**Материалы для подготовки к экзамену по ОП.14 ОООП.**

КМПО РАНХиГС, 41ИС-21, 42ИС-21, 43ИС-21, 44ИС-21

Преподаватель: Гусятинер Л. Б., 03.12.2024

**Инструкция по выполнению задания:**

*Общее.*

на оценки «3» и «4» разрешается:

- при выполнении задания 1 - использовать конспекты и отчёты по лабораторным работам, выполненные на очных парах;

- задание 4 – дополнительное, на оценку «5».

*Вариатив.*

*При выполнении задания 3:*

на оценку "5":

- необходимо использовать PyQT;

- необходимо использовать СУБД MySQL или PostgreSQL;

- в случае невозможности сравнения возбуждать исключение;

на оценки "3" или "4" разрешается

- использовать встроенные и / или библиотечные контейнерные классы;

– не использовать сеттеры и геттеры.

***Примечания по использованию языков и инструментов***.

1. Основные языки: C#, Java, Python, разрешенные на демо-экзамене.
2. На очном экзамене работа исключительно на компьютерах колледжа
3. В связи с тем, что ряд студентов по предварительной установке изучали Си++, принимаются решения на этом языке.

**Задание 1.**

**- "Утиная" типизация**

в Python означает, что вместо проверки типа объекта (например, "это строка или число?"), мы просто используем объект, исходя из того, какие действия он может выполнять. Простыми словами: в Python важно не то, к какому классу относится объект, а то, что он умеет делать. Вы можете передавать любой объект, если он поддерживает нужные методы или свойства.

Пример:

def make\_sound(animal):

print(animal.make\_sound())

class Duck:

def make\_sound(self):

return "Кря-кря"

class Dog:

def make\_sound(self):

return "Гав-гав"

duck = Duck()

dog = Dog()

make\_sound(duck) # Вывод: Кря-кря

make\_sound(dog) # Вывод: Гав-гав

Функция make\_sound не проверяет, является ли animal уткой или собакой.

Она просто вызывает метод make\_sound() объекта.

Почему это полезно?

Не нужно тратить время на проверки типов (isinstance или type).

Код становится более универсальным и читаемым.

Легче добавлять новые классы, которые соответствуют нужному поведению.

**- Абстрактные типы данных и классы. Сравнение**

Абстрактный тип данных - Концепция, определяющая поведение данных и операций над ними (интерфейс). Реализуется через интерфейсы, модули или абстрактные базовые классы (ABC). Коллекции Python: list, dict, set реализуют интерфейс АТД. Фокус на описании операций, а не на данных.

Классы python - Конкретная реализация, объединяющая данные и методы. Используются классы с помощью ключевого слова class. Описание и реализация структуры объекта (данные и методы). Включает данные (атрибуты) и методы для их обработки.

Пример: Абстрактный тип данных vs Класс

Абстрактный тип данных (АТД):

Используем модуль abc для задания интерфейса.

from abc import ABC, abstractmethod

# Абстрактный тип данных (интерфейс для стека)

from abc import ABC, abstractmethod

# Абстрактный тип данных (интерфейс для стека)

class AbstractStack(ABC):

@abstractmethod

def push(self, item):

"""Добавить элемент в стек"""

pass

@abstractmethod

def pop(self):

"""Удалить и вернуть последний элемент из стека"""

pass

@abstractmethod

def is\_empty(self):

"""Проверить, пуст ли стек"""

pass

Этот код говорит: "Если вы хотите создать стек, он должен уметь выполнять эти три операции (push, pop, is\_empty)". Но никакой реализации здесь нет.

from abc import ABC, abstractmethod

# Абстрактный тип данных (интерфейс для стека)

class AbstractStack(ABC):

@abstractmethod

def push(self, item):

"""Добавить элемент в стек"""

pass

@abstractmethod

def pop(self):

"""Удалить и вернуть последний элемент из стека"""

pass

@abstractmethod

def is\_empty(self):

"""Проверить, пуст ли стек"""

pass

Этот код говорит: "Если вы хотите создать стек, он должен уметь выполнять эти три операции (push, pop, is\_empty)". Но никакой реализации здесь нет.

Отличия в Python:

АТД в Python чаще всего реализуются с использованием модуля abc (абстрактных базовых классов). Они задают требования к будущим классам.

Классы — это конкретная реализация, которая может (но не обязана) следовать интерфейсу АТД.

Важное:

АТД задают контракт, а классы реализуют этот контракт.

Использование абстрактных базовых классов в Python делает код более структурированным, что особенно полезно в крупных проектах.

**- Внутренние (вложенные) классы**

это классы, определённые внутри другого класса. Они обычно используются, когда логически связанный класс имеет смысл только в контексте внешнего класса.

### Зачем нужны вложенные классы?

1. **Логическая группировка:** Если класс используется только внутри другого класса.
2. **Инкапсуляция:** Вложенный класс недоступен вне внешнего класса.
3. **Сокращение области видимости:** Упрощает структуру программы, ограничивая использование вложенного класса

Простой пример: Вложенный класс

python

Копировать код

class OuterClass:

def \_\_init\_\_(self, name):

self.name = name

class InnerClass:

def \_\_init\_\_(self, value):

self.value = value

def display(self):

return f"Inner value: {self.value}"=

# Использование

outer = OuterClass("Outer")

inner = OuterClass.InnerClass(42)

print(inner.display()) # Вывод: Inner value: 42

Пример с зависимостью вложенного класса от внешнего:

python

Копировать код

class OuterClass:

def \_\_init\_\_(self, name):

self.name = name

class InnerClass:

def \_\_init\_\_(self, outer\_instance, value):

self.outer\_name = outer\_instance.name # Доступ к внешнему классу

self.value = value

def display(self):

return f"Outer name: {self.outer\_name}, Inner value: {self.value}"

# Использование

outer = OuterClass("MainClass")

inner = outer.InnerClass(outer, 99)

print(inner.display()) # Вывод: Outer name: MainClass, Inner value: 99

Объяснение:

Внутренний класс InnerClass может ссылаться на данные из внешнего класса через переданный экземпляр outer\_instance.

### Когда использовать:

Если класс тесно связан с другим классом.

Для упрощения структуры и группировки логически связанных классов.

Если вложенный класс не должен быть доступен в глобальной области видимости.

**- Генерация исключений**

Генерация исключений в Python позволяет явно "выбрасывать" ошибки в программе с помощью ключевого слова raise. Это полезно, когда нужно остановить выполнение кода из-за неправильного состояния программы или некорректных данных.

Синтаксис

python

Копировать код

raise ExceptionType("Сообщение об ошибке")

ExceptionType — тип исключения, например, ValueError, TypeError, KeyError и т. д.

"Сообщение об ошибке" — текст, который будет отображаться, когда исключение будет сгенерировано.

Простой пример: Генерация исключения

python

Копировать код

def divide(a, b):

if b == 0:

raise ZeroDivisionError("Деление на ноль невозможно")

return a / b

try:

result = divide(10, 0)

except ZeroDivisionError as e:

print(f"Ошибка: {e}")

**- Инкапсуляция и области видимости. Защита на уровне объекта и класса.**

В Python инкапсуляция — это механизм сокрытия деталей реализации и предоставления доступа к данным и методам через специально определённый интерфейс. Инкапсуляция помогает защищать данные от некорректного использования и разделять внутреннюю реализацию объекта от внешнего взаимодействия.

Защита на уровне объекта:

Используется для ограничения доступа к данным экземпляра.

Пример:

python

Копировать код

class Account:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Приватный атрибут

def get\_balance(self):

return self.\_\_balance # Доступ через метод

Защита на уровне класса:

Используется для ограничения доступа к данным, общим для всех экземпляров.

Пример:

python

Копировать код

class Singleton:

\_\_instance = None # Приватный атрибут класса

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if cls.\_\_instance is None:

cls.\_\_instance = super().\_\_new\_\_(cls)

return cls.\_\_instance

Главное отличие:

Уровень объекта: Защищает данные конкретного экземпляра.

Уровень класса: Защищает данные, относящиеся ко всему классу.

Инкапсуляция помогает контролировать доступ к данным и улучшает безопасность кода.

Области видимости в Python

Публичные (public): Доступны везде, как внутри класса, так и за его пределами.

Защищённые (protected): Доступны внутри класса и в его подклассах.

Приватные (private): Доступны только внутри класса, скрыты от внешнего мира.

Python не имеет жёсткой системы ограничения доступа (как, например, Java или C++), но использует соглашения об именовании:

public: Имя обычное (например, variable).

protected: Имя начинается с одного подчёркивания (например, \_variable).

private: Имя начинается с двух подчёркиваний (например, \_\_variable).

class Example:

def \_\_init\_\_(self):

self.public = "Публичный" # Доступен всем

self.\_protected = "Защищённый" # Доступен только внутри класса и его подклассов

self.\_\_private = "Приватный" # Доступен только внутри класса

def get\_private(self):

return self.\_\_private # Доступ через метод

**- История развития объектно-ориентированных языков, их классификация и архитектура. Основные понятия объектно-ориентированного анализа. Объектная декомпозиция.**

1. *Ранний этап (1960-е):*

Появление концепции "объекта":

В языке Simula (1967) впервые использованы объекты и классы. Он был разработан для моделирования реальных процессов.

Основы положили начало идее инкапсуляции, наследования и полиморфизма.

1. 1970-е годы:

Создание Smalltalk (1972): первый полностью объектно-ориентированный язык.

Внёс важные концепции: объекты как "единицы взаимодействия", сообщения между объектами.

1. 1980-е годы:

Расширение объектно-ориентированных концепций:

C++ (1983) — добавил объектно-ориентированные возможности в язык C.

Objective-C (1984) — объединение идей Smalltalk с C.

Введение объектного подхода в программирование для больших систем.

1. *1990-е годы:*

Появление массовых языков:

Java (1995) — "пиши один раз, используй везде".

Python (1991) — простой синтаксис с поддержкой объектно-ориентированного стиля.

Ruby (1995) — ориентирован на удобство разработки.

Распространение объектно-ориентированного подхода в индустрии.

1. 2000-е годы и далее:

Развитие современных языков:

C# (2000) — объектно-ориентированный язык от Microsoft.

Swift (2014) — современный язык для разработки приложений Apple.

Интеграция объектной парадигмы с другими (функциональной, декларативной).

Классификация объектно-ориентированных языков

1. По уровню абстракции:

* Низкоуровневые (например, C++): позволяют управлять памятью и системными ресурсами.
* Высокоуровневые (например, Python, Java): сосредоточены на удобстве использования.

1. По парадигмам:

* Чисто объектно-ориентированные: Smalltalk, Ruby.
* Смешанные: Python, C++, Swift (поддерживают ООП и другие парадигмы).

1. По применению:

* Системные: C++, Objective-C.
* Прикладные: Java, Python, C#.
* Скриптовые: Ruby, Python.

Архитектура объектно-ориентированных языков

1. Объекты:

Содержат данные (атрибуты) и поведение (методы).

Инкапсулируют данные, защищая их от прямого доступа.

1. Классы:

Шаблоны для создания объектов.

Определяют структуру и поведение объектов.

1. Механизмы:

Наследование: создание новых классов на основе существующих.

Полиморфизм: способность объектов разных классов обрабатывать одинаковые сообщения.

Инкапсуляция: скрытие деталей реализации.

Основные понятия объектно-ориентированного анализа (ООА)

Объект- сущность из реального мира, представленная в системе (например, клиент, заказ).

Объекты характеризуются атрибутами и операциями.

Класс - группа объектов с одинаковыми свойствами и поведением.

Наследование - Иерархическая организация классов. Например, "Автомобиль" → "Грузовик" и "Легковой".

Ассоциация и агрегация:

Ассоциация: связь между объектами (например, "Клиент" связан с "Заказом").

Агрегация: "часть-целое" (например, "Машина" состоит из "Двигателя").

Полиморфизм- Возможность замены объекта класса базового типа объектом класса-наследника.

Объектная декомпозиция

Процесс разбиения системы на объекты, которые взаимодействуют друг с другом.

Основная цель — представление системы в виде иерархии объектов, моделирующих реальный мир.

Этапы объектной декомпозиции:

Выделение объектов: анализ реальных сущностей (например, клиент, заказ, продукт).

Определение атрибутов: данные, которые описывают объект.

Определение методов: действия, которые объект может выполнять.

Установление связей между объектами: как объекты взаимодействуют.

Преимущества:

* Упрощение разработки больших систем.
* Повышение гибкости и повторного использования компонентов.
* Улучшение масштабируемости.

Пример объектной декомпозиции

Представим систему интернет-магазина:

Объекты:

Клиент, Заказ, Продукт, Корзина.

Атрибуты объектов:

Клиент: имя, email.

Заказ: номер заказа, статус.

Продукт: название, цена.

Методы:

Клиент: зарегистрироваться, авторизоваться.

Корзина: добавить товар, удалить товар.

Заказ: оплатить, отменить.

Связи:

Клиент связан с заказами (один клиент может иметь несколько заказов).

Заказ содержит продукты.

Итог: система строится из взаимосвязанных объектов, каждый из которых решает свою задачу.

**- Источники данных и компоненты доступа к базам данных.**

Источники данных и компоненты доступа к базам данных — это ключевые элементы, которые позволяют программам взаимодействовать с базами данных.

1. Источники данных

Источники данных — это места, где хранятся данные, с которыми будет работать приложение. Они могут быть представлены в виде:

Реляционных баз данных (MySQL, PostgreSQL, Oracle, SQL Server), где данные организованы в таблицы и связаны между собой.

Нереляционных баз данных (MongoDB, Redis, Cassandra), где данные хранятся в более гибких форматах, таких как JSON или key-value пары.

Файлы: текстовые файлы, CSV, XML или JSON файлы могут быть источниками данных, особенно для небольших приложений.

Веб-сервисы: RESTful или SOAP API, которые предоставляют данные через HTTP запросы.

2. Компоненты доступа к базам данных

Компоненты доступа к данным обеспечивают связь между приложением и источниками данных. Эти компоненты позволяют выполнять запросы, вставлять, обновлять или удалять данные из базы. Вот основные компоненты:

API (Application Programming Interface): Позволяет приложению взаимодействовать с базой данных или сервисом. Например:

ORM (Object-Relational Mapping): Это технология для работы с реляционными базами данных, которая позволяет представлять данные в виде объектов в программном коде.

import sqlite3

# Подключение к базе данных

conn = sqlite3.connect('example.db')

# Создание курсора для выполнения SQL-запросов

cursor = conn.cursor()

# Пример запроса

cursor.execute("SELECT \* FROM users")

# Вывод всех записей

rows = cursor.fetchall()

for row in rows:

print(row)

# Закрытие соединения

conn.close()

Основные этапы работы с базой данных:

* Подключение к базе данных.
* Выполнение запросов (например, SELECT, INSERT, UPDATE, DELETE).
* Обработка данных (получение результатов запросов).
* Закрытие соединения с базой данных.

- **Классы для работы с датой и временем**

Python содержит несколько инструментов, которые разработчик может использовать для работы с датой и временем.

Модуль datetime

* datetime.date
* datetime.timedelta
* datetime.datetime

1. datetime.date: Этот класс представляет дату (год, месяц, день) без времени.

import datetime

today = datetime.date.today()

print("Сегодня:", today)

custom\_date = datetime.date(2024, 12, 10)

print("Произвольная дата:", custom\_date)

1. datetime.time:

Этот класс представляет время (часы, минуты, секунды и микро-секунды) без привязки к конкретной дате.

import datetime

# Создание объекта времени

time\_obj = datetime.time(14, 30, 0) # 14:30:00

print("Время:", time\_obj)

# Время с микро-секундами

time\_with\_micro = datetime.time(14, 30, 0, 123456)

print("Время с микро-секундами:", time\_with\_micro)

1. datetime.datetime:

Этот класс сочетает в себе как дату, так и время (год, месяц, день, час, минута, секунда, микро-секунда).

import datetime

# Текущее время и дата

now = datetime.datetime.now()

print("Текущая дата и время:", now)

# Создание произвольной даты и времени

custom\_datetime = datetime.datetime(2024, 12, 10, 14, 30, 0)

print("Произвольная дата и время:", custom\_datetime)

# Форматирование даты и времени

formatted\_now = now.strftime("%Y-%m-%d %H:%M:%S")

print("Форматированная дата и время:", formatted\_now)

1. datetime.timezone

Этот класс используется для работы с часовыми поясами. Он позволяет создавать объекты с привязкой к конкретному часовому поясу.

import datetime

# Создание временной зоны с поправкой в 3 часа

tz = datetime.timezone(datetime.timedelta(hours=3))

now\_in\_timezone = datetime.datetime.now(tz)

print("Текущее время с временной зоной:", now\_in\_timezone)

datetime.now(): Возвращает текущую дату и время.

datetime.today(): Возвращает текущую дату.

datetime.strptime(date\_string, format): Преобразует строку в объект datetime по указанному формату.

datetime.strftime(format): Преобразует объект datetime в строку по указанному формату.

date.replace(year, month, day): Создает новый объект date, изменив его значения (год, месяц, день).

Модуль dataclasses предоставляет декоратор dataclass, который позволяет создавать data-классы - подобные позволяют значительно сократить шаблонный код классов. Как правило, такие классы предназначены для хранения некоторого состояния, некоторых данных и когда не требуется какое-то поведение в виде функций.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

tom = Person("Tom", 38)

print(f"Name: {tom.name} Age: {tom.age}") # Name: Tom Age: 38

Здесь определен класс Person, у которого в функции конструктора определены два атрибута: name и age. Далее создаем один объект этого класса и выводим значения его атрибутов на консоль.

**- Классы для работы с файлами и каталогами**

Модуль os предоставляет функции для работы с операционной системой, включая манипуляцию с файлами и каталогами.

**os.path**: содержит функции для работы с путями файлов и каталогов.

* mkdir(): создает новую папку
* rmdir(): удаляет папку
* rename(): переименовывает файл
* remove(): удаляет файл

import os

# Получить текущую рабочую директорию

current\_directory = os.getcwd()

print("Текущая директория:", current\_directory)

# Создать новый каталог

os.mkdir("new\_folder")

# Проверить существование файла

file\_exists = os.path.exists("file.txt")

print("Файл существует:", file\_exists)

# Список файлов в каталоге

files = os.listdir(".")

print("Файлы в каталоге:", files)

# Удалить файл

os.remove("file.txt")

# Удалить каталог

os.rmdir("new\_folder")

os.rename(): Переименование файла или каталога.

python

Копировать код

import os

os.rename("old\_name.txt", "new\_name.txt")

os.remove(): Удаление файла.

python

Копировать код

os.remove("file.txt")

os.mkdir(): Создание каталога.

python

Копировать код

os.mkdir("new\_folder")

os.rmdir(): Удаление пустого каталога.

python

Копировать код

os.rmdir("new\_folder")

**- Конструкторы и деструкторы**

1. Конструкторы

Конструктор — это специальный метод в классе, который вызывается при создании нового объекта этого класса. В Python конструктор называется \_\_init\_\_().

\_\_init\_\_(self, ...): вызывается при создании экземпляра класса, инициализирует атрибуты объекта.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

2. Деструкторы

Деструктор — это метод, который вызывается при удалении объекта. В Python деструктор называется \_\_del\_\_() и используется для очистки ресурсов (например, закрытие файлов, соединений с базой данных и т.д.).

\_\_del\_\_(self): вызывается при удалении объекта (при сборке мусора), но не гарантируется немедленное выполнение.

Пример:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_del\_\_(self):

print(f"Object {self.name} is being deleted.")

# Создание и удаление объекта

person1 = Person("Alice", 30)

del person1 # Здесь будет вызван \_\_del\_\_()

В Python деструкторы часто не используются напрямую, так как сборщик мусора автоматически управляет удалением объектов, когда на них больше нет ссылок.

Применение \_\_del\_\_() может быть не всегда предсказуемым, поэтому лучше использовать контекстные менеджеры (например, с with), когда нужно явно управлять ресурсами.

- **Множественное наследование. Порядок разрешения методов (MRO)**

Множественное наследование позволяет классам наследовать поведение от нескольких родительских классов.

class Animal:

def speak(self):

print("Animal speaks")

class Mammal:

def breathe(self):

print("Mammal breathes")

class Dog(Animal, Mammal):

def bark(self):

print("Dog barks")

dog = Dog()

dog.speak() # Animal speaks

dog.breathe() # Mammal breathes

dog.bark() # Dog barks

MRO определяет порядок, в котором Python будет искать методы и атрибуты в иерархии наследования. Это особенно важно в случае множественного наследования, чтобы правильно разрешить, какой метод из нескольких классов должен быть вызван.

В Python MRO основан на алгоритме C3-линеаризации, который сохраняет логику наследования и предотвращает проблемы.

class A:

def speak(self):

print("Class A")

class B(A):

def speak(self):

print("Class B")

class C(A):

def speak(self):

print("Class C")

class D(B, C):

pass

d = D()

d.speak() # Output: Class B

Здесь класс D наследует от классов B и C, а метод speak будет вызван из класса B, потому что в MRO для класса D B идет раньше, чем C.

Чтобы увидеть порядок поиска методов, можно использовать атрибут mro():

print(D.mro())

- Обработка событий клавиатуры и мыши

- Объектная модель программы. Определение объекта и класса, атрибута и свойства. Отношения, основные типы отношений.

Алгоритм C3-линеаризации обеспечивает следующий порядок:

Левый класс в списке родителей имеет более высокий приоритет.

Порядок наследования сохраняется, но родительские классы выбираются с учетом их положения в иерархии, чтобы избежать конфликтов и обеспечить правильную инициализацию.

- Оконные классы

Оконные классы в Python используются для создания оконных приложений с графическим пользовательским интерфейсом (GUI). Для работы с окнами и интерфейсами Python предоставляет несколько библиотек, таких как Tkinter, PyQt, wxPython и другие.

Основные классы Tkinter:

Tk() — основной класс для создания окна.

Label — для отображения текста или изображений.

Button — для создания кнопок.

Entry — для создания полей ввода.

Frame — контейнер для группировки других виджетов.

Text — для многострочных текстовых полей.

Canvas — для рисования графики.

Toplevel — для создания дополнительных окон.

import tkinter as tk

# Создание основного окна

root = tk.Tk()

root.title("Пример окна") # Название окна

root.geometry("400x300") # Размер окна

# Создание метки

label = tk.Label(root, text="Привет, Tkinter!")

label.pack() # Размещение метки в окне

# Создание кнопки

button = tk.Button(root, text="Нажми меня", command=lambda: label.config(text="Кнопка нажата!"))

button.pack() # Размещение кнопки в окне

# Запуск главного цикла обработки событий

root.mainloop()

**- Основные подходы к разработке программного обеспечения. Понятие объектно-ориентированного программирования (ООП).**

* Процедурное программирование:
* Основано на последовательном выполнении инструкций (функций или процедур).
* Фокусируется на алгоритмах и логике обработки данных.
* Пример: C, Pascal.

Объектно-ориентированное программирование (ООП):

* Основной концепт — объект, который объединяет данные и методы для их обработки.
* ООП способствует организации кода, снижению его сложности, повышению повторного использования и модульности.
* Ключевые принципы:
* Инкапсуляция: скрытие деталей реализации внутри объекта и предоставление только необходимых интерфейсов.
* Наследование: создание новых классов на основе существующих, что позволяет повторно использовать код.
* Полиморфизм: способность объектов разных типов обрабатывать данные одинаково, предоставляя общий интерфейс.
* Абстракция: выделение важных характеристик объекта и игнорирование несущественных деталей.
* Пример: Python, Java, C++.

Функциональное программирование:

* Рассматривает вычисления как оценку математических функций и избегает изменения состояния и данных.
* Пример: Haskell, Lisp, Scala.

Событийно-ориентированное программирование:

* Программирование, основанное на реагировании на события (например, пользовательские действия, таймеры).
* Пример: JavaScript, Node.js.

Понятие объектно-ориентированного программирования (ООП)

ООП — это парадигма программирования, основанная на использовании объектов, которые являются экземплярами классов. Классы — это шаблоны для создания объектов, которые могут содержать данные (атрибуты) и функции (методы). ООП делает код более структурированным, гибким и масштабируемым.

Ключевые понятия ООП:

Объект: Экземпляр класса, который содержит состояние и поведение (данные и методы).

Класс: Шаблон, описывающий структуру и поведение объектов.

Наследование: Механизм, позволяющий создавать новые классы на основе существующих.

Инкапсуляция: Процесс скрытия внутренней реализации объектов, предоставляя только необходимый интерфейс.

Полиморфизм: Способность объектов с разным типом вести себя одинаково при использовании общего интерфейса.

Абстракция: Процесс упрощения модели путем выделения важных характеристик объектов.

**- Перегрузка операций. Операция сложения**

Перегрузка операций в Python позволяет изменить поведение стандартных операторов для объектов пользовательских классов через магические методы (методы с двойными подчеркиваниями).

Основные магические методы для перегрузки операций:

Сложение (+):

Метод: \_\_add\_\_(self, other)

def \_\_add\_\_(self, other):

return Vector(self.x + other.x, self.y + other.y)

Вычитание (-):

Метод: \_\_sub\_\_(self, other)

def \_\_sub\_\_(self, other):

return Vector(self.x - other.x, self.y - other.y)

Умножение (\*):

Метод: \_\_mul\_\_(self, other)def \_\_mul\_\_(self, other):

return Vector(self.x \* other.x, self.y \* other.y)

Деление (/):

Метод: \_\_truediv\_\_(self, other)

def \_\_truediv\_\_(self, other):

return Vector(self.x / other.x, self.y / other.y)

Сравнение (==):

Метод: \_\_eq\_\_(self, other)

def \_\_eq\_\_(self, other):

return self.x == other.x and self.y == other.y

Отрицание (-obj):

Метод: \_\_neg\_\_(self)

def \_\_neg\_\_(self):

return Vector(-self.x, -self.y)

Приведение к строковому типу (str()):

Метод: \_\_str\_\_(self)

def \_\_str\_\_(self):

return f"Vector({self.x}, {self.y})"

class Vector:

def \_\_init\_\_(self, x, y):

self.x, self.y = x, y

def \_\_add\_\_(self, other):

return Vector(self.x + other.x, self.y + other.y)

def \_\_repr\_\_(self):

return f"Vector({self.x}, {self.y})"

v1, v2 = Vector(2, 3), Vector(4, 5)

print(v1 + v2) # Vector(6, 8)

Операция сложения в Python определяется методом \_\_add\_\_. Когда два объекта сложены с использованием оператора +, Python вызывает этот метод для первого объекта и передает второй объект как аргумент.

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

# Перегрузка операции сложения для объектов класса Vector

return Vector(self.x + other.x, self.y + other.y)

def \_\_repr\_\_(self):

return f"Vector({self.x}, {self.y})"

# Создание двух объектов класса Vector

v1 = Vector(2, 3)

v2 = Vector(4, 1)

# Сложение объектов

v3 = v1 + v2

print(v3) # Output: Vector(6, 4)

Объяснение:

Метод \_\_add\_\_ реализует поведение оператора сложения (+). Он принимает второй объект (other) и возвращает новый объект, который является результатом сложения.

В примере сложение двух объектов Vector происходит поэлементно (по x и по y).

Метод \_\_repr\_\_ используется для удобного представления объектов при выводе на экран.

**- Перегрузка операций. Операция сравнения**

Перегрузка операций сравнения позволяет изменять поведение операторов сравнения для объектов пользовательских классов. В Python для этого используются специальные магические методы.

Магические методы для перегрузки операций сравнения:

Равенство (==):

Метод: \_\_eq\_\_(self, other)

def \_\_eq\_\_(self, other):

return self.x == other.x and self.y == other.y

Неравенство (!=):

Метод: \_\_ne\_\_(self, other)

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

Меньше чем (<):

Метод: \_\_lt\_\_(self, other)

def \_\_lt\_\_(self, other):

return self.x < other.x and self.y < other.y

Меньше или равно (<=):

Метод: \_\_le\_\_(self, other)

def \_\_le\_\_(self, other):

return self.x <= other.x and self.y <= other.y

Больше чем (>):

Метод: \_\_gt\_\_(self, other)

def \_\_gt\_\_(self, other):

return self.x > other.x and self.y > other.y

Больше или равно (>=):

Метод: \_\_ge\_\_(self, other)

def \_\_ge\_\_(self, other):

return self.x >= other.x and self.y >= other.y

Оператор == сравнивает два объекта на равенство.

Оператор < сравнивает объекты по величине (например, по длине вектора).

Перегрузка операций сравнения позволяет удобно сравнивать объекты пользовательских классов, используя стандартные операторы.

**- Полиморфизм: назначение и семантика.**

Полиморфизм — это один из основных принципов объектно-ориентированного программирования (ООП), который позволяет объектам разных типов обрабатывать одинаковые сообщения или методы, предоставляя при этом разные реализации в зависимости от типа объекта.

Назначение полиморфизма:

* Гибкость и расширяемость кода: Полиморфизм позволяет разрабатывать программы, в которых можно использовать разные типы объектов одинаково, не заботясь о их конкретных классах.
* Упрощение кода: С помощью полиморфизма можно использовать один интерфейс для объектов различных классов, что уменьшает необходимость писать повторяющийся код.
* Обобщение: Позволяет создавать более общие функции и методы, которые могут работать с любыми типами данных, реализующими определенные методы.

Семантика полиморфизма:

Полиморфизм можно разделить на два типа:

1. Полиморфизм подтипов (или наследуемый полиморфизм):

Это способность объекта использовать методы своего класса или родительского класса. Например, если классы Dog и Cat наследуют от родительского класса Animal, то можно вызвать метод speak() для объектов обоих классов, и они будут выполнять свой собственный код.

1. Полиморфизм перегрузки:

Перегрузка методов или операторов позволяет использовать одно имя метода для разных операций в зависимости от числа или типа аргументов.

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

def make\_sound(animal):

print(animal.speak())

# Использование полиморфизма

dog = Dog()

cat = Cat()

make\_sound(dog) # Woof!

make\_sound(cat) # Meow!

Преимущества полиморфизма:

Позволяет работать с разными объектами одинаковым способом.

Упрощает расширение системы — добавление нового типа объектов не требует изменений в существующем коде.

Повышает читаемость и поддерживаемость кода.

**- Принципы объектно-ориентированного программирования.**

Принципы объектно-ориентированного программирования (ООП) включают основные концепции, которые помогают организовать код таким образом, чтобы он был гибким, расширяемым и поддерживаемым. Вот основные принципы ООП:

1. Инкапсуляция:

Инкапсуляция означает скрытие внутренней реализации объекта и предоставление только необходимого интерфейса для взаимодействия с ним.

Цель: Защита данных объекта от несанкционированного доступа и упрощение интерфейса. Это помогает избежать нежелательных изменений состояния объекта и делает код более безопасным и понятным.

Пример: Использование модификаторов доступа (например, private, protected, public), а также геттеров и сеттеров для взаимодействия с приватными аттрибутами объекта.

class BankAccount:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Приватный аттрибут

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance

2. Наследование:

Определение: Наследование позволяет создавать новые классы на основе существующих, заимствуя их атрибуты и методы.

Цель: Переиспользование кода и создание иерархий классов, что упрощает разработку и поддержку программы.

Пример: Класс Dog может наследовать от класса Animal, добавляя или изменяя функциональность.

class Animal:

def speak(self):

return "Some sound"

class Dog(Animal):

def speak(self):

return "Woof!"

3. Полиморфизм:

Определение: Полиморфизм позволяет объектам разных типов реагировать на одинаковые вызовы методов, предоставляя разные реализации.

Цель: Обеспечить гибкость и универсальность, позволяя использовать одинаковые интерфейсы для разных типов объектов.

Пример: Вызов одного метода для разных классов, например, метод speak() может иметь различные реализации для объектов класса Dog и Cat.

class Dog:

def speak(self):

return "Woof!"

class Cat:

def speak(self):

return "Meow!"

def make\_sound(animal):

print(animal.speak())

dog = Dog()

cat = Cat()

make\_sound(dog) # Woof!

make\_sound(cat) # Meow!

4. Абстракция:

Абстракция скрывает сложность системы, показывая только важные характеристики объекта и скрывая детали реализации.

Цель: Упростить взаимодействие с объектами, скрывая ненужные детали и предоставляя только необходимую информацию.

Пример: В Python можно использовать абстрактные классы и методы с помощью модуля abc.

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Woof!"

**- Рефлексия и интроспекция**

Рефлексия и интроспекция — это два подхода для динамического анализа объектов и классов в Python, которые позволяют получать информацию о программных сущностях во время выполнения программы.

1. Интроспекция:

Определение: Интроспекция — это процесс изучения объектов, классов или функций во время выполнения программы. Это позволяет получать информацию о структуре объекта, его методах и аттрибутах.

Пример: С помощью встроенных функций Python можно исследовать типы объектов, их атрибуты и методы.

Основные функции для интроспекции:

type(): Возвращает тип объекта.

dir(): Возвращает список атрибутов и методов объекта.

getattr(): Получение атрибута объекта по имени.

hasattr(): Проверка существования атрибута у объекта.

class Dog:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

return "Woof!"

dog = Dog("Buddy")

# Интроспекция

print(type(dog)) # <class '\_\_main\_\_.Dog'>

print(dir(dog)) # ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', ...]

print(getattr(dog, 'name')) # Buddy

print(hasattr(dog, 'speak')) # True

2. Рефлексия:

Определение: Рефлексия — это более широкое понятие, которое позволяет не только изучать, но и изменять структуру объектов в процессе выполнения программы. Рефлексия обычно включает интроспекцию, но также дает возможность динамически изменять поведение объектов.

Пример: В Python это можно делать с помощью динамических функций и методов.

Основные функции для рефлексии:

setattr(): Установка значения атрибута объекта.

delattr(): Удаление атрибута объекта.

callable(): Проверка, можно ли вызвать объект как функцию.

class Person:

def \_\_init\_\_(self, name):

self.name = name

def greet(self):

return f"Hello, my name is {self.name}"

person = Person("Alice")

# Рефлексия

setattr(person, 'age', 30) # Динамически добавляем атрибут

print(person.age) # 30

delattr(person, 'age') # Удаляем атрибут

print(hasattr(person, 'age')) # False

# Проверка, можно ли вызвать метод

print(callable(person.greet)) # True

Основные различия:

Интроспекция — это процесс получения информации о объекте (например, его атрибуты, методы или тип).

Рефлексия — это более широкий подход, который включает интроспекцию и позволяет также изменять объект во время выполнения программы.

**- Сериализация и десериализация. Pickle**

Сериализация и десериализация — это процессы преобразования данных или объектов в формат, который можно сохранить на диск или передать по сети, а затем восстановить обратно в исходный объект.

1. Сериализация:

Процесс преобразования объекта (например, списка, словаря или экземпляра класса) в формат, который можно сохранить (например, в файл, строку) или передать.

В Python для этого часто используется модуль pickle.

2. Десериализация:

Обратный процесс — восстановление объекта из сериализованного формата.

Модуль pickle:

Модуль pickle позволяет сериализовать (сохранять) и десериализовать (восстанавливать) объекты Python в / из байтовых потоков.

pickle — это модуль в Python, предназначенный для сериализации (преобразования объекта в формат, который можно сохранить или передать) и десериализации (восстановления объекта из сохранённого состояния) объектов Python.

Основные функции:

pickle.dump(obj, file) — сериализация объекта obj и запись в файл file.

pickle.load(file) — десериализация объекта из файла file.

Пример:

python

Копировать код

import pickle

# Пример объекта для сериализации

data = {'name': 'Alice', 'age': 30, 'is\_student': False}

# Сериализация: сохранение объекта в файл

with open('data.pkl', 'wb') as f:

pickle.dump(data, f)

# Десериализация: загрузка объекта из файла

with open('data.pkl', 'rb') as f:

loaded\_data = pickle.load(f)

print(loaded\_data) # {'name': 'Alice', 'age': 30, 'is\_student': False}

Преимущества и недостатки pickle:

Преимущества:

Легко использовать для сохранения сложных объектов (например, словарей, списков, экземпляров классов).

Может работать с почти любыми объектами Python.

Недостатки:

Не совместим с другими языками: формат pickle специфичен для Python, и данные, сериализованные с помощью pickle, не могут быть легко прочитаны в других языках программирования.

Безопасность: Загрузка данных через pickle может быть небезопасной, если источник данных ненадежен, так как это может привести к выполнению вредоносного кода.

**- Сериализация и десериализация. Файлы json**

JSON (JavaScript Object Notation) — это текстовый формат для обмена данными, который легко читается и используется во многих языках программирования, включая Python. В отличие от pickle, который работает только в Python, JSON поддерживается большинством языков и используется для обмена данными между различными системами.

Основные операции с JSON в Python:

Сериализация (преобразование Python-объекта в JSON).

Десериализация (преобразование JSON-строки обратно в Python-объект).

Модуль json

Python предоставляет встроенный модуль json для работы с JSON-данными.

import json

# Пример объекта для сериализации

data = {'name': 'Alice', 'age': 30, 'is\_student': False}

# Сериализация: объект Python -> JSON (в строку)

json\_data = json.dumps(data)

print(json\_data) # Выведет: {"name": "Alice", "age": 30, "is\_student": false}

# Запись JSON в файл

with open('data.json', 'w') as file:

json.dump(data, file)

# Десериализация: JSON (строка) -> объект Python

loaded\_data = json.loads(json\_data)

print(loaded\_data) # Выведет: {'name': 'Alice', 'age': 30, 'is\_student': False}

# Загрузка JSON из файла

with open('data.json', 'r') as file:

file\_data = json.load(file)

print(file\_data) # Выведет: {'name': 'Alice', 'age': 30, 'is\_student': False}

Функции модуля json:

json.dumps(obj): Преобразует Python-объект в строку JSON.

json.dump(obj, file): Записывает Python-объект в файл в формате JSON.

json.loads(json\_string): Преобразует строку JSON обратно в Python-объект.

json.load(file): Загружает JSON-данные из файла и преобразует их в Python-объект.

Преимущества JSON:

* Читаемость: Формат JSON легко читается и редактируется человеком.
* Межъязыковая совместимость: JSON поддерживается большинством языков программирования, что делает его отличным выбором для обмена данными между различными системами.

Ограничения:

* JSON не поддерживает сериализацию некоторых сложных Python-объектов (например, функции или объекты с нестандартными типами).
* В JSON все данные представляются как строки, числа, булевы значения, массивы или объекты.

Пример использования JSON для обмена данными между системами:

С помощью JSON можно передавать данные между Python-программой и веб-сервисом, где он часто используется как формат обмена данными (например, в REST API).

**- Статический (Ad hoc) полиморфизм**

Статический полиморфизм — это форма полиморфизма, при которой выбор метода или операции происходит на этапе компиляции, а не на этапе выполнения программы. Это означает, что соответствующие методы или функции выбираются заранее в зависимости от типа аргументов или параметров.

Основные примеры статического полиморфизма:

Перегрузка функций: Когда несколько функций или методов имеют одинаковое имя, но отличаются по числу или типу параметров. Компилятор выбирает нужную функцию на основе сигнатуры (списка параметров) при вызове. Пример на Python:

class Printer:

def print(self, msg: str):

print(msg)

def print(self, msg: str, times: int):

for \_ in range(times):

print(msg)

Перегрузка операторов: Позволяет изменить поведение операторов для работы с объектами определенного класса. Например, перегрузка оператора сложения для сложения двух объектов. Пример на Python:

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Point(self.x + other.x, self.y + other.y)

Статический полиморфизм важен тем, что он позволяет создавать более гибкие и удобные интерфейсы, где разработчик может использовать одно имя для разных операций, зависящих от контекста вызова.

Однако стоит отметить, что в отличие от динамического полиморфизма (который работает через механизмы позднего связывания), статический полиморфизм ограничен компилятором и не поддерживает динамическое принятие решений на основе типов объектов во время выполнения.

**- Типовые компоненты для обслуживания баз данных.**

Типовые компоненты для обслуживания баз данных включают в себя различные элементы, которые помогают эффективно управлять, обслуживать и оптимизировать работу с базами данных. Вот несколько ключевых компонентов:

Система управления базами данных: СУБД — это основная программа, которая управляет доступом к базе данных, хранением данных и выполнением запросов. Она предоставляет интерфейс для взаимодействия с базой данных, обеспечивая поддержку транзакций, безопасности, интеграции и резервного копирования.

Примеры: MySQL, PostgreSQL, Oracle DB, Microsoft SQL Server.

Менеджер подключения: Этот компонент управляет соединениями между приложениями и базой данных, обеспечивая эффективное использование ресурсов (например, пул соединений для повторного использования соединений с базой данных).

Индексирование: Индексы ускоряют поиск и извлечение данных, что особенно важно для больших баз данных. Индексы создаются на основе одного или нескольких столбцов таблиц и позволяют значительно снизить время выполнения запросов.

Резервное копирование и восстановление: Система резервного копирования позволяет регулярно создавать копии данных и восстанавливать их в случае потери или повреждения. Включает в себя полное, инкрементальное и дифференциальное резервное копирование.

Оптимизатор запросов: Оптимизатор анализирует запросы SQL и выбирает наилучший план выполнения, чтобы минимизировать время ответа и использование ресурсов. Он также может решать, какие индексы использовать, какие операции выполнять параллельно и в каком порядке.

Менеджер транзакций: Обеспечивает поддержку транзакций, гарантируя, что операции с базой данных выполняются атомарно, консистентно, изолированно и с возможностью восстановления (ACID-принципы). Это помогает поддерживать целостность данных.

Мониторинг производительности: Включает инструменты для мониторинга и анализа работы базы данных, такие как нагрузка на сервер, время отклика запросов, использование памяти и процессора. Эти данные помогают оптимизировать производительность.

Безопасность и контроль доступа: Обеспечивает механизм аутентификации и авторизации пользователей, чтобы только авторизованные лица имели доступ к данным. Это может включать в себя механизмы для шифрования данных и журналирования действий пользователей.

Обслуживание и оптимизация: Это включает в себя регулярные задачи, такие как реорганизация таблиц, обновление статистики, удаление неиспользуемых данных, сжатие данных и очистка журналов транзакций для предотвращения излишнего роста базы данных.

API и интерфейсы для взаимодействия с приложениями: Программные интерфейсы (например, ODBC, JDBC) и библиотеки, предоставляющие возможность для взаимодействия приложения с базой данных, обрабатывают запросы, передачи данных и ошибки.

Эти компоненты работают совместно, обеспечивая бесперебойную работу базы данных, высокую производительность и безопасность данных.

**- Универсальный язык моделирования UML. Диаграмма классов**

UML (Unified Modeling Language) — это универсальный язык для моделирования, который используется для проектирования и описания программных систем. Одной из самых популярных диаграмм в UML является диаграмма классов, которая представляет структуру системы через классы, их атрибуты, методы и взаимосвязи между ними.

Диаграмма классов UML

Диаграмма классов в UML используется для отображения:

Классов, их атрибутов и методов (операций).

Отношений между классами, таких как ассоциации, наследование, агрегация и композиция.

Основные элементы диаграммы классов:

Классы: Классы отображаются прямоугольниками, которые обычно разделены на три части:

Имя класса — в верхней части.

Атрибуты — во второй части.

Методы (операции) — в третьей части.

Пример:

sql

Копировать код

+-----------------+

| Person |

+-----------------+

| - name: String |

| - age: Integer |

+-----------------+

| + setName(): void|

| + getName(): String|

+-----------------+

**- Универсальный язык моделирования UML. Диаграмма прецедентов**

Диаграмма прецедентов (Use Case Diagram) в UML — это один из видов диаграмм, который используется для моделирования функциональных требований системы с точки зрения пользователя. Эта диаграмма описывает взаимодействие между пользователями (или другими системами) и системой, а также показывается, какие операции (прецеденты) система должна выполнять в ответ на действия пользователей.

Основные элементы диаграммы прецедентов UML:

Акторы (Actors): Актор — это внешний объект (пользователь или другая система), который взаимодействует с системой. Актор может быть:

Пользователем (например, клиент, администратор).

Системой (например, внешняя система или сервис, с которым взаимодействует ваша система).

Акторы отображаются как человечки (или иконки) или прямоугольники с именами, и они могут быть подключены к прецедентам, с которыми они взаимодействуют.

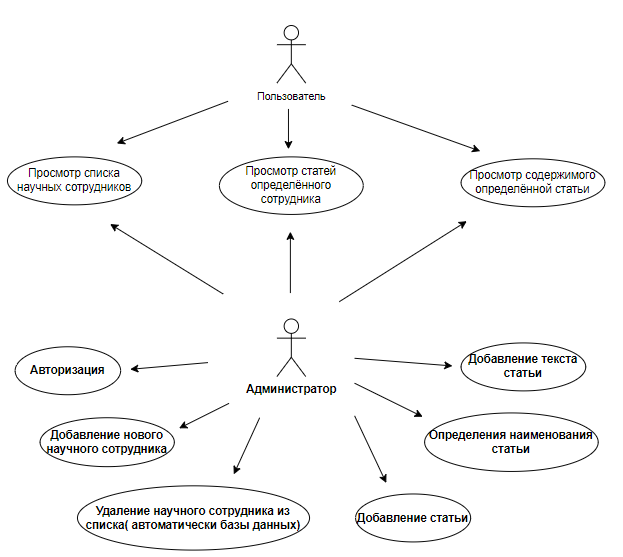
Прецеденты (Use Cases): Прецедент — это определенная задача или функция, которую система должна выполнять в ответ на действие актора. Прецеденты описывают, что система будет делать, но не как она будет это делать. Прецеденты отображаются как овалы с описанием действия (например, «Регистрация пользователя», «Добавление товара в корзину»).

Связи (Relationships): Связи показывают, как акторы взаимодействуют с прецедентами, а также как прецеденты могут быть связаны между собой. В UML могут быть использованы следующие типы связей:

Ассоциация: Линия между актором и прецедентом, показывающая взаимодействие.

Расширение (Extend): Указывает, что один прецедент может быть расширен другим, при этом дополнительная функциональность может быть вызвана в зависимости от условий.

Включение (Include): Указывает, что один прецедент всегда будет включать выполнение другого, т.е. оба прецедента выполняются всегда, когда срабатывает основной прецедент.



**Схемы построения задания 2**

**- База данных «» содержит таблицы: ... Построить контекстную диаграмму для ...**

**- В базе данных содержатся сведения о .... Таблицы: .... Спроектировать соответствующие классы. Построить диаграмму классов.**

**- Разработать классы для моделирования ... : ... Текстовое описание задачи. Построить контекстную диаграмму для ...**

**- Спроектировать класс ... с атрибутами:... Спроектировать ... Построить контекстную диаграмму для ...**

class Book:  
 def \_\_init\_\_(self, title: str, auyhor: str, year: int):  
 self.title = title  
 self.auyhor = auyhor  
 self.year = year

- Спроектировать класс ... с операциями ... Разработать pytest / unittest для тестирования класса.

class Calculate:  
 def pluss(self, a, b):  
 return a + b  
   
class Matematika:  
 def minus(self,a,c,f):  
 return (a + c)/f

import pytest  
from Calcс import Calculate  
  
@pytest.fixture  
  
def calc():  
 return Calculate()  
  
def test\_add(calc):  
 assert calc.add(2, 3) == 6  
  
def test\_minus(calc):  
 assert calc.minus(10, 5,3)==5

**- Спроектировать класс ... на основе односвязного списка. Переопределить методы сложения, умножения, сравнения. Реализовать метод .... Разработать pytest / unittest для класса**

**!!!- Спроектировать класс «» с атрибутами.... Спроектировать класс «» на основе кольцевого однонаправленного списка Построить диаграмму классов.**  
class Node:

def \_\_init\_\_(self, data):

self.data = data # данные узла

self.next = None # ссылка на следующий узел

class CircularLinkedList:

def \_\_init\_\_(self):

self.head = None # указатель на начало списка

def append(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

new\_node.next = self.head # кольцо: следующий узел — это сам head

else:

current = self.head

while current.next != self.head:

current = current.next

current.next = new\_node

new\_node.next = self.head # кольцо: следующий узел — это сам head

def display(self):

if not self.head:

print("Список пуст")

return

current = self.head

while True:

print(current.data, end=" -> ")

current = current.next

if current == self.head:

break

print("(head)")

**!!!- Спроектировать класс ...: на основе очереди с приоритетами. Построить контекстную диаграмму для ...**  
class PriorityQueue:

def \_\_init\_\_(self):

self.queue = [] # Очередь в виде списка

def push(self, item, priority):

"""Добавить элемент с приоритетом."""

self.queue.append((priority, item))

# Сортируем очередь по приоритету (чем ниже число, тем выше приоритет)

self.queue.sort(key=lambda x: x[0])

def pop(self):

"""Удалить элемент с наивысшим приоритетом."""

if not self.is\_empty():

return self.queue.pop(0)[1] # Удаляем первый элемент (с наивысшим приоритетом)

return None

def peek(self):

"""Посмотреть элемент с наивысшим приоритетом без удаления."""

if not self.is\_empty():

return self.queue[0][1] # Первый элемент в очереди

return None

def is\_empty(self):

"""Проверить, пуста ли очередь."""

return len(self.queue) == 0

# Тестирование

pq = PriorityQueue()

pq.push("Task 1", priority=3)

pq.push("Task 2", priority=1)

pq.push("Task 3", priority=2)

print("Peek:", pq.peek()) # Ожидается: Task 2

print("Pop:", pq.pop()) # Ожидается: Task 2

print("Pop:", pq.pop()) # Ожидается: Task 3

print("Pop:", pq.pop()) # Ожидается: Task 1

**- Спроектировать классы при помощи наследования и агрегации для моделирования ... . Построить диаграмму классов.**

class Book:

def \_\_init\_\_(self, name, auto, year):

self.name = name

self.auto = auto

self.year = year

def info(self):

return f"Книга: {self.name}, автор: {self.auto}, год: {self.year}"

def \_\_str\_\_(self):

return self.info()

class EBook(Book):

def \_\_init\_\_(self, name, auto, year, file):

super().\_\_init\_\_(name, auto, year)

self.file = file

def \_\_str\_\_(self):

return f"Эл. книга: {self.name}, автор: {self.auto}, год: {self.year}, файл: {self.file}"

class Library:

def \_\_init\_\_(self, name):

self.name = name

self.books = [] # Создаём список для хранения книг

def addB(self, book):

if isinstance(book, Book): # Проверяем, что добавляется объект класса Book или его потомка

self.books.append(book)

return True

return False

def show(self):

print(f"{self.name}")

for book in self.books:

print(book)

# Создаём объекты

book = Book("ILOVEMAKS", "SMOFIA", "2022")

ebook = EBook("ILOVEMAKS", "SMOFIA", "2022", "cdv2022MS")

lib = Library("Моя голова...")

# Проверяем методы

lib.addB(book)

lib.addB(ebook)

lib.show()

**- Спроектировать классы, описывающие ...: . Построить диаграмму классов.**

**class Car:**

def \_\_init\_\_(self, mark, color, price):

self.mark = mark

self.color = color

self.price = price

def \_\_str\_\_(self):

return f"Машина {self.mark} цвета {self.color} за {self.price}."

class Moped(Car):

def \_\_init\_\_(self, mark, color, price):

super().\_\_init\_\_(mark, color, price)

def \_\_str\_\_(self):

return f"Мопед {self.mark} цвета {self.color} за {self.price}."

car = Car("BMW X6", "чёрный", "600000")

mop = Moped("Mazda", "синий", "30000")

print(car)

print(mop)

**- Спроектировать классы: .... Унаследовать ... от ... В классе переопределить операции сравнения, добавления элемента. Реализовать методы ... Построить диаграмму классов.**class Prmoyg:  
 def \_\_init\_\_(self, rina, dlina):  
 self.rina = rina  
 self.dlina = dlina  
  
 def area(self):  
 return self.rina \* self.dlina  
  
 def perimetr(self):  
 return 2 \* (self.rina + self.dlina)  
  
 def \_\_str\_\_(self):  
 return f'Prmoyg({self.rina}, {self.dlina})'  
  
 def \_\_eq\_\_(self, other):  
 if isinstance(other, Prmoyg):  
 return self.area() == other.area()  
 return False  
  
 def \_\_lt\_\_(self, other):  
 if isinstance(other, Prmoyg):  
 return self.area() < other.area()  
 return False  
  
 def \_\_add\_\_(self, other):  
 if isinstance(other, Prmoyg):  
 new\_rina = self.rina + other.rina  
 new\_dlina = self.dlina + other.dlina  
 return Prmoyg(new\_rina, new\_dlina)  
 return False  
  
  
class Qwadrt(Prmoyg):  
 def \_\_init\_\_(self, storona):  
 super().\_\_init\_\_(storona, storona)  
  
 def \_\_str\_\_(self):  
 return f"Qwadrt({self.dlina})"  
  
 def \_\_eq\_\_(self, other):  
 if isinstance(other, Qwadrt):  
 return self.area() == other.area()  
 return False  
  
 def \_\_add\_\_(self, other):  
 if isinstance(other, Qwadrt):  
 new\_qw = (self.area() + other.area()) \*\* 0.5  
 return Qwadrt(new\_qw)  
 return False  
  
  
*# Создание объектов*pr1 = Prmoyg(4, 6)  
pr2 = Prmoyg(3, 9)  
  
qw1 = Qwadrt(5)  
qw2 = Qwadrt(8)  
  
*# Вывод*print(pr1)  
print(qw1)  
  
print(pr1 == pr2) *# Сравнение площадей прямоугольников*print(qw1 == qw2) *# Сравнение площадей квадратов*p3 = pr1 + pr2  
print(p3)  
  
qw3 = qw1 + qw2  
print(qw3)

**Схемы построения задания 3**

- Написать визуальную программу для создания, редактирования и вывода ... в Excel-файл.

- Написать программу для демонстрации .... Предусмотреть обработку ошибок в случае ... Данные и лог движения хранить в текстовых файлах.

- Написать программу для демонстрации ... с визуальным интерфейсом. Программа должна формировать ... в Excel. Ввод данных- с клавиатуры.

- Написать программу, демонстрирующую .... Данные создавать случайным образом и выводить на экран

- Написать программу, демонстрирующую работу с классами. Переопределить операции добавления, удаления, сравнения. Данные и лог хранить в json-файлах. Использовать меню.

- Разработать визуальное приложение для демонстрации работы с системой классов. Программа должна формировать на экране отчет «...». Способ хранения данных определяется студентом самостоятельно

- Разработать программу для демонстрации работы с классами. Программа должна сформировать и вывести в Excel-файл извещение на оплату по указанному счету.

- Разработать программу для демонстрации системы классов. Программа должна выводить в Excel-файл отчет...

- Разработать программу для демонстрации системы классов. Программа должна выдавать в текстовый файл:...

- Разработать программу для демонстрации системы классов. Программа должна сформировать и вывести в текстовый файл отчет ... Данные хранить в базе данных