и распространяться в виде отдельных пакетов или модулей, которые можно импортировать в ваши программы. Вот некоторые примеры библиотек:

1.Библиотеки стандартной библиотеки: Многие языки программирования, такие как Python, содержат стандартную библиотеку, которая включает в себя множество модулей и функций для общих задач, таких как работа с файлами, сетью, математические вычисления и т. д.

2.Библиотеки для работы с графикой: Например, библиотека Pygame для создания компьютерных игр или библиотеки для визуализации данных, такие как Matplotlib для Python.

3.Библиотеки для работы с базами данных: Например, SQLAlchemy для Python, Hibernate для Java.

4.Библиотеки для машинного обучения и искусственного интеллекта: Например, TensorFlow, PyTorch, scikit-learn для Python.

5.Библиотеки для веб-разработки: Например, Express.js для Node.js, Django и Flask для Python.

6.Библиотеки для обработки текста и работы с языками: Например, NLTK для обработки текста на естественных языках, регулярные выражения для обработки строк.

7.Библиотеки для работы с графами и сетями: Например, NetworkX для Python.

8.Библиотеки для разработки пользовательского интерфейса: Например, Qt для C++ или PyQt/PySide для Python.

Чтобы использовать библиотеку в своем проекте, вам обычно нужно выполнить следующие шаги:

Установка библиотеки: Если библиотека не включена в стандартную библиотеку вашего языка программирования, вам нужно будет установить ее с помощью инструмента управления пакетами, такого как “pip” для Python или “npm” для Node.js.

Импорт библиотеки: В вашем коде вы импортируете нужные модули или классы из библиотеки.

Использование функций и классов: Вы используете функции и классы из библиотеки для выполнения необходимых задач в вашем проекте.

Примеры:

```
# Импорт библиотеки
import math

# Использование функции из библиотеки
result = math.sqrt(25)  # Вычисление квадратного корня
print(result)
```

результат:

```
5.0
```

Библиотеки значительно упрощают разработку, так как они предоставляют готовый и проверенный код для широкого спектра задач.

Импорт модуля в Python осуществляется с использованием ключевого слова import.

Давайте рассмотрим несколько примеров импорта модулей в Python:

1.Импорт всего модуля:

Предположим, у нас есть файл math_operations.py с таким содержанием:

```python
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Теперь мы можем импортировать этот модуль и использовать его функции:

```python
import math_operations

result = math_operations.add(5, 3)
print("Сумма:", result)

difference = math_operations.subtract(10, 4)
print("Разность:", difference)
```

Результат:

```
Сумма: 8
Разность: 6
```

2.Импорт конкретных функций из модуля:

Если нам нужны только определенные функции из модуля, мы можем импортировать их напрямую:

```python
from math_operations import add, subtract

result = add(7, 2)
print("Сумма:", result)

difference = subtract(15, 6)
print("Разность:", difference)
```

результат:

```
Сумма: 9
Разность: 9
```

3.Импорт модуля с алиасом:

Мы также можем использовать алиас (псевдоним) при импорте:

```python
import math_operations as mo
result = mo.add(8, 4)
print("Сумма:", result)
difference = mo.subtract(20, 7)
print("Разность:", difference)
```

результат:

```
Сумма: 12
Разность: 13
```

4.Импорт из стандартной библиотеки:

Python имеет множество встроенных модулей. Например, для работы с датами и временем мы можем импортировать модуль datetime следующим образом:

```python
import datetime

current_time = datetime.datetime.now()
print("Текущее время:", current_time)
```

результат:

```
Текущее время: 2023-09-18 10:14:29.957550
```

Стандартные библиотеки в python.

Python имеет обширную стандартную библиотеку, которая включает в себя множество модулей и функций для решения различных задач. Вот некоторые из основных модулей и функций в стандартной библиотеке Python:

1.Модуль math:
math.sqrt(): Квадратный корень.
math.pow(): Возведение в степень.
math.sin(), math.cos(), math.tan(): Тригонометрические функции.
math.pi, math.e: Константы.
2.Модуль datetime:
datetime.datetime(): Работа с датами и временем.
datetime.timedelta(): Разница между двумя датами.
datetime.date(), datetime.time(): Работа с датой и временем отдельно.
3.Модуль os:
os.path: Работа с путями к файлам и директориям.
os.listdir(): Список файлов в директории.
os.mkdir(), os.makedirs(): Создание директорий.
os.remove(), os.unlink(): Удаление файлов.
4.Модуль json:
json.dumps(): Сериализация объекта Python в формат JSON.
json.loads(): Десериализация JSON в объект Python.
5.Модуль urllib:
urllib.request.urlopen(): Выполнение HTTP-запросов.
urllib.parse.urlencode(): Кодирование параметров URL.
6.Модуль collections:
collections.Counter(): Подсчет элементов в последовательности.
collections.defaultdict(): Словарь с значениями по умолчанию.
7.Модуль random:
random.random(): Генерация случайного числа от 0 до 1.
random.randint(): Генерация случайного целого числа в заданном диапазоне.
8.Модуль csv:
csv.reader(): Чтение данных из CSV-файла.
csv.writer(): Запись данных в CSV-файл.
9.Модуль socket:
socket.socket(): Создание сокета для сетевого взаимодействия.
socket.connect(), socket.bind(), socket.listen(): Основные сетевые операции.
10.Модуль re: Регулярные выражения для поиска и замены текста в строках.

Глава №2.Работа с данными

№4.Работа с файлами и ввод/вывод данных

Открытие, чтение и запись файлов.
Форматированный вывод данных.

Основы работы с файлами в Python включают в себя функцию **open()**, режимы открытия файлов и использование менеджера контекста **with** для автоматического закрытия файлов. Вот более подробное описание этих концепций и примеры чтения текстовых файлов:

Функция **open()** и режимы открытия файлов

Функция **open()** используется для открытия файлов в Python. Она принимает два обязательных аргумента: имя файла и режим открытия. Режим определяет, как можно использовать файл. Вот некоторые распространенные режимы:

1.**'r' (чтение)**: Открывает файл для чтения. Если файл не существует, будет сгенерировано исключение.

2.**'w' (запись)**: Открывает файл для записи. Если файл существует, он будет перезаписан. Если файл не существует, он будет создан.

3.**'a' (добавление)**: Открывает файл для записи, добавляя данные в конец файла. Если файл не существует, он будет создан.

4.**'b' (бинарный режим)**: Открывает файл в бинарном режиме, который позволяет работать с бинарными данными, например, изображениями или видео. Этот режим можно добавить к любому из вышеперечисленных режимов.

Менеджер контекста **with** для автоматического закрытия файлов

Python поддерживает использование менеджера контекста **with**, который обеспечивает автоматическое закрытие файла после завершения блока кода, в котором используется файл. Это рекомендуется использовать для избегания утечки ресурсов и обеспечения правильного закрытия файла, даже если произошла ошибка в процессе выполнения кода.

Вот конкретный пример работы с файлами в Python, где мы будем создавать, записывать и читать текстовый файл:

```
# Создаем и записываем данные в файл
with open('example.txt', 'w') as file:
    file.write('Привет, мир!\n')
    file.write('Это пример работы с файлами в Python.\n')
    file.write('Мы можем записывать данные в файл и читать их из него.\n')

# Чтение данных из файла и вывод на экран
with open('example.txt', 'r') as file:
    content = file.read()
    print("Содержимое файла:")
    print(content)
```

результат:

```
Содержимое файла:
Привет, мир!
Это пример работы с файлами в Python.
Мы можем записывать данные в файл и читать их из него.
```

Этот код выполняет следующие действия:

1.Открывает файл "example.txt" в режиме записи ('w') и записывает в

него три строки текста.

2.Затем он открывает тот же файл в режиме чтения ('r') и считывает содержимое файла в переменную content.

3.Выводит содержимое файла на экран.

Таким образом, данный пример демонстрирует, как создавать, записывать и читать текстовые файлы в Python с использованием менеджера контекста **with.**

Форматированный вывод данных python

В Python существует несколько способов форматирования вывода данных. Вот некоторые из них:

1. F-строки (f-strings):

F-строки (f-strings) представляют собой удобный и читаемый способ форматирования строк, введенный в Python 3.6 и выше. Они позволяют вставлять значения переменных непосредственно в строку, помещая их в фигурные скобки {} и предваряя строку символом f или F. Например:

```
name = "Alice"
age = 30
formatted_string = f"Имя: {name}, Возраст: {age}"
print(formatted_string)
```

результат:

```
Имя: Alice, Возраст: 30
```

2. Метод format():

Метод format() используется для форматирования строк, заменяя заполнители в строке на значения, переданные методу. Заполнители указываются в фигурных скобках {} и могут быть пронумерованными или без номеров. Примеры:

```
name = "Bob"
age = 25
formatted_string = "Имя: {}, Возраст: {}".format(name, age)
print(formatted_string)

# Или с использованием номеров
formatted_string = "Имя: {0}, Возраст: {1}".format(name, age)
print(formatted_string)
```

результат:

```
Имя: Bob, Возраст: 25
Имя: Bob, Возраст: 25
```

3. Оператор % (старый способ):

Оператор % используется для форматирования строк, подобно тому, как это делается в языке C. В этом случае, используются специальные символы, такие как %s для строк и %d для целых чисел. Пример:

```
name = "Chynara"
age = 40
formatted_string = "Имя: %s, Возраст: %d" % (name, age)
print(formatted_string)
```

результат:

```
Имя: Chynara, Возраст: 40
```

4. Метод join():

Если у вас есть список значений, которые вы хотите объединить в строку, можно использовать **метод join()**:

```
items = ["яблоко", "груша", "банан"]
formatted_string = ", ".join(items)
print(formatted_string)
```

результат:

```
яблоко, груша, банан
```

5. Метод str.format_map() (Python 3.2 и выше):

Этот метод позволяет форматировать строки, используя словарь в качестве источника значений для заполнителей:

```
data = {'name': 'David', 'age': 35}
formatted_string = "Имя: {name}, Возраст: {age}".format_map(data)
print(formatted_string)
```

результат:

```
Имя: David, Возраст: 35
```

Эти методы форматирования строк дают вам большую гибкость в управлении выводом данных и позволяют создавать читаемый и структурированный вывод. F-строки на данный момент считаются наиболее современным и читаемым способом форматирования строк в Python.

Тема №5.Обработка исключений

Обработка ошибок и исключений.

Создание пользовательских исключений.

Часть 1: Обработка исключений и ошибок

I.Введение в исключения и обработку ошибок.

- Понятие исключения.
- Зачем нужна обработка ошибок.

Понятие исключения

Исключение (exception) - это событие или объект, которое возникает в ходе выполнения программы и указывает на некорректное или неожиданное состояние или действие. Исключения используются для обработки ошибок и исключительных ситуаций в программе. Когда происходит исключение, выполнение программы переходит к специальному блоку кода, предназначенному для обработки этого исключения, вместо аварийного завершения программы.

Основные характеристики исключений:

1.Возникновение исключений: Исключения могут возникать в разных частях программы при выполнении определенных операций или в ответ на определенные события, например, при делении на ноль, доступе к несуществующему файлу или индексу списка.

2.Обработка исключений: Чтобы предотвратить аварийное завершение программы при возникновении исключения, можно использовать механизм обработки исключений. В Python это реализуется с помощью блока **try-except**, который позволяет определить, как обрабатывать исключение.

3.Иерархия исключений: Исключения в Python имеют иерархию, где базовый класс - **BaseException**, и от него наследуются различные типы исключений, такие как **Exception, TypeError**, ValueError, и многие другие. Вы можете выбирать наиболее подходящий тип исключения для конкретной ситуации.

4.Создание пользовательских исключений: В дополнение к встроенным исключениям, Python позволяет создавать пользовательские исключения, чтобы обрабатывать специфические ошибки или события, связанные с вашей программой.

Примеры типичных событий, которые могут вызвать исключения:

- Деление на ноль.
- Доступ к несуществующему файлу.
- Индексирование списка за пределами его размера.
- Попытка преобразования строки в число, если строка не содержит числовые данные.

Исключения помогают программистам обнаруживать и обрабатывать ошибки, делая код более надежным и предсказуемым.

Зачем нужна обработка ошибок

Обработка ошибок является важной частью программирования, и ее важность объясняется несколькими ключевыми аспектами:

- Предотвращение аварийного завершения программы: Одной из основных целей обработки ошибок является предотвращение аварийного завершения программы при возникновении непредвиденных ситуаций. Если программа не обрабатывает ошибки, она может выйти из строя и завершиться, что может быть нежелательным в продуктивной среде.
- Повышение надежности программы: Обработка ошибок позволяет программе обнаруживать и корректно реагировать на ошибки и исключительные ситуации, что увеличивает надежность программного обеспечения. Пользовательский опыт также улучшается, так как программа может сообщать о проблемах и предлагать варианты действий.
- Логирование и отладка: Обработка ошибок обеспечивает механизм для регистрации информации об ошибках и событиях, происходящих в программе. Это облегчает процесс

отладки и анализа проблем, так как вы можете видеть, где и почему возникают ошибки.

- Улучшение безопасности: Обработка ошибок помогает предотвратить уязвимости и ошибки в безопасности, так как позволяет программе корректно обрабатывать попытки злоупотребления или внедрения вредоносного кода.
- Повышение управляемости: Обработка ошибок позволяет программистам контролировать поток выполнения программы в случае возникновения ошибок. Это означает, что можно определить, какие действия следует предпринять при возникновении ошибки, и какие части программы должны быть пропущены.
- Улучшение пользовательского опыта: Обработка ошибок позволяет предоставить пользователю информацию о проблемах и предложить понятные инструкции о том, как решить проблему или что делать дальше. Это помогает создать более дружелюбный интерфейс для пользователя.
- Соблюдение требований и стандартов: В некоторых случаях, таких как разработка программного обеспечения для безопасных систем или соблюдение нормативных требований, обработка ошибок может быть обязательной.

В целом, обработка ошибок является важным аспектом разработки программного обеспечения, который способствует созданию более надежных, безопасных и управляемых приложений.

II.Использование блока try-except.

Синтаксис блока try-except.

Обработка конкретных исключений.

Блок try-except - это механизм в Python для обработки исключений, который позволяет программистам предотвратить аварийное завершение программы при возникновении ошибок и выполнить альтернативные действия. Вот синтаксис блока try-except:

```
try:
    # Код, который может вызвать исключение
    # ...
except <Исключение_1>:
    # Обработка исключения_1
    # ...
except <Исключение_2>:
    # Обработка исключения_2
    # ...
else:
    # Код, который выполняется, если исключение не возникло
    # ...
finally:
    # Код, который выполняется всегда, независимо от того, возникло исключение или нет
    # ...
```

Объяснение элементов блока try-except:

try: В блоке try помещается код, который может вызвать исключение. Программа будет выполняться в этом блоке до момента возникновения исключения.

except: Блок except указывает, как обрабатывать исключение. Вы

можете указать один или несколько блоков except для обработки разных типов исключений. Если указанное исключение возникнет в блоке try, выполнение программы перейдет к соответствующему блоку except.

Исключение_1, Исключение_2, и так далее: Это имена типов исключений, которые вы хотите перехватить. Например, ZeroDivisionError, ValueError, FileNotFoundError и так далее.

else: Блок else содержит код, который выполняется, если исключение не возникло в блоке try.

finally: Блок finally содержит код, который выполняется всегда, независимо от того, возникло исключение или нет. Это полезно, например, для закрытия ресурсов, таких как файлы или сетевые соединения.

Пример использования блока try-except:

```
try:
    num = int(input("Введите число: "))
    result = 10 / num
except ZeroDivisionError:
    print("Деление на ноль не допустимо.")
except ValueError as e:
    print(f"Ошибка преобразования: {e}")
else:
    print(f"Результат: {result}")
finally:
    print("Завершение программы.")
```

Этот пример демонстрирует, как обработать исключения ZeroDivisionError и ValueError с использованием блока try-except.

Глава №3. Объектно-ориентированное программирование (ООП)
Тема.№6. ВВедение в ООП: Обзор парадигм программирования. Основные концепции ООП. Классы и объекты: Определение классов и объектов. Атрибуты и методы классов.Создание класса и объекта.

Введение в программирование и парадигмы программирования.

Объясним объектно-ориентированное программирование **(ООП)** простыми словами.

ООП — это способ написать программу, используя образы объектов. Все вокруг нас, в первом мире, можно представить как объекты. Например, машина — это объект, у нее есть такие характеристики, как марка, модель, цвет, и она может выполнять такие действия, как движение и остановка.

В программировании объектов - это как машины. У них тоже есть характеристики (атрибуты) и способность выполнять действия (методы).

Пример:

Предполагается, что у нас есть класс "Автомобиль". Этот класс имеет атрибуты (характеристики), такие как марка и модель, и методы

(действия), такие как «завести» и «остановить». Мы можем создать несколько объектов этого класса, представляющих разные автомобили разных марок и моделей. Каждый из этих объектов будет иметь свои собственные значения атрибутов и сможет выполнять методы.

Важно отметить, что ООП помогает нам организовать код более структурированно и упорядоченно, делая его более простым для понимания и поддержки.

Введение в объектно-ориентированное программирование (ООП) - это основополагающий этап при изучении программирования. ООП — это парадигма программирования, ориентированная на программы моделирования для определения объектов, каждый из которых имеет свойства (атрибуты) и методы (функции), которые могут взаимодействовать друг с другом. Давайте рассмотрим основные понятия и термины ООП:

1.Объекты:Объекты представляют собой отдельные экземпляры классов. Классы определяют характер и поведение объектов.

2. Классы:Класс - это шаблон или чертеж, определяющий структуру и поведение объектов. Классифицируйте атрибуты (свойства) и методы (функции), которые будут иметь объекты этого класса.

3.Атрибуты:Атрибуты (или поля) класса представляют собой данные, которые относятся к объекту. Например, если у нас есть класс «Автомобиль», то его атрибутами могут быть «марка», «модель» и «год выпуска».

4.Методы:Методы класса - это функции, которые определяют поведение объектов этого класса. Они могут изменять атрибуты объекта или выполнять какие-либо операции. Например, метод «завести» для класса «Автомобиль» может изменить статус автомобиля на «заведен».

5.Инкапсуляция: Инкапсуляция - это принцип, согласно данным атрибутам и методам класса, скрытым от прямого доступа извне. Это сделано в целях безопасности и контроля состояния объекта.

6.Наследование:Наследование - это механизм, создающий новые классы на основе существующих. Подкласс наследует атрибуты и методы суперкласса и может включать или изменять их.

7.Полиморфизм:Полиморфизм — это способность объектов разных классов использовать общий интерфейс. Это позволяет одному методу работать с объектами разных типов.

Обзор парадигм программирования помогут вам лучше понять,что такое ООП и как оно связано с языком программирования Python.

Парадигмы программирования - это основные подходы к разработке программного обеспечения, которые определяют структуру, стиль и методологию написания кода. ООП является одной из наиболее известных парадигм программирования. Давайте рассмотрим основные парадигмы, а затем перейдем к ООП и Python.

Вот обзор некоторых основных парадигм программирования:

1.Императивное программирование: Описание шагов, которые

необходимо выполнить, чтобы добиться желаемого результата на компьютере.
Примеры языков: C, C++, Java.
Примеры команд: применение запроса, циклы, условные операторы.

2.Декларативное программирование:Описывает желаемый результат, а не шаги для его достижения.
Примеры языков: SQL, HTML, CSS.
Примеры: запросы к базе данных, описание структуры веб-страниц.

3.Функциональное программирование: Основано на математических функциях и работе с данными без изменения их состояния.
Примеры языков: Haskell, Lisp, JavaScript (часто).
Функции рассматриваются как объекты первого класса, абстрагирующие вычисления.

4.Логическое программирование: Программа состоит из логических утверждений и фактов, система выдает результаты на основе логических правил.
Пример языка: Пролог.
Используется в искусственном интеллекте и экспертных знаниях.

5.Объектно-ориентированное программирование (ООП): Основано на концепциях объектов, которые имеют состояние (атрибуты) и поведение (методы).
Примеры языков: Python, Java, C++.
Подход Позволяет моделировать реальные объекты и их взаимодействие.

Классы и объекты — это основная концепция объектно-ориентированного программирования (ООП). Давайте определим, что такое классы и объекты:

Класс (Класс):

- Класс - это создание шаблона или чертежа, описывающего, как создавать объекты.
- Он определяет атрибуты (переменные) и методы (функции), которые будут иметь объекты.
- Класс можно рассматривать как определение нового типа данных.

Объект (Объект):

- Объект — это конкретный экземпляр класса, созданный на основе его определения.
- Объекты имеют собственные значения атрибутов, которые могут отличаться от атрибутов других объектов того же класса.
- Объекты могут включать методы, термостаты в классе.

Создание класса и объекта пример:

Простой пример создания класса и объекта на Python может быть связан с моделированием геометрической фигуры, например, прямоугольника. Вот как это может выглядеть:

```python
# Определение класса Rectangle (прямоугольник)
class Rectangle:
    # Конструктор класса для инициализации атрибутов
    def __init__(self, width, height):
        self.width = width    # Атрибут "width" (ширина)
        self.height = height  # Атрибут "height" (высота)

    # Метод для вычисления площади прямоугольника
    def calculate_area(self):
        return self.width * self.height

# Создание объекта класса Rectangle
my_rectangle = Rectangle(5, 3)

# Вызов метода для вычисления площади
area = my_rectangle.calculate_area()

# Вывод результата
print(f"Площадь прямоугольника: {area}")
```

результат:

```
Площадь прямоугольника: 15
```

Тема №7. Основы классов и объектов
Конструктор и инициализация: Роль конструктора.
Метод __init__ и инициализация атрибутов объекта.
Передача аргументов конструктору.
Атрибуты и методы:Объявление атрибутов и методов класса.Доступ к атрибутам и вызов методов объекта. Специальный атрибут self.

Концепция классов и объектов является фундаментальной в объектно-ориентированном программировании (ООП). Давайте подробнее рассмотрим каждый аспект этой концепции с определениями.

1. Определение класса:

Класс в Python - это абстрактный шаблон, который определяет состояние (атрибуты) и поведение (методы) объектов. Он представляет собой структуру данных, которая описывает, какие данные могут содержаться в объектах класса и какие операции можно выполнять над этими объектами. Классы обычно создаются с целью создания множества объектов с общими характеристиками.

Пример определения класса:

```python
class Car:
    # Это определение класса "Car". Он описывает характеристики и поведение автомобилей.
    pass
```

В этом определении класса **Car**, у нас пока нет атрибутов или методов, но класс служит основой для создания объектов,

представляющих конкретные автомобили.
pass позволяет создать класс **Car**, который в данный момент не имеет никаких атрибутов или методов. Вы можете добавить их позже.

2.Конструктор и метод __init__:

Конструктор - это специальный метод класса, который вызывается при создании нового объекта данного класса. В Python, конструктор называется **__init__**, и его целью является инициализация атрибутов объекта.

Метод __init__:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
```

В этом примере __init__ принимает три аргумента: make, model и year, и инициализирует атрибуты make, model и year объекта Car. Конструктор вызывается автоматически при создании объекта класса и позволяет устанавливать начальные значения атрибутов.

Инициализация атрибутов относится к процессу установки начальных значений для атрибутов объекта при создании этого объекта. Атрибуты представляют переменные, которые хранят данные объекта класса и позволяют хранить информацию о состоянии объекта. Процесс инициализации атрибутов часто выполняется с использованием конструктора класса.

3. Создание объекта:

Объект - это конкретный экземпляр класса, содержащий данные и способный выполнять методы, определенные в классе. Создание объекта в Python осуществляется путем вызова конструктора класса.

Пример создания объекта:

```
my_car = Car("Toyota", "Camry", 2023)
```

Здесь мы создаем объект my_car класса Car, и конструктор __init__ инициализирует атрибуты make, model и year объекта.

Эти три шага (определение класса, конструктор и создание объектов) являются основой ООП в Python и позволяют создавать абстрактные структуры, которые представляют реальные объекты или концепции в вашей программе. Классы и объекты помогают организовать и управлять данными и функциональностью вашей программы более структурированно и эффективно.

4. Атрибуты и методы:

Атрибуты - это переменные, которые хранят данные объекта, а методы - это функции, которые определяют поведение объекта. Они могут быть объявлены внутри класса.

Давайте рассмотрим пример класса с атрибутами и методами:

```python
class Car:
    def __init__(self, make, model, year):
        # Конструктор класса, инициализирующий атрибуты объекта
        self.make = make  # Производитель
        self.model = model  # Модель
        self.year = year  # Год выпуска
        self.mileage = 0  # Пробег, начальное значение - 0

    def start(self):
        # Метод, который запускает автомобиль
        return f"Автомобиль {self.year} {self.make} {self.model} был запущен."

    def drive(self, distance):
        # Метод, который изменяет пробег автомобиля
        self.mileage += distance  # Увеличиваем пробег
        return f"Проезжено {distance} миль."

    def honk(self):
        # Метод, который издает звук сигнала
        return "Гудок! Гудок!"

# Создаем объект класса Car
my_car = Car("Toyota", "Camry", 2023)

# Доступ к атрибутам и вызов методов объекта
print(my_car.make)  # Выводит: "Toyota"
print(my_car.year)  # Выводит: 2023

print(my_car.start())  # Запускает метод и выводит: "Автомобиль 2023 Toyota Camry был запущен."
print(my_car.drive(50))  # Вызывает метод и выводит: "Проезжено 50 миль."
print(my_car.honk())  # Вызывает метод и выводит: "Гудок! Гудок!"
print(f"Пробег: {my_car.mileage} миль")  # Выводит: "Пробег: 50 миль"
```

В этом примере класс **Car** имеет атрибуты (**make, model, year и mileage**) и методы (**start, drive и honk**):

Атрибуты (**self.make, self.model, self.year и self.mileage)**: Они представляют данные, принадлежащие объекту. В конструкторе __init__, атрибуты **make, model и year** инициализируются значениями, переданными при создании объекта. mileage инициализируется нулевым значением.

Методы **(start, drive и honk**): Методы выполняют операции с данными объекта. Метод start возвращает строку, сообщающую, что машина была запущена. Метод drive увеличивает mileage на указанное расстояние и возвращает сообщение о поезде. Метод honk возвращает строку, представляющую звук автомобильного гудка.

После создания объекта **my_car** класса **Car**, мы получаем доступ к его атрибутам и вызываем его методы, как показано в примере. Это демонстрирует, как класс Car может быть использован для создания объектов, представляющих конкретные автомобили, с различными характеристиками и возможностями.

5. Специальный атрибут self:

self - это специальный атрибут, который представляет текущий объект. Он используется для доступа к атрибутам и методам объекта внутри класса.

В методах класса, вы обычно будете использовать self для доступа к атрибутам объекта, например, self.attribute1. self необходим для указания, что вы обращаетесь к атрибутам и методам объекта, а не к локальным переменным функции.

Это основные концепции классов и объектов в Python. Классы позволяют создавать шаблоны для объектов, и объекты представляют конкретные экземпляры класса с уникальными данными и поведением.

Тема №8. Наследование и полиморфизм
Наследование:Определение наследования.Создание подклассов и суперклассов.Иерархия наследования.
Полиморфизм:Полиморфизм и переопределение методов.Абстрактные классы и методы.

Введение в исследование и полиморфизм в ООП:

Наследование - это один из ключей ООП, который позволяет создавать новые классы на основе существующих. В этом механизме существует класс-родитель (суперкласс или базовый класс), который может передавать свои свойства и методы другим классам (подклассам или производным классам). Наследование позволяет предотвратить повторное использование кода, а также создать иерархию классов для объектов организации.

Полиморфизм - это концепция, которая позволяет реагировать разным объектам по одному и тому же методу или оператору, специфичному для каждого объекта способом. Это позволяет более гибко работать с объектами и их методами в зависимости от их типа. Полиморфизм подразумевает переопределение методов в подклассах, чтобы они могли работать согласованно с методами суперкласса.

Теперь давайте рассмотрим, как эти концепции применяются в Python:

Python и наследование:

В классах Python можно изучать свойства и методы других классов. Это уточнение путем определения подклассов с использованием ключевого слова **class** и указания суперкласса в скобках. Наследование позволяет создавать иерархию классов для кода логической организации и переиспользования функциональности.

Пример на Python:

```
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

# Создаем объекты
dog = Dog("Buddy")
# Вызываем метод speak для объекта dog и выводим результат
print(dog.speak())  # Вывод: "Buddy says Woof!"
```

Animal- это базовый класс, который имеет конструктор **__init__**, который инициализирует атрибут **name** объекта. Этот класс служит в качестве суперкласса (или родительского класса) для других классов, которые могут иметь общие характеристики с животными.

Dog- это подкласс, который на **Animal.** У класса **Dog** есть собственный

speak,**speak** переопределение в классе **Dog**.

Python и полиморфизм:

В Python **полиморфизм** происходит путем переопределения методов в подклассах. Если подкласс переопределяет метод, находящийся в суперклассе, объекты этого подкласса могут использовать новый метод. Это позволяет объектам разных типов реагировать на методы, специфичные для своего типа.

Пример на Python:

```
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"
```

Здесь класс **Cat** переопределяет метод **speak**, и объекты класса **Cat** будут использовать его собственное представление.

Таким образом, исследование и полиморфизм являются мощными концепциями ООП, которые позволяют создавать более гибкие и легко сопровождаемые программы на Python и других языках.

Пример более точного понимания темы:

```
class Animal:
   def __init__(self, name):
        self.name = name

class Dog(Animal):
   def speak(self):
       return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Создаем объекты
animal = Animal("Generic Animal")
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Вызываем метод speak для разных объектов и выводим результат
print(dog.speak())  # Вывод: "Buddy says Woof!"
print(cat.speak())  # Вывод: "Whiskers says Meow!"
```

Этот код иллюстрирует использование наследования и полиморфизма в Python, где подкл. **Dog** и **Cat** у них есть собственный способ реализации **speak**, несмотря на то, что они являются наследниками класса **Animal**:

1.class Animal: Здесь создан класс **Animal**, который служит в качестве базового класса.

2.def __init__(self, name): Это метод конструктора **(__init__)** для класса **Animal**. Он инициализирует объект **Animal** с помощью аргумента **name**.

3.self.name = name: Внутри конструктора настроены атрибуты объекта **self.name rav.name.**

4.class Dog(Animal): Здесь создается подкласс **Dog**, который наследует от базового класса **Animal.**

5.def speak(self): Это метод **speak** для класса `**Dog**. Он использует символ, передающий звук издаваемой собаки.

6.return f"{self.name} says Woof!": В методе **speak** формирования строки, используя имя объекта **self.name**, и добавляется **"говорит Гав!"**.

7.class Cat(Animal): Здесь появился еще один подкласс **Cat**, который также наследует от базового класса **Animal**.

8.def speak(self): Это метод **speak** для класса **Cat.** Он держит в руках символ, передающий звук, издаваемый кот.

9.return f"{self.name} says Meow!": В методе speakформирования строки, используя имя объекта **self.name**, и добавляется **"говорит Мяу!"**.

10.Создание объектов:

animal = Animal("Generic Animal"): Создается объект **animal** класса **Animal** с именем «**Generic Animal**».

dog = Dog("Buddy"): Создается объект **dog** класса **Dog** с именем **"Buddy".**

cat = Cat("Whiskers"): Создается объект **cat** класса **Cat** с именем **"Whiskers"**.

11.Вызов методов и выводы результатов:

print(dog.speak()): Вызов **speak** для объекта **dog** и выводится результат, который в данном случае будет «**Приятель говорит Гав!**».

print(cat.speak()): Вызывается метод **speak** для объекта **cat**, и выводится

Этот код иллюстрирует использование наследования и полиморфизма в Python, где подкл.**Dog** и **Cat** у них есть собственный способ реализации **speak**, несмотря на то, что они являются наследниками класса **Animal**.

Абстрактные классы и методы — это концепция объектно-ориентированного программирования, которая используется для создания шаблонов и определений интерфейсов, без обеспечения реализации. В языке Python абстрактные классы и методы реализуются с помощью модуляций **abc(абстрактных базовых классов).**

Последний класс - это класс, который не предполагает создание экземпляров и служащих только в качестве шаблона для подклассов. Он может сохранять абстрактные методы, которые должны быть реализованы в подклассах. Абстрактные классы обычно определяют общее поведение групп подклассов.

Квадратный метод — это метод, который объявляется в абстрактном классе, но не обеспечивает реализацию. Его необходимо обязательно переопределить в подклассе. Абстрактные методы используются для определения интерфейса, который должен поддерживать все подклассы.

Пример использования абстрактных классов и методов в Python с помощью модуля abc:

```python
from abc import ABC, abstractmethod

class Shape(ABC):  # Объявление абстрактного класса
       def area(self):  # Объявление абстрактного метода
        pass

class Circle(Shape):  # Подкласс, который наследует от абстрактного класса
    def __init__(self, radius):
        self.radius = radius

    def area(self):  # Реализация абстрактного метода
        return 3.14 * self.radius ** 2

# Создание объекта подкласса и вызов метода area
circle = Circle(5)
print(circle.area())  # Вывод: 78.5
```

В этом определении Shape— абстрактный класс с абстрактным методом area, который должен быть реализован в подклассе. Circle- подкласс Shape, который реализует метод area. Когда объект circle сталкивается с методом area, он возвращает площадь круга.

Методическая разработка аудиторных форм работы (Краткое содержание практических занятий)

Практическая работа №1

Программирование функции в Python

Цель работы: получение практических навыков программирования в создании и использовании функций.

Общие положения

Функция – особым образом сгруппированный набор команд, которые выполняются последовательно, но воспринимаются как единое целое. При этом функция может возвращать (или не возвращать) свой результат.

Функция это блок организованного, многократно используемого кода, который используется для выполнения конкретного задания. Функции обеспечивают лучшую модульность приложения и значительно повышают уровень повторного использования кода.

Функции – это такие участки кода, которые изолированы от остальной программы и выполняются только тогда, когда вызываются. Вы уже встречались с функциями sqrt(), len() и print(). Они все обладают общим свойством: они могут принимать параметры (ноль, один или несколько), и они могут возвращать значение (хотя могут и не возвращать). Например, функция sqrt() принимает один параметр и возвращает значение (корень числа). Функция print() принимает переменное число параметров и ничего не возвращает.

Создание функции

Для того чтобы использовать какую-нибудь собственную функцию, вначале необходимо ее объявить (создать).

Блок функции начинается с ключевого слова def, после которого следуют название функции и круглые скобки ( ). Любые аргументы, которые принимает функция, должны находиться внутри этих скобок. После скобок идет двоеточие и с новой строки с отступом начинается тело функции.

def <имя функции>(аргументы):

<тело функции>

Пример функции в Python:

def my_function(argument):

print (argument)

После функции до кода, который находится вне функции, необходимо делать отступ в две пустые строки для повышения читаемости кода. Если у вас есть несколько функций в одном файле, между кодом одной и сигнатурой другой функции тоже надо оставлять две пустые строки.

Вызов функции

После создания функции, ее можно исполнять, вызывая из другой функции или напрямую из оболочки Python. Для вызова функции следует ввести ее имя и добавить в скобках аргументы.

<имя функции>(аргументы)

Обращение к ранее объявленной функции с целью выполнения ее команд называется вызовом.

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания. Аргументы (параметры) могут изменять поведение функции.

Аргументы функции в Python

Вызывая функцию, мы можем передавать ей следующие типы аргументов:

1. Обязательные аргументы (Required arguments)
2. Аргументы-ключевые слова (Keyword argument)
3. Аргументы по умолчанию (Default argument)
4. Аргументы произвольной длины (Variable-length argumens)

Обязательные аргументы функции. Если при создании функции мы указали количество передаваемых ей аргументов и их порядок, то и вызывать ее мы должны с тем же количеством аргументов, заданных в нужном порядке.

Например:

```
1  def bigger(a,b):#В описании функции указано, что она принимает 2 аргумен
2      if a>b:
3          print(a)
4      else:
5          print(b)
6
7
8  bigger(5,6)#Корректное использование функции
```

Аргументы - ключевые слова используются при вызове функции. Благодаря ключевым аргументам, вы можете задавать произвольный (то есть не такой, каким он описан при создании функции) порядок аргументов.

Например:

```
1  def person(name,age):
2      print(name,"is",age,"years old")
3
4
5  person(name="Chika",age=23)
```

Аргументы, заданные по умолчанию. Аргумент по умолчанию, это аргумент, значение для которого задано изначально, при создании функции. Например:

```
def space(planet_name,center='Star'):
        print(planet_name,"is orbiting a",center)


space("Mars")#В результате получим: Mars is orbiting a Star
```

```
space("Mars","Black Hole")#В результате получим: Mars is orbiting a Black Hole
```

Аргументы произвольной длины. Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции. В этом случае следует использовать аргументы произвольной длины. Они задаются произвольным именем переменной, перед которой ставится звездочка (*). Например:

```
def unknown(*args):
    for argument in args:
        print(argument)


unknown("hello","word")     # напечатает оба слова, каждое с новой строки
unknown(1,2,3,4,5)          # напечатает все числа, каждое с новой строки
unknown()                   # ничего не выведет
```

Ключевое слово return

Выражение **return** прекращает выполнение функции и возвращает указанное после выражения значение. Выражение **return** без аргументов это то же самое, что и выражение return None. Соответственно, теперь становится возможным, например, присваивать результат выполнения функции какой-либо переменной. Например:

```
def bigger(a,b):
    if a>b:
        return a    # Если a больше чем b, то возвращаем a и прекращаем выполнение функции
    return b   # Незачем использовать else. Если мы дошли до этой строки, то b, точно не меньше чем a


num = bigger(23,24) # присваиваем результат функции bigger переменной num
```

Пустая функция.

Чтобы создать пустую функцию, нужно в её теле использовать оператор заглушки pass. Тогда она будет существовать и не выполнять никаких действий. Такие функции могут использоваться для различных специфических задач, например, при работе с классами, асинхронной отправкой форм.

def example():

pass

Область видимости переменных

В Python две базовых области видимости переменных:

- локальные переменные - переменные, создаваемые внутри функций, недоступны извне и существуют только внутри функции.
- глобальные переменные – переменные, создаваемые вне функции, могут быть доступны из функций.

Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:

```
age = 44  #глобальная переменная age

def info():
    print(age)  # печатаем глобальную переменную age

def local_info():
    age = 22    #создаем локальную переменную age
    print(age)

info()          #напечатает 44
local_info()    #напечатает 22
```

Важно помнить, что для того чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. Однако, если перед нами стоит задача изменить глобальную переменную внутри функции - необходимо использовать ключевое слово **global.**

Например:

```
age = 13  #глобальная переменная age

def get_older():
    global age  # функция изменяющая глобальную переменную
    age+=1

print(age)  #напечатает 13
get_older() #увеличиваем age на 1
print(age)  #напечатает 14
```

Рекурсией называется процесс, когда функция вызывает саму себя, а сама функция называется рекурсивной.

Классическим примером рекурсии может послужить функция вычисления факториала числа.

Напомним, что факториалом числа, например, 5 является произведение всех натуральных (целых) чисел от 1 до 5.

То есть, 1 * 2 * 3 *4 * 5

```
def f(num):
    if num==0:
        return 1    #факториал нуля равен единице
    return f(num-1)*num


print(f(5))         # выведет число 120
```

Рекурсию рекомендуется использовать только там, где это действительно необходимо. Интерпретатор Python автоматически выделяет память для

выполняющейся функции, если вызовов самой себя будет слишком много, это приведёт к переполнению стека и аварийному завершению программы.

Функции высшего порядка

Функции, которые принимают или возвращают другие функции, называются **функциями высшего порядка.**

Функция filter

Функция **filter** принимает критерий отбора элементов, а затем сам список элементов. Возвращает она список из элементов, удовлетворяющих критерию.

Чтобы этой функцией воспользоваться, нужно сообщить функции **filter** критерий, который говорит, брать элемент в результирующий список или нет. Давайте напишем простую функцию, которая проверяет, что слово длиннее шести букв, и затем отберем с ее помощью длинные слова.

```
def is_word_long(word):
    return len(word)>6
words = ['В','новом','списке','останутся','только','длинные','слова']
for word in filter(is_word_long,words):
    print(word)
```

останутся
длинные

С методом filter вам не нужно вручную создавать и заполнять список, достаточно указать условие отбора.

Лямбда-функции

Часто в качестве аргумента для функций высшего порядка мы хотим использовать совсем простую функцию. Причем нередко такая функция нужна в программе только в одном месте, поэтому ей необязательно даже иметь имя.

Такие короткие безымянные (анонимные) функции можно создавать инструкцией

lambda <аргументы>: <выражение>

Такая инструкция создаст функцию, принимающую указанный список аргументов и возвращающую результат вычисления выражения.

В языке Python тело лямбда-функции имеет ровно одно выражение. Скобки вокруг аргументов не пишутся, аргументы от выражения отделяет двоеточие.

Теперь мы можем записать функцию, проверяющую длину слова, следующим образом:

```
lambda word: len(word)>6
words = ['В','новом','списке','останутся','только','длинные','слова']
long_words = list(filter(lambda word: len(word)>6,words))
print(long_words)
```

Лямбда-функция – полноценная функция. Ее можно использовать в составе любых конструкций. Например, созданную лямбда-функцию, можно присвоить какой-либо переменной:

```
add = lambda x, y:
    x + y
add(3, 5) # 8
add(1, add(2, 3)) # 6
```

Порядок выполнения работы

Задание 1. Напишите программу, которая вычислит значение выражения. Значение выражения вида $\sqrt{a^2 + b^2 + cos^3(ab)}$ вычислять с использованием функции.

$$L=\frac{\sqrt{x^2+z^2+cos^3(xz)}+\sqrt{y^2+x^2+cos^3(yx)}}{\sqrt{z^2+y^2+cos^3(zy)}}.$$

Задание 2. Напишите программу, которая вычислит значение выражения. Значение факториала вычислять с использованием функции (Факториалом числа является произведение всех натуральных (целых) чисел).

$$C = \frac{n!}{m!(n-m)!}$$

Задание 3. Составить программу, согласно полученному варианту задания, используя функции высшего порядка. Список чисел вводится пользователем.

Вариант	Задание
1	Напишите программу, которая подсчитает и выведет сумму кубов всех трехзначных чисел, делящихся на 8.
2	Напишите программу, которая подсчитывает и выводит сумму кубов отрицательных чисел списка
3	Напишите программу для печати четных чисел из заданного списка.
4	Напишите программу, которая извлекает из списка числа, делимые на 17
5	Напишите программу, которая извлекает из списка трехзначные четные числа, делимые на 6.
6	Напишите программу, которая извлекает из списка нечетные числа, делимые на 11.
7	Напишите программу, которая подсчитает и выведет произведение квадратов всех однозначных чисел, делящихся на 2.
8	Напишите программу для печати нечетных чисел из заданного списка
9	Напишите программу, которая подсчитает и выведет сумму квадратов всех двузначных чисел, делящихся на 7.
10	Напишите программу, которая подсчитает и выведет сумму квадратов положительных чисел списка.

Практическая работа №2
Работа над созданием модулей

Цель работы: получение практических навыков программирования в создании модуля.

Оборудование: ПЭВМ.

Контрольные вопросы:

1.Где находятся все модули Python?

2.Какие виды модулей есть в Python?

3.В чем суть модульного программирования?

4.Чем модуль отличается от подпрограмм?

5.Как использовать модули в Python?

Теоретические сведения :

Модули в Python - это просто файлы Python с расширением .py. Модуль оформляется в виде отдельного файла с исходным кодом.

Имя модуля будет именем файла. Модуль Python может иметь набор функций, классов или переменных, определенных и реализованных.

Создание модуля

Чтобы создать свой модуль в Python достаточно сохранить код в файл с расширением.py Теперь он доступен в любом другом файле.

Например, создадим файл mymodule.py, в которой определим какие-нибудь функции:

```
def hello():
    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b
```

Теперь в этой же папке создадим другой файл, например, main.py:

```
import mymodle
mymodle.hello()
print(mymodle.fib(10))
```

Выведет:

```
Hello, world!
55
```

Модуль нельзя именовать также, как и ключевое, также имена модулей нельзя начинать с цифры и не стоит называть модуль также, как какую-либо из встроенных функций.

Порядок выполнения работы

Задание. Составить программу, согласно полученному варианту задания.

Вариант	**Задание**
1	Список A содержит N чисел. Перепишите из списка A в список B только те элементы, значения которых не равны заданному значению K. Назначение функции: переписывание

	из одного списка в другой только тех элементов, которые не совпадают с заданным значением. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
2	Список A содержит N упорядоченных по возрастанию чисел. Вставьте в список некоторое значение K так, чтобы упорядоченность списка не нарушилась. Назначение функции: вставка заданного значения в упорядоченный список. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
3	Список A содержит N элементов, значения которых не определены. Организуйте запрос: «Сколько элементов списка следует заполнить?» Заполните список заданным числовым значением K. Назначение функции: заполнение заданного числа элементов заданным значением. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
4	Список A содержит N чисел. Найдите количество элементов списка, значения которых превышают заданное значение K. Назначение функции: подсчет количества элементов, превышающих заданное значение. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
5	Список A содержит N чисел. Найдите сумму значений элементов списка, меньших заданного значения K. Назначение функции: суммирование элементов, значения которых меньше заданного. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
6	Список A содержит N чисел. Найдите произведение значений элементов списка. Назначение функции: вычисление произведения элементов списка. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
7	Список A содержит N чисел. Удалите из списка значение с порядковым номером K. Назначение функции: удаление из списка значения с заданным порядковым номером. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
8	Список A содержит N чисел. Значения всех элементов

	списка, отличающихся от заданного значения К, замените другим заданным значением М. Назначение функции: замена значений элементов списка. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
9	Список А содержит N чисел. Найдите в нем элемент с наибольшим значением. Назначение функции: поиск в списке элемента с наибольшим значением. Оформите созданную функцию в виде программного модуля и подключите его к основной программе.
10	Список А содержит N чисел. Разработайте программу, с помощью которой можно определить количество наибольших элементов в нем. Назначение 1 функции: нахождение максимального элемента. Назначение 2 функции: подсчет количества максимальных элементов. Оформите созданные функции в виде программного модуля и подключите его к основной программе.

Практическая работа №3
Создание и запись в текстовый файл.Чтение текстового файла и обработка данных

Целью работы: является владение навыками создания, записи и чтения текстовых файлов на языке программирования, а также обработка данных, хранящихся в таких файлах.

Контрольные вопросы:

1.Как создать текстовый файл на Python и записать в него текст?

2.Как открыть существующий текстовый файл для чтения в Python?

3.Как считать критерием текстовый файл в переменную в Python?

4.Как записать данные в конец существующего текстового файла без удаления существующего изменения?

5.Как обработать каждый символ в текстовом файле и настроить режим работы?

Теоретические сведения:

Текстовый файл - это файл, который содержит текстовую информацию, представленную в виде символов последовательности.
Операции с текстовыми файлами включают создание файлов, запись данных в них, чтение данных из них и обработку информации, хранящейся в файлах.

Для работы с текстовыми файлами на языке программирования обычно используются стандартные библиотеки или модули, обеспечивающие соответствующие функции.

Открытие файлов: Для открытия файлов в Python используется функция **open().** Она принимает два основных аргумента: имя файла и режим доступа (например, «r» для чтения, «w» для записи, «a» для добавления данных в конец файла и другие). Пример: file = open("example.txt", "r").

Закрытие файлов: После завершения работы с файлом важно закрыть его с помощью метода close(), чтобы уменьшить ресурс и убедиться, что изменения сохранены. Пример: file.close().

Запись данных: Для записи данных в файл используется метод write(). Он позволяет записывать текстовую информацию в файл. Пример: file.write("Привет, мир!").

Чтение данных : Для чтения данных из файла можно использовать методы read(), readline(), или readlines(). read()считывает весь файл, readline()читает один текст, a readlines()читает все строки и возвращает их в виде списка.

Исключения : при работе с файлами важны возможные исключения, такие как FileNotFoundError(если файл не существует), PermissionError(если нет правого доступа) и другие. Рекомендуется использовать блоки try...exceptдля исключений.

Контекстные менеджеры : Для гарантированного закрытия файла после использования рекомендуется использовать контекстные менеджеры с оператором with.

Варианты заданий:

№	Задание
1	Создание и запись в файл: 1.Создайте текстовый файл с именем "example.txt". 2.Запишите в этот файл следующий текст: "Привет, мир!" 3.Закрыть файл.

2	Чтение файла и вывод данных: 1.Откройте файл «example.txt» для чтения. 2.Создайте важный файл и выведите его на экран. 3.Закрыть файл.
3	Обработка данных из файла: 1.Откройте файл «data.txt» для чтения. Файл содержит числа, разделенные пробелами. 2.Прочитайте числа из файла и вычислите их сумму. 3.Вы создаете структуру на экране. 4.Закрыть файл.
4	Запись данных в файл: 1.Создайте текстовый файл "names.txt". 2.Запросите у пользователя ввод нескольких имен (каждое имя в новой строке) и сохраните их в файле. 3.Закрыть файл.
5	Поиск слов в файле: 1.Откройте файл «text.txt» для чтения. Файл содержит текст. 2.Запросите у пользователя слово для поиска. 3.Проверьте, есть ли это слово в тексте файла, и вы введете сообщение.
6	Копирование файла статистики: 1.Откройте файл «source.txt» для чтения. 2.Откройте файл «destination.txt» для записей. 3.Скопируйте требования "source.txt" в "destination.txt". Закройте оба файла.
7	Подсчет слов в тексте: 1.Откройте файл «text.txt» для чтения. 2.Подпишите количество слов в тексте (слова разделяются пробелами или другими разделителями). 3.Вы вводите количество слов на экране. Закрыть файл.
8	Фильтрация данных в файл: 1.Откройте файл «data.txt» для чтения. Файл содержит числа, разделенные пробелами. 2.Прочтите числа из файла и создайте новый файл "filtered_data.txt", в котором будут только четные числа. 3.Закройте оба файла.
9	Замена текста в файле: 1.Откройте файл «text.txt» для чтения. 2.Замените все вхождения слова «старое» на «новое» и сохраните

	изменения в файле. 3.Закрыть файл.
10	Объединение файлов изменений: 1.Откройте файлы «file1.txt» и «file2.txt» для чтения. 2.Считайте важные оба файла и внесите их в новый файл "merged.txt". 3.Закройте все файлы.