МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Объектно ориентированное программирование»
для студентов направлений 09.03.03 «Прикладная информатика».

Ставрополь
2023

ВВЕДЕНИЕ

Целью дисциплины является обучение студентов основам объектно-ориентированного проектирования и программирования в современных средах разработки ПО.

Задачами дисциплины являются:

- Основой задачей изучения курса является получение знаний и практических навыков в области проектирования и разработки объектно-ориентированных программ. В результате изучения курса студент должен иметь представление о предпосылках возникновения ООП и его месте в эволюции парадигм программирования, знать принципы объектно-ориентрованного проектирования и программирования, а также уметь разрабатывать объектно-ориентированные программы на языках

Лабораторная работа №1.
Основы объектно ориентированного программирования на ЯП Python.

Цель работы: изучить базовые понятия (классы, подклассы и методы) Реализовать фундаментальные принципы объектно-ориентированного программирования.

Формируемые компетенции: ПК-7, ПК-8

1. Теоретическая часть

1.1. Введение

Объектно-ориентированное программирование (ООП) – это парадигма программирования, в которой для представления данных и для проведения операций над этими данными используются объекты.

Объекты, в свою очередь, являются экземплярами классов – с этой точки зрения классы можно назвать шаблонами для создания объектов определенного типа. Классы определяют:

структуру данных, которые характеризуют объект;

свойства (атрибуты) и статус (состояние) объекта;

операции, которые можно совершать с данными объекта (методы).

В этом примере класс Car (автомобиль) имеет атрибуты make, model, year (марка, модель, год выпуска):фундаментальных принципа объектно-ориентированного программирования: инкапсуляция и наследование.

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

Атрибуты – это свойства, характеристики объекта. Они определяют качества и состояние объекта. Атрибуты объекта перечисляют внутри__init__ метода класса – он вызывается каждый раз при создании экземпляра класса. Параметр self создает ссылку на экземпляр класса и позволяет получить доступ к атрибутам и методам объекта. Для создания экземпляра Car достаточно вызвать класс, передавая в скобках значения, соответствующие его атрибутам:

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

Теперь, когда атрибутам объекта присвоены значения, можно к ним обращаться – для этого используют выражение название_объекта.атрибут:

```
print(f'Марка машины {my_car.make},'
      f'\nмодель {my_car.model},'
      f'\nгод выпуска - {my_car.year}'
      )
```

Результат:

```
Марка машины Toyota,
модель Corolla,
```

```
год выпуска - 2023
```

Car – пример простейшего класса: у него нет ни подклассов, ни методов, кроме обязательного __init__. Метод – это функция, которая определяет поведение объекта. Проиллюстрируем создание метода на примере класса WashingMachine – здесь метод remaining_warranty_time() определяет срок истечения гарантии на стиральную машину:

```
import datetime
class WashingMachine:
    def __init__(self, brand, model, purchase_date, warranty_length):
        self.brand = brand
        self.model = model
        self.purchase_date = purchase_date
        self.warranty_length = warranty_length

    def remaining_warranty_time(self):
        today = datetime.date.today()
        warranty_end_date = self.purchase_date +
datetime.timedelta(days=self.warranty_length)
        remaining_time = warranty_end_date - today
        if remaining_time.days < 0:
            return "Срок действия гарантии истек."
        else:
            return "Срок действия гарантии истекает через {}
дней.".format(remaining_time.days)

# создаем объект стиральной машины
my_washing_machine = WashingMachine("LG", "FH4U2VCN2", datetime.date(2022, 5,
7), 1550)

# вызываем метод для проверки срока истечения гарантии
print(my_washing_machine.remaining_warranty_time())
```

Результат:

```
Срок действия гарантии истекает через 1218 дней.
```

Теперь рассмотрим чуть более сложный пример с подклассами и методами. Предположим, что нам нужно разработать CRM для автосалона. В ПО автосалона должен быть класс Vehicle (транспортное средство), который имеет набор атрибутов:

марка;

модель;

год выпуска;

стоимость.

Среди методов должна быть операция display_info(), которая отображает информацию о конкретном транспортном средстве, а помимо классов, в ПО необходимо использовать подклассы.

Подкласс – это класс, который наследует все атрибуты и методы родительского класса (также известного как базовый класс или суперкласс), но при этом может иметь дополнительные, свои собственные, атрибуты и методы. Концепцию наследования мы подробнее разберем ниже.

В ПО для автосалона необходимо создать подкласс Car (легковой автомобиль), который наследует все атрибуты и методы класса Vehicle, и при этом имеет дополнительные атрибуты, например количество дверей и стиль кузова. Аналогично, мы можем создать подкласс Truck (грузовик), который наследует все атрибуты и методы класса Vehicle, и к тому же имеет свои атрибуты – длину кузова и тяговую мощность.

В итоге, взаимодействие классов, подклассов и методов будет выглядеть так:

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

    def display_info(self):
        print(f"Марка: {self.make}"
        f"\nМодель: {self.model}"
        f"\nГод выпуска: {self.year}"
        f"\nСтоимость: {self.price} руб")

class Car(Vehicle):
    def __init__(self, make, model, year, price, num_doors, body_style):
        super().__init__(make, model, year, price)
        self.num_doors = num_doors
        self.body_style = body_style

class Truck(Vehicle):
    def __init__(self, make, model, year, price, bed_length,
towing_capacity):
        super().__init__(make, model, year, price)
        self.bed_length = bed_length
        self.towing_capacity = towing_capacity
```

Создадим экземпляры классов и вызовем метод display_info() для вывода информации о них:

```
# создаем объект "легковой автомобиль"
car = Car("Toyota", "Camry", 2022, 2900000, 4, "седан")
# создаем объект "грузовик"
truck = Truck("Ford", "F-MAX", 2023, 6000000, "6162", "13 т")
# выводим информацию о легковом автомобиле и грузовике
car.display_info()
truck.display_info()
```

Результат:

```
Марка: Toyota
Модель: Camry
Год выпуска: 2022
Стоимость: 2900000 руб
Марка: Ford
Модель: F-MAX
Год выпуска: 2023
Стоимость: 6000000 руб
```

В этом примере используется встроенная функция super(), которая позволяет вызывать методы родительского суперкласса из подкласса. Этот прием позволяет переиспользовать методы и расширять их функциональность. В данном случае вызывается метод инициализации super().__init__, который позволяет применить атрибуты суперкласса к подклассу. При необходимости, помимо унаследованных, можно определить новые свойства, которые относятся только к конкретному подклассу.

Рассмотрим еще один пример – библиотечную программу для хранения информации о книгах и их статусах (есть в наличии, выдана абоненту, получена от абонента и так далее). Здесь класс Book определяет различные характеристики книги – title, author, ISBN, а также задает методы check_out() и check_in(), которые выдают / принимают книги, и сообщают о статусах:

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.checked_out = False

    def check_out(self):
        if self.checked_out:
            print("Книга находится у абонента.")
        else:
            self.checked_out = True
            print("Выдаем книгу абоненту.")

    def check_in(self):
        if not self.checked_out:
            print("Книга в наличии.")
        else:
            self.checked_out = False
            print("Принимаем книгу в библиотеку.")
```

Создадим объект книги и проверим статусы:

```
# создаем объект книги
book1 = Book("Война и мир", "Л.Н. Толстой", "978-0743273565")
# выдаем книгу, проверяем статус
book1.check_out()
# проверяем статус повторно
book1.check_out()
# принимаем книгу от читателя
book1.check_in()
# проверяем статус книги повторно
book1.check_in()
```

Результат:

```
Выдаем книгу абоненту.
Книга находится у абонента.
Принимаем книгу в библиотеку.
Книга в наличии.
```

Классы, объекты, атрибуты и методы – самые простые, самые базовые понятия ООП. Эти базовые концепции, в свою очередь, лежат в основе фундаментальных принципов ООП.

1.2. Фундаментальные принципы ООП

ООП основывается на четырех фундаментальных принципах: инкапсуляции, наследовании, полиморфизме и абстракции.

Инкапсуляция – механизм сокрытия деталей реализации класса от других объектов. Достигается путем использования модификаторов доступа public, private и protected, которые соответствуют публичным, приватным и защищенным атрибутам.

Наследование – процесс создания нового класса на основе существующего класса. Новый класс, называемый подклассом или производным классом, наследует свойства и методы существующего класса, называемого суперклассом или базовым классом.

Полиморфизм – способность объектов принимать различные формы. В ООП полиморфизм позволяет рассматривать объекты разных классов так, как если бы они были объектами одного класса.

Абстракция – процесс определения существенных характеристик объекта и игнорирования несущественных характеристик. Это позволяет создавать абстрактные классы, которые определяют общие свойства и поведение группы объектов, не уточняя детали каждого объекта.

1.2.1. Инкапсуляция

Сделаем атрибуты title, author и isbn класса Book приватными – теперь доступ к ним возможен только внутри класса:

```
class Book:
    def __init__(self, title, author, isbn):
        self.__title = title  # приватный
        self.__author = author  # приватный
        self.__isbn = isbn  # приватный
```

Чтобы получить доступ к этим атрибутам извне класса, мы определяем методы getter и setter, которые обеспечивают контролируемый доступ к атрибутам:

```
def get_title(self):
        return self.__title

    def set_title(self, title):
        self.__title = title

    def get_author(self):
        return self.__author

    def set_author(self, author):
        self.__author = author

    def get_isbn(self):
        return self.__isbn
```

```
    def set_isbn(self, isbn):
        self.__isbn = isbn
```

В этом примере методы get_title(), get_author() и get_isbn() являются получающими методами (геттерами), которые позволяют нам получать значения приватных атрибутов извне класса. Методы set_title(), set_author() и set_isbn() – устанавливающие методы (сеттеры), которые позволяют нам устанавливать значения частных атрибутов извне класса.

Создадим экземпляр объекта и попытаемся получить доступ к его названию с помощью обычного метода:

```
book1 = Book("Террор", "Дэн Симмонс", "558-0743553565")
# пытаемся получить доступ к приватному атрибуту
print(book1.__title)
```

Результат – ошибка:

```
AttributeError: 'Book' object has no attribute '__title'
```

Воспользуемся геттерами:

```
# получаем приватные атрибуты с помощью геттеров
print(book1.get_title())
print(book1.get_author())
print(book1.get_isbn())
```

Результат:

```
Террор
Дэн Симмонс
558-0743553565
```

Изменим название с помощью сеттера и выведем результат:

```
# изменяем название с помощью сеттера
book1.set_title("Эндимион")
print(book1.get_title())
```

Результат:

```
Эндимион
```

1.2.2. Наследование

Для иллюстрации концепции наследования мы определим класс **Publication**, который имеет свойства, общие для всех публикаций – **title**, **author** и **year**, а также общий метод display():

```
class Publication:
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

    def display(self):
        print("Название:", self.title)
        print("Автор:", self.author)
        print("Год выпуска:", self.year)
```

Теперь создадим два подкласса **Book** и **Magazine**, которые наследуют все свойства и методы от класса **Publication**, и кроме того, имеют свои атрибуты. Подкласс Book добавляет свойство **isbn** и переопределяет метод display() для включения свойства **isbn**. Подкласс Magazine добавляет свойство **issue_number** (номер выпуска) и переопределяет методdisplay()для включения свойства **issue_number**:

```
class Book(Publication):
    def __init__(self, title, author, year, isbn):
        super().__init__(title, author, year)
        self.isbn = isbn

    def display(self):
        super().display()
        print("ISBN:", self.isbn)

class Magazine(Publication):
    def __init__(self, title, author, year, issue_number):
        super().__init__(title, author, year)
        self.issue_number = issue_number

    def display(self):
        super().display()
        print("Номер выпуска:", self.issue_number)
```

Теперь, если мы создадим экземпляр класса **Book** или класса **Magazine**, мы сможем вызвать метод display()для отображения свойств объекта. Сначала будет вызван метод display() подкласса (Book или Magazine), который в свою очередь вызовет метод display()суперкласса **Publication** с помощью функции super(). Это позволяет нам повторно использовать код суперкласса и избежать дублирования кода в подклассах:

```
# создаем объект книги
book1 = Book("Выбор", "Эдит Эгер", 2019, "112-3333273566")

# создаем объект выпуска журнала
magazine1 = Magazine("Вокруг света", "коллектив авторов", 2023, 3)

# выводим информацию о книге и номере журнала
book1.display()
magazine1.display()
```

Результат:

```
Название: Выбор
Автор: Эдит Эгер
Год выпуска: 2019
ISBN: 112-3333273566
Название: Вокруг света
Автор: коллектив авторов
Год выпуска: 2023
Номер выпуска: 3
```

1.2.3. Абстракция

Одна из основных целей использования абстракции в ООП – повышение гибкости и упрощение разработки. Абстрактный подход

помогает создавать интерфейсы и классы, которые определяют только те свойства и методы, которые необходимы для выполнения определенной задачи. Это позволяет создавать более гибкие и масштабируемые приложения, которые легко поддаются изменению и расширению.

Предположим, что нам нужно написать программу, которая работает с графическими объектами разных типов. Для решения этой задачи удобно создать абстрактный класс Shape (фигура), определяющий абстрактные методы, которые могут быть использованы для работы с любой фигурой. Затем мы можем создать конкретные классы для конкретных типов фигур – окружность, квадрат, треугольник и т.д., которые расширяют базовый класс Shape. При этом мы можем использовать только те свойства и методы, которые необходимы для выполнения конкретной задачи, игнорируя детали реализации, которые не имеют значения в данном контексте.

Абстрактный подход помогает эффективно решать ряд сложных задач:

- Позволяет выделять существенные характеристики объекта, игнорируя все незначительные детали.
- Принуждает подклассы к реализации конкретных методов или к выполнению определенных требований путем определения абстрактных методов или свойств. Таким образом, абстракция позволяет определять общие интерфейсы для классов, но при этом гарантирует, что каждый подкласс будет реализовывать свою версию этих методов или свойств.
- Позволяет создавать общие модели объектов, которые могут использоваться для создания конкретных объектов.
- Упрощает работу со сложными системами, которые включают множество взаимодействующих компонентов, и позволяет создавать расширяемые, модульные приложения.

1.2.4. Абстрактные классы в Python

Для работы с абстрактными классами в Python используют модуль **abc**. Он предоставляет:

- **abc.ABC** – базовый класс для создания абстрактных классов. Абстрактный класс содержит один или несколько абстрактных методов, то есть методов без определения (пустых, без кода). Эти методы необходимо переопределить в подклассах.
- **abc.abstractmethod** – декоратор, который указывает, что метод является абстрактным. Этот декоратор применяется к методу внутри абстрактного класса. Класс, который наследует свойства и методы от абстрактного класса, должен реализовать все абстрактные методы, иначе он также будет считаться абстрактным.

Рассмотрим пример абстрактного класса **Book**:

```
from abc import ABC, abstractmethod

class Book(ABC):
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

```
    @abstractmethod
    def get_summary(self):
        pass

class Fiction(Book):
    def get_summary(self):
        print(f'"{self.title}" - роман в стиле исторический фикшн, автор -
{self.author}')

class NonFiction(Book):
    def get_summary(self):
        print(f'"{self.title}" - книга в стиле нон фикшн, автор -
{self.author}')

class Poetry(Book):
    pass
```

Класс **Book** имеет абстрактный метод **get_summary()**. Два подкласса Book (**Fiction** и **NonFiction**) реализуют метод **get_summary()**, а третий подкласс **Poetry** – нет. Когда мы создаем экземпляры Fiction и NonFiction и вызываем их методы get_summary(), получаем ожидаемый результат:

```
fiction_book = Fiction("Террор", "Дэн Симмонс")
nonfiction_book = NonFiction("Как писать книги", "Стивен Кинг")
fiction_book.get_summary()
nonfiction_book.get_summary()
```

Вывод:

```
"Террор" - роман в стиле исторический фикшн, автор - Дэн Симмонс
"Как писать книги" - книга в стиле нон фикшн, автор - Стивен Кинг
```

А вот вызов **Poetry** приведет к ошибке, поскольку в этом подклассе метод **get_summary()** не реализован:

```
poetry_book = Poetry("Стихотворения", "Борис Пастернак")
```

Вывод:

```
TypeError: Can't instantiate abstract class Poetry with abstract methods
get_summary
```

Приведенный выше пример показывает, что семейство родственных классов (**Fiction** и **NonFiction** в нашем случае) может иметь общий интерфейс (метод **get_summary()**), но реализация этого интерфейса может быть разной. Мы также убедились, что любой подкласс **Book** должен реализовать метод **get_summary()**, чтобы обеспечить согласованную, безошибочную работу приложения.

Теперь рассмотрим чуть более сложный пример, который продемонстрирует, как можно комбинировать абстракцию с другими концепциями ООП. Определим абстрактный класс **Recipe** (рецепт), который имеет абстрактный метод **cook()**. Затем создадим три подкласса **Entree**, **Dessert** и **Appetizer** (основное блюдо, десерт и закуска). **Entree** и **Dessert** имеют свои собственные методы **cook()**, в отличие

от **Appetizer** и **PartyMix**. **PartyMix** (орешки, чипсы, крекеры) является подклассом **Appetizer** и имеет свою реализацию **cook()**:

```
from abc import ABC, abstractmethod

class Recipe(ABC):
    @abstractmethod
    def cook(self):
        pass

class Entree(Recipe):
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def cook(self):
        print(f"Готовим на медленном огне смесь ингредиентов ({', 
'.join(self.ingredients)}) для основного блюда")

class Dessert(Recipe):
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def cook(self):
        print(f"Смешиваем {', '.join(self.ingredients)} для десерта")

class Appetizer(Recipe):
    pass

class PartyMix(Appetizer):
    def cook(self):
        print("Готовим снеки - выкладываем на поднос орешки, чипсы и 
крекеры")
```

В этом примере наряду с абстракцией используются концепции полиморфизма и наследования.

Наследование заключается в том, что подклассы Entree, Dessert и PartyMix наследуют абстрактный метод cook() от абстрактного базового класса Recipe. Это означает, что все они имеют ту же сигнатуру (название и параметры) метода cook(), что и абстрактный метод, определенный в классе Recipe.

Полиморфизм проявляется в том, что каждый подкласс класса Recipe реализует метод cook() по-разному. Например, Entree реализует cook() для вывода инструкций по приготовлению основного блюда на медленном огне, а Dessert реализует cook() для вывода инструкций по смешиванию ингредиентов десерта. Эта разница в реализации является примером полиморфизма, когда различные объекты могут рассматриваться как объекты, которые относятся к одному типу, но при этом ведут себя по-разному:

```
entree = Entree(["курица", "рис", "овощи"])
dessert = Dessert(["мороженое", "шоколадные чипсы", "мараскиновые вишни"])
partymix = PartyMix()
entree.cook()
dessert.cook()
partymix.cook()
```

Результат:

```
Готовим на медленном огне смесь ингредиентов (курица, рис, овощи) для
основного блюда
Смешиваем мороженое, шоколадные чипсы, мараскиновые вишни для десерта
Готовим снеки - выкладываем на поднос орешки, чипсы и крекеры
```

Вызов метода cook() для подкласса Appetizer приведет к ожидаемой ошибке:

```
appetizer = Appetizer()
appetizer.cook()
```

Результат:

```
TypeError: Can't instantiate abstract class Appetizer with abstract methods
cook
```

1.2.5. Полиморфизм

Полиморфизм позволяет обращаться с объектами разных классов так, как будто они являются объектами одного класса. Реализовать полиморфизм можно через наследование, интерфейсы и перегрузку методов. Этот подход имеет несколько весомых преимуществ:

- Позволяет использовать различные реализации методов в зависимости от типа объекта, что делает код более универсальным и удобным для использования.
- Уменьшает дублирование кода – можно написать одну функцию для работы с несколькими типами объектов.
- Позволяет использовать общие интерфейсы и абстракции для работы с объектами разных типов.
- Обеспечивает гибкость и расширяемость – можно добавлять новые типы объектов без необходимости изменять существующий код. Это дает возможность разработчикам встраивать новые функции в программу, не нарушая ее существующую функциональность.

Полиморфизм тесно связан с абстракцией:

- Абстракция позволяет скрыть детали реализации объекта и предоставить только необходимый интерфейс для работы с ним. Это помогает упростить код, сделать его более понятным и гибким.
- Полиморфизм предоставляет возможность использовать один и тот же интерфейс для работы с разными объектами, которые могут иметь различную реализацию. Этот подход значительно упрощает расширение функциональности ПО.

Таким образом, абстракция позволяет определить общий интерфейс для работы с объектами, а полиморфизм позволяет использовать этот интерфейс для работы с различными объектами, которые могут иметь различную реализацию.

Рассмотрим полиморфизм на примере класса **Confectionary** (кондитерские изделия):

```
class Confectionary:
```

```
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def describe(self):
        print(f"{self.name} по цене {self.price} руб/кг")

class Cake(Confectionary):
    def describe(self):
        print(f"{self.name} торт стоит {self.price} руб/кг")

class Candy(Confectionary):
    def describe(self):
        print(f"{self.name} конфеты стоимостью {self.price} руб/кг")

class Cookie(Confectionary):
    pass
```

В этом примере мы определяем базовый класс **Confectionary**, который имеет атрибуты **name** и **price**, а также метод **describe()**. Затем мы определяем три подкласса класса Confectionary: **Cake**, **Candy** и **Cookie**. Cake и Candy переопределяют метод **describe()** своими собственными реализациями, которые включают тип кондитерского изделия (торт и конфеты соответственно), а Cookie наследует дефолтный метод describe() от Confectionary.

Если создать экземпляры этих классов и вызвать их методы describe(), можно убедиться, что результат зависит от реализации метода в каждом конкретном подклассе:

```
cake = Cake("Пражский", 1200)
candy = Candy("Шоколадные динозавры", 560)
cookie = Cookie("Овсяное печенье с миндалем", 250)

cake.describe()
candy.describe()
cookie.describe()
```

Вывод:

```
Пражский торт стоит 1200 руб/кг
Шоколадные динозавры конфеты стоимостью 560 руб/кг
Овсяное печенье с миндалем по цене 250 руб/кг
```

Теперь разберем на примере класса **Beverage** (напиток) взаимодействие полиморфизма с другими концепциями ООП. Beverage – родительский класс, который содержит:

- атрибуты названия, объема и цены;
- методы для получения и установки этих атрибутов;
- метод для вывода описания напитка.

Soda (газировка) – дочерний класс Beverage, у него есть дополнительный атрибут **flavor** (вкус) и собственный метод **describe()**, включающий flavor. **DietSoda** – еще один дочерний класс Soda, который наследует все атрибуты и методы Soda, но переопределяет метод **describe()**, чтобы указать, что газировка является диетической:

```
class Beverage:
    def __init__(self, name, size, price):
        self._name = name
        self._size = size
        self._price = price

    def get_name(self):
        return self._name

    def get_size(self):
        return self._size

    def get_price(self):
        return self._price

    def set_price(self, price):
        self._price = price

    def describe(self):
        return f'{self._size} л газировки "{self._name}" стоит {self._price}
руб.'

class Soda(Beverage):
    def __init__(self, name, size, price, flavor):
        super().__init__(name, size, price)
        self._flavor = flavor

    def get_flavor(self):
        return self._flavor

    def describe(self):
        return f'{self._size} л газировки "{self._name}" со вкусом
"{self._flavor}" стоит {self._price} руб.'

class DietSoda(Soda):
    def __init__(self, name, size, price, flavor):
        super().__init__(name, size, price, flavor)

    def describe(self):
        return f'{self._size} л диетической газировки "{self._name}" со
вкусом "{self._flavor}" стоит {self._price} руб.'

regular_soda = Soda('Sprite', 0.33, 45, 'лимон')
print(regular_soda.describe())
diet_soda = DietSoda('Mirinda', 0.33, 50, 'мандарин')
print(diet_soda.describe())
regular_soda = Soda('Буратино', 1.5, 65, 'дюшес')
print(regular_soda.describe())
```

Этот пример демонстрирует:

- Инкапсуляцию, поскольку атрибуты защищены символами подчеркивания и могут быть доступны только через методы **getter** и **setter**.
- Наследование, поскольку Soda и DietSoda наследуют атрибуты и метод от Beverage.

- Полиморфизм, поскольку каждый класс имеет свою собственную версию метода describe(), который возвращает различные результаты в зависимости от конкретного класса.

  Вывод:

```
0.33 л газировки "Sprite" со вкусом "лимон" стоит 45 руб.
0.33 л диетической газировки "Mirinda" со вкусом "мандарин" стоит 50 руб.
1.5 л газировки "Буратино" со вкусом "дюшес" стоит 65 руб.
```

2. Практическая часть

Пример 1

Напишите класс **MusicAlbum**, у которого есть:

- Атрибуты **title**, **artist**, **release_year**, **genre**, **tracklist** (название, исполнитель, год выхода, жанр, список треков.
- Метод play_random_track() для вывода случайного названия песни.

Пример использования:

```
album4 = MusicAlbum("Deutschland", "Rammstein", 2019, "Neue Deutsche Härte",
                    ["Deutschland", "Radio", "Zeig dich", "Ausländer", "Sex",
                     "Puppe", "Was ich liebe", "Diamant", "Weit weg",
"Tattoo",
                     "Hallomann"])
print("Название:", album4.title)
print("Исполнитель:", album4.artist)
print("Год:", album4.release_year)
print("Жанр:", album4.genre)
print("Треки:", album4.tracklist)
album4.play_random_track()
```

Вывод:

```
Название: Deutschland
Исполнитель: Rammstein
Год: 2019
Жанр: Neue Deutsche Härte
Треки: ['Deutschland', 'Radio', 'Zeig dich', 'Ausländer', 'Sex', 'Puppe',
'Was ich liebe', 'Diamant', 'Weit weg', 'Tattoo', 'Hallomann']
Воспроизводится трек 7: Was ich liebe
```

Решение:

```
class MusicAlbum:
    def __init__(self, title, artist, release_year, genre, tracklist):
        self.title = title
        self.artist = artist
        self.release_year = release_year
        self.genre = genre
        self.tracklist = tracklist

    def play_track(self, track_number):
        print(f"Воспроизводится трек  {track_number}:
{self.tracklist[track_number - 1]}")

    def play_random_track(self):
        track_number = random.randint(1, len(self.tracklist))
        self.play_track(track_number)
```

Пример 2

Создайте класс **Student**, который имеет:

- атрибуты **name**, **age**, **grade**, **scores** (имя, возраст, класс, оценки);
- метод average_score – для вычисления среднего балла успеваемости.

Пример использования:

```
student2 = Student("Егор Данилов", 12, "5В", [5, 4, 4, 5])
print("Имя:", student2.name)
print("Возраст:", student2.age)
print("Класс:", student2.grade)
print("Оценки:", *student2.scores)
print("Средний балл:", student2.average_score())
```

Вывод:

```
Имя: Егор Данилов
Возраст: 12
Класс: 5В
Оценки: 5 4 4 5
Средний балл: 4.5
```

Решение:

```
class Student:
    def __init__(self, name, age, grade, scores):
        self.name = name
        self.age = age
        self.grade = grade
        self.scores = scores

    def average_score(self):
        return sum(self.scores) / len(self.scores)
```

Пример 3

Напишите класс **Recipe** с двумя методами:

- print_ingredients(self) – выводит список продуктов, необходимых для приготовления блюда;
- cook(self) – сообщает название выбранного рецепта и уведомляет о готовности блюда.

Пример использования:

```
# создаем рецепт спагетти болоньезе
spaghetti = Recipe("Спагетти болоньезе", ["Спагетти", "Фарш", "Томатный
соус", "Лук", "Чеснок", "Соль"])

# печатаем список продуктов для рецепта спагетти
spaghetti.print_ingredients()

# готовим спагетти
spaghetti.cook()

# создаем рецепт для кекса
cake = Recipe("Кекс", ["Мука", "Яйца", "Молоко", "Сахар", "Сливочное масло",
"Соль", "Ванилин"])

# печатаем рецепт кекса
cake.print_ingredients()

# готовим кекс
cake.cook()
```

Вывод:

```
Ингредиенты для Спагетти болоньезе:
- Спагетти
- Фарш
- Томатный соус
```

```
- Лук
- Чеснок
- Соль
Сегодня мы готовим Спагетти болоньезе.
Выполняем инструкцию по приготовлению блюда Спагетти болоньезе...
Блюдо Спагетти болоньезе готово!
Ингредиенты для Кекс:
- Мука
- Яйца
- Молоко
- Сахар
- Сливочное масло
- Соль
- Ванилин
Сегодня мы готовим Кекс.
Выполняем инструкцию по приготовлению блюда Кекс...
Блюдо Кекс готово!
```

Решение:

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

    def print_ingredients(self):
        print(f"Ингредиенты для {self.name}:")
        for ingredient in self.ingredients:
            print(f"- {ingredient}")

    def cook(self):
        print(f"Сегодня мы готовим {self.name}.")
        print(f"Выполняем инструкцию по приготовлению блюда {self.name}...")
        print(f"Блюдо {self.name} готово!")
```

Пример 4

Напишите суперкласс **Publisher** (издательство) и два подкласса **BookPublisher** (книжное издательство) и **NewspaperPublisher** (газетное издательство).

Родительский класс Publisher имеет два атрибута **name** и **location** (название, расположение) и два метода:

- get_info(self) – предоставляет информацию о названии и расположении издательства;
- publish(self, message) – выводит информацию об издании, которое находится в печати.

Подклассы **BookPublisher** и **NewspaperPublisher** используют метод super().__init__(name, location) суперкласса для вывода информации о своих названии и расположении, и кроме того, имеют собственные атрибуты:

- BookPublisher – **num_authors** (количество авторов).
- NewspaperPublisher– **num_pages** (количество страниц в газете).

Пример использования:

```
publisher = Publisher("АБВГД Пресс", "Москва")
publisher.publish("Справочник писателя")

book_publisher = BookPublisher("Важные Книги", "Самара", 52)
```

```
book_publisher.publish("Приключения Чебурашки", "В.И. Пупкин")

newspaper_publisher = NewspaperPublisher("Московские вести", "Москва", 12)
newspaper_publisher.publish("Новая версия Midjourney будет платной")
```

Вывод:

```
Готовим "Справочник писателя" к публикации в АБВГД Пресс (Москва)
Передаем рукопись 'Приключения Чебурашки', написанную автором В.И. Пупкин в
издательство Важные Книги (Самара)
Печатаем свежий номер со статьей "Новая версия Midjourney будет платной" на
главной странице в издательстве Московские вести (Москва)
```

Решение:

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

    def get_info(self):
        return f"{self.name} ({self.location})"

    def publish(self, message):
        print(f'Готовим "{message}" к публикации в {self.get_info()}')

class BookPublisher(Publisher):
    def __init__(self, name, location, num_authors):
        super().__init__(name, location)
        self.num_authors = num_authors

    def publish(self, title, author):
        print(f"Передаем рукопись '{title}', написанную автором {author} в
издательство {self.get_info()}")

class NewspaperPublisher(Publisher):
    def __init__(self, name, location, num_pages):
        super().__init__(name, location)
        self.num_pages = num_pages

    def publish(self, headline):
        print(f'Печатаем свежий номер со статьей "{headline}" на главной
странице в издательстве {self.get_info()}')
```

Пример 5

Создайте класс **BankAccount**, который имеет следующие **свойства**:

- **balance** – приватный атрибут для хранения текущего баланса счета;
- **interest_rate** –приватный атрибут для процентной ставки;
- **transactions** – приватный атрибут для списка всех операций, совершенных по счету.

Класс **BankAccount** должен иметь следующие **методы**:

- deposit(amount) – добавляет сумму к балансу и регистрирует транзакцию;
- withdraw(amount) – вычитает сумму из баланса и записывает транзакцию;

- add_interest()– добавляет проценты к счету на основе interest_rate и записывает транзакцию;
- history()– печатает список всех операций по счету.

Пример использования:

```
# создаем объект счета с балансом 100000 и процентом по вкладу 0.05
account = BankAccount(100000, 0.05)

# вносим 15 тысяч на счет
account.deposit(15000)

# снимаем 7500 рублей
account.withdraw(7500)

# начисляем проценты по вкладу
account.add_interest()

# печатаем историю операций
account.history()
```

Вывод:

```
Внесение наличных на счет: 15000
Снятие наличных: 7500
Начислены проценты по вкладу: 5375.0
```

Решение:

```
class BankAccount:
    def __init__(self, balance, interest_rate):
        self.__balance = balance
        self.__interest_rate = interest_rate
        self.__transactions = []

    def deposit(self, amount):
        self.__balance += amount
        self.__transactions.append(f"Внесение наличных на счет: {amount}")

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
            self.__transactions.append(f"Снятие наличных: {amount}")
        else:
            print("Недостаточно средств на счете")

    def add_interest(self):
        interest = self.__balance * self.__interest_rate
        self.__balance += interest
        self.__transactions.append(f"Начислены проценты по вкладу:
{interest}")

    def history(self):
        for transaction in self.__transactions:
            print(transaction)
```

Пример 6

Создайте класс **Employee** (сотрудник), который имеет следующие **приватные** свойства:

- **name** – имя сотрудника;

- **age** – возраст;
- **salary** – оклад;
- **bonus** – премия.

Класс **Employee** должен иметь следующие методы:

- get_name()– возвращает имя сотрудника;
- get_age()– возвращает возраст;
- get_salary() – возвращает зарплату сотрудника;
- set_bonus(bonus) – устанавливает свойство **bonus**;
- get_bonus() – возвращает бонус для сотрудника;
- get_total_salary() – возвращает общую зарплату сотрудника (оклад + бонус).

Пример использования:

```
# создаем сотрудника с именем, возрастом и зарплатой
employee = Employee("Марина Арефьева", 30, 90000)

# устанавливаем бонус для сотрудника
employee.set_bonus(15000)

# выводим имя, возраст, зарплату, бонус и общую зарплату сотрудника
print("Имя:", employee.get_name())
print("Возраст:", employee.get_age())
print("Зарплата:", employee.get_salary())
print("Бонус:", employee.get_bonus())
print("Итого начислено:", employee.get_total_salary())
```

Вывод:

```
Имя: Марина Арефьева
Возраст: 30
Зарплата: 90000
Бонус: 15000
Итого начислено: 105000
```

Решение:

```
class Employee:
    def __init__(self, name, age, salary):
        self.__name = name
        self.__age = age
        self.__salary = salary
        self.__bonus = 0

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def get_salary(self):
        return self.__salary

    def set_bonus(self, bonus):
        self.__bonus = bonus

    def get_bonus(self):
        return self.__bonus
```

```
    def get_total_salary(self):
        return self.__salary + self.__bonus
```

Пример 7

Напишите класс **Animal**, обладающий свойствами **name**, **species**, **legs**, в которых хранятся данные о кличке, виде и количестве ног животного. Класс также должен иметь два метода – voice() и move(), которые сообщают о том, что животное подает голос и двигается.

Создайте два подкласса – **Dog** и **Bird**. Подкласс **Dog** имеет атрибут **breed** (порода) и метод bark(), который сообщает о том, что собака лает. Подкласс **Bird** обладает свойством **wingspan** (размах крыльев) и методом fly(), который уведомляет о полете птицы.

Пример использования:

```
dog = Dog("Геральт", "доберман", 4)
bird = Bird("Вася", "попугай", 2)
dog.voice()
bird.voice()
dog.move()
bird.move()
dog.bark()
bird.fly()
```

Вывод:

```
Геральт подает голос
Вася подает голос
Геральт дергает хвостом
Вася дергает хвостом
доберман Геральт лает
попугай Вася летает
```

Решение:

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

    def voice(self):
        print(f"{self.name} подает голос")

    def move(self):
        print(f"{self.name} дергает хвостом")

class Dog(Animal):
    def __init__(self, name, breed, legs):
        super().__init__(name, breed, legs)
        self.breed = breed

    def bark(self):
        print(f"{self.breed} {self.name} лает")

class Bird(Animal):
    def __init__(self, name, species, wingspan):
        super().__init__(name, species, 2)
```

```
        self.wingspan = wingspan

    def fly(self):
        print(f"{self.species} {self.name} летает")
```

Пример 8

Создайте класс **Shape** (геометрическая фигура) со свойствами **name** и **color** (название и цвет). У этого класса должны быть три подкласса – **Circle** (окружность), **Rectangle** (прямоугольник), и **Triangle** (треугольник). Каждый подкласс наследует атрибут **color** и методdescribe() родительского класса **Shape**, и при этом имеет дополнительные свойства и методы:

- **Circle** – атрибут **radius** и метод area() для вычисления площади.
- **Rectangle** – атрибуты **length** и **width**, свой метод area().
- **Triangle** – атрибуты **base** и **height** (основание и высота), собственный метод area().

Пример использования:

```
circle = Circle("красный", 5)
rectangle = Rectangle("синий", 3, 4)
triangle = Triangle("фиолетовый", 6, 8)
circle.describe()
rectangle.describe()
triangle.describe()
print(f"Площадь треугольника {triangle.area()}, окружности {circle.area()},
прямоугольника {rectangle.area()} см.")
```

Вывод:

```
Это геометрическая фигура, цвет - красный.
Это окружность. Радиус - 5 см, цвет - красный.
Это геометрическая фигура, цвет - синий.
Это синий прямоугольник. Длина - 3 см, ширина - 4 см.
Это геометрическая фигура, цвет - фиолетовый.
Это фиолетовый треугольник с основанием 6 см и высотой 8 см.
Площадь треугольника 24.0, окружности 78.5, прямоугольника 12 см.
```

Решение:

```
class Shape:
    def __init__(self, color):
        self.color = color

    def describe(self):
        print(f"Это геометрическая фигура, цвет - {self.color}.")


class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

    def describe(self):
        super().describe()
```

```
        print(f"Это окружность. Радиус - {self.radius} см, цвет -
{self.color}.")


class Rectangle(Shape):
    def __init__(self, color, length, width):
        super().__init__(color)
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def describe(self):
        super().describe()
        print(f"Это {self.color} прямоугольник. Длина - {self.length} см,
ширина - {self.width} см.")


class Triangle(Shape):
    def __init__(self, color, base, height):
        super().__init__(color)
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

    def describe(self):
        super().describe()
        print(f"Это {self.color} треугольник с основанием {self.base} см и
высотой {self.height} см.")
```

Пример 9

Для ПО кондитерской фабрики нужно написать родительский класс **Candy** (Конфеты). Этот класс имеет атрибуты **name**, **price**, **weight** (наименование, цена, вес). Подклассы **Chocolate**, **Gummy**, **HardCandy** (шоколад, жевательный мармелад, леденец) наследуют все атрибуты суперкласса **Candy**. Кроме того, у них есть и свои атрибуты:

- **Chocolate** – **cocoa_percentage** (процент содержания какао) и **chocolate_type** (сорт шоколада).
- **Gummy** – **flavor** и **shape** (вкус и форма).
- **HardCandy** – **flavor** и **filled** (вкус и начинка).

Пример использования:

```
chocolate = Chocolate(name="Швейцарские луга", price=325.50, weight=220,
cocoa_percentage=40, chocolate_type="молочный")
gummy = Gummy(name="Жуй-жуй", price=76.50, weight=50, flavor="вишня",
shape="медведь")
hard_candy = HardCandy(name="Crazy Фрукт", price=35.50, weight=25,
flavor="манго", filled=True)

print("Шоколадные конфеты:")
print(f"Название конфет: {chocolate.name}")
print(f"Стоимость: {chocolate.price} руб")
print(f"Вес брутто: {chocolate.weight} г")
```

```
print(f"Процент содержания какао: {chocolate.cocoa_percentage}")
print(f"Тип шоколада: {chocolate.chocolate_type}")
print("\nМармелад жевательный:")
print(f"Название конфет: {gummy.name}")
print(f"Стоимость: {gummy.price} руб")
print(f"Вес брутто: {gummy.weight} г")
print(f"Вкус: {gummy.flavor}")
print(f"Форма: {gummy.shape}")
print("\nФруктовые леденцы:")
print(f"Название конфет: {hard_candy.name}")
print(f"Стоимость: {hard_candy.price} руб")
print(f"Вес брутто: {hard_candy.weight} г")
print(f"Вкус: {hard_candy.flavor}")
print(f"Начинка: {hard_candy.filled}")
```

Вывод:

```
Шоколадные конфеты:
Название конфет: Швейцарские луга
Стоимость: 325.5 руб
Вес брутто: 220 г
Процент содержания какао: 40
Тип шоколада: молочный

Мармелад жевательный:
Название конфет: Жуй-жуй
Стоимость: 76.5 руб
Вес брутто: 50 г
Вкус: вишня
Форма: медведь

Фруктовые леденцы:
Название конфет: Crazy Фрукт
Стоимость: 35.5 руб
Вес брутто: 25 г
Вкус: манго
Начинка: True
```

Решение:

```
class Candy:
    def __init__(self, name, price, weight):
        self.name = name
        self.price = price
        self.weight = weight

class Chocolate(Candy):
    def __init__(self, name, price, weight, cocoa_percentage,
chocolate_type):
        super().__init__(name, price, weight)
        self.cocoa_percentage = cocoa_percentage
        self.chocolate_type = chocolate_type

class Gummy(Candy):
    def __init__(self, name, price, weight, flavor, shape):
        super().__init__(name, price, weight)
        self.flavor = flavor
        self.shape = shape

class HardCandy(Candy):
    def __init__(self, name, price, weight, flavor, filled):
```

```
        super().__init__(name, price, weight)
        self.flavor = flavor
        self.filled = filled
```

Пример 10

Для военной игры-стратегии нужно написать класс **Soldier** (солдат). Класс имеет атрибуты **name**, **rank** и **service_number** (имя, воинское звание, порядковый номер), причем звание и номер – **приватные** свойства.

Напишите методы для:

- получения воинского звания;
- подтверждения порядкового номера;
- повышения в звании;
- понижения в звании.

Кроме того, нужно создать декоратор для вывода информации о персонаже.

Пример использования:

```
soldier1 = Soldier("Иван Сусанин", "рядовой", "12345")
soldier1.get_rank()
soldier1.promote()
soldier1.demote()
```

Вывод:

```
Создан новый игровой персонаж типа Soldier с атрибутами: {'name': 'Иван
Сусанин', '_Soldier__rank': 'рядовой', '_Soldier__service_number': '12345'}
Персонаж Иван Сусанин имеет звание рядовой
Иван Сусанин повышен в звании, он теперь ефрейтор
Иван Сусанин понижен в звании, он теперь рядовой
```

Решение:

```
RANKS = ["рядовой", "ефрейтор", "младший сержант", "сержант", "старший
сержант",
         "прапорщик", "старший прапорщик"]

def print_info(cls):
    class NewClass(cls):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            print(f"Создан новый игровой персонаж типа {cls.__name__} с
атрибутами: {self.__dict__}")

        def get_rank(self):
            print(f"Персонаж {self.name} имеет звание {self._Soldier__rank}")

        def promote(self):
            super().promote()
            print(f"{self.name} повышен в звании, он теперь
{self._Soldier__rank}")

        def demote(self):
            super().demote()
            print(f"{self.name} понижен в звании, он теперь
{self._Soldier__rank}")

    return NewClass
```

```
@print_info
class Soldier:
    def __init__(self, name, rank, service_number):
        self.name = name
        self.__rank = rank
        self.__service_number = service_number

    def verify_service_number(self, service_number):
        return self.__service_number == service_number

    def promote(self):
        if self.__rank in RANKS[:-1]:
            self.__rank = RANKS[RANKS.index(self.__rank) + 1]

    def demote(self):
        if self.__rank in RANKS[1:]:
            self.__rank = RANKS[RANKS.index(self.__rank) - 1]
```

Пример 11

В далекой-далекой галактике Федерация ведет ожесточенную войну с клингонами. Звездолеты Федерации оснащены мощными фазерами, а клингонские корабли – смертоносными фотонными торпедами. Обе стороны разработали усовершенствованные варп-двигатели для перемещения со сверхсветовой скоростью, и оборудовали свои корабли системами самоуничтожения на случай чрезвычайной ситуации.

Для игры, посвященной этой войне, нужно создать абстрактный класс **Starship** с методами **warp_speed()**, **fire_weapon()** и **self_destruct()**. Кроме того, нужно создать два подкласса **FederationStarship** и **KlingonWarship**, которые наследуют абстрактные методы **Starship** и реализуют свои собственные версии методов **warp_speed()**, **fire_weapon()** и **self_destruct()**.

Пример использования:

```
enterprise = FederationStarship()
bird_of_prey = KlingonWarship()

enterprise.warp_speed()
bird_of_prey.warp_speed()

enterprise.fire_weapon()
bird_of_prey.fire_weapon()

enterprise.self_destruct()
bird_of_prey.self_destruct()
```

Вывод:

```
Включить варп-двигатели!
Включить маскировочное устройство!
Выпустить фотонные торпеды!
Стрелять из фазеров!
Запускаю систему самоуничтожения...
Запускаю протокол самоуничтожения...
```

Решение:

```
from abc import ABC, abstractmethod

class Starship(ABC):
    @abstractmethod
    def warp_speed(self):
        pass

    @abstractmethod
    def fire_weapon(self):
        pass

    @abstractmethod
    def self_destruct(self):
        pass

class FederationStarship(Starship):
    def warp_speed(self):
        print("Включить варп-двигатели!")

    def fire_weapon(self):
        print("Выпустить фотонные торпеды!")

    def self_destruct(self):
        print("Запускаю систему самоуничтожения...")

class KlingonWarship(Starship):
    def warp_speed(self):
        print("Включить маскировочное устройство!")

    def fire_weapon(self):
        print("Стрелять из фазеров!")

    def self_destruct(self):
        print("Запускаю протокол самоуничтожения...")
```

Пример 12

Для ПО ресторана нужно разработать модуль, помогающий контролировать использование фруктов и овощей на кухне. Создайте абстрактный класс **Ingredient** с методами **get_name()** и **get_quantity()**. Затем создайте два подкласса **Vegetable** и **Fruit**, которые наследуют абстрактные методы от **Ingredient** и реализуют свои собственные версии методов **get_name()** и **get_quantity()**.

Пример использования:

```
  carrot = Vegetable("Морковь", 5)
apple = Fruit("Яблоки", 10)

print(carrot.get_name())
print(carrot.get_quantity())

print(apple.get_name())
print(apple.get_quantity())
```

Вывод:

```
Морковь
5 кг
Яблоки
```

```
10 кг
```

Решение:

```
from abc import ABC, abstractmethod

class Ingredient(ABC):
    @abstractmethod
    def get_name(self):
        pass

    @abstractmethod
    def get_quantity(self):
        pass

class Vegetable(Ingredient):
    def __init__(self, name, quantity):
        self.name = name
        self.quantity = quantity

    def get_name(self):
        return self.name

    def get_quantity(self):
        return f'{self.quantity} кг'

class Fruit(Ingredient):
    def __init__(self, name, quantity):
        self.name = name
        self.quantity = quantity

    def get_name(self):
        return self.name

    def get_quantity(self):
        return f'{self.quantity} кг'
```

Пример 13

Для военной стратегии необходимо создать абстрактный класс **Soldier**. Каждый солдат должен уметь двигаться, защищаться и атаковать, поэтому Soldier имеет три абстрактных метода: **move()**, **attack()** и **defend()**. Два конкретных класса, **Infantry** (пехота) и **Cavalry** (кавалерия), будут наследовать и реализовывать эти методы. В игре также должен быть класс **Army**, который будет добавлять солдат в армию и выполнять операции атаки и защиты.

Чтобы гарантировать, что используются только экземпляры класса Soldier, нужно создать декоратор **validatesoldier**, который будет проверять тип объекта. Если объект не является экземпляром класса Soldier, декоратор выдаст ошибку TypeError. Декоратор будет применяться к методам **move()**, **attack()** и **defend()** классов Infantry и Cavalry.

Пример использования:

```
army = Army()
army.add_soldier(Infantry())
army.add_soldier(Cavalry())
army.add_soldier(Infantry())
```

```
army.add_soldier(Cavalry())

army.attack()
army.defend()
```

Вывод:

```
Пехота передвигается в пешем порядке
Пехота участвует в ближнем бою
Кавалерия передвигается верхом
Кавалерия переходит в атаку
Пехота передвигается в пешем порядке
Пехота участвует в ближнем бою
Кавалерия передвигается верхом
Кавалерия переходит в атаку
Пехота передвигается в пешем порядке
Пехота держит строй
Кавалерия передвигается верхом
Кавалерия защищает фланги
Пехота передвигается в пешем порядке
Пехота держит строй
Кавалерия передвигается верхом
Кавалерия защищает фланги
```

Решение:

```
from abc import ABC, abstractmethod

def validate_soldier(func):
    def wrapper(self):
        if not isinstance(self, Soldier):
            raise TypeError("Объект не является экземпляром Soldier")
        return func(self)
    return wrapper

class Soldier(ABC):
    @abstractmethod
    def move(self):
        pass

    @abstractmethod
    def attack(self):
        pass

    @abstractmethod
    def defend(self):
        pass

class Infantry(Soldier):
    @validate_soldier
    def move(self):
        print("Пехота передвигается в пешем порядке")

    @validate_soldier
    def attack(self):
        print("Пехота участвует в ближнем бою")

    @validate_soldier
    def defend(self):
        print("Пехота держит строй")
```

```
class Cavalry(Soldier):
    @validate_soldier
    def move(self):
        print("Кавалерия передвигается верхом")

    @validate_soldier
    def attack(self):
        print("Кавалерия переходит в атаку")

    @validate_soldier
    def defend(self):
        print("Кавалерия защищает фланги")

class Army:
    def __init__(self):
        self.soldiers = []

    def add_soldier(self, soldier):
        self.soldiers.append(soldier)

    def attack(self):
        for soldier in self.soldiers:
            soldier.move()
            soldier.attack()

    def defend(self):
        for soldier in self.soldiers:
            soldier.move()
            soldier.defend()
```

Пример 14

Палеонтологам, работающим в заповеднике для динозавров, понадобилось ПО для отслеживания множества травоядных и плотоядных подопечных. Данные, которые нужно учитывать по каждому динозавру – имя, вид, рост, вес и рацион питания.

Создайте абстрактный класс **Dinosaur** с методами **get_personal_name()**, **get_breed()**, **get_height()**, **get_weight()** и **get_diet()**. Затем создайте два подкласса **Carnivore** (плотоядный) и **Herbivore** (травоядный), которые наследуют методы Dinosaur и реализуют свои собственные версии get_personal_name(), get_breed(), get_height(), get_weight() и get_diet(). Кроме того, создайте класс **DinosaurPark**, который содержит список динозавров и имеет методы **list_dinosaurs()**, **list_carnivores()** и **list_herbivores()** для вывода списков a) всех динозавров, b) плотоядных и c) травоядных особей.

Пример использования:

```
t_rex = Carnivore('Тираннозавр', 'Рекс', 4800, 560)
velociraptor = Carnivore('Велоцираптор', 'Зубастик', 30, 70)
stegosaurus = Herbivore('Стегозавр', 'Стегга', 7100, 420)
triceratops = Herbivore('Трицератопс', 'Трипси', 8000, 290)

park = DinosaurPark()

park.add_dinosaur(t_rex)
park.add_dinosaur(velociraptor)
park.add_dinosaur(stegosaurus)
```

```
park.add_dinosaur(triceratops)


for dinosaur in park.list_dinosaurs():
    print(f'Имя: {dinosaur[0]}\n'
    f'Вид: {dinosaur[1]}\n'
    f'Вес: {dinosaur[2]} кг\n'
    f'Рост: {dinosaur[3]} см\n'
    f'Рацион: {dinosaur[4]}\n')
```

Вывод:

```
Имя: Рекс
Вид: Тираннозавр
Вес: 4800 кг
Рост: 560 см
Рацион: Плотоядный

Имя: Зубастик
Вид: Велоцираптор
Вес: 30 кг
Рост: 70 см
Рацион: Плотоядный

Имя: Стегга
Вид: Стегозавр
Вес: 7100 кг
Рост: 420 см
Рацион: Травоядный

Имя: Трипси
Вид: Трицератопс
Вес: 8000 кг
Рост: 290 см
Рацион: Травоядный
```

Решение:

```
from abc import ABC, abstractmethod

class Dinosaur(ABC):
    @abstractmethod
    def get_personal_name(self):
        pass

    @abstractmethod
    def get_breed(self):
        pass

    @abstractmethod
    def get_height(self):
        pass

    @abstractmethod
    def get_weight(self):
        pass

    @abstractmethod
    def get_diet(self):
        pass
```

```
class Carnivore(Dinosaur):
    def __init__(self, breed, personal_name, height, weight):
        self.breed = breed
        self.personal_name = personal_name
        self.height = height
        self.weight = weight

    def get_personal_name(self):
        return self.personal_name

    def get_breed(self):
        return self.breed

    def get_height(self):
        return self.height

    def get_weight(self):
        return self.weight

    def get_diet(self):
        return 'Плотоядный'

class Herbivore(Dinosaur):
    def __init__(self, breed, personal_name, height, weight):
        self.breed = breed
        self.personal_name = personal_name
        self.height = height
        self.weight = weight

    def get_personal_name(self):
        return self.personal_name

    def get_breed(self):
        return self.breed

    def get_height(self):
        return self.height

    def get_weight(self):
        return self.weight

    def get_diet(self):
        return 'Травоядный'

class DinosaurPark:
    def __init__(self):
        self.dinosaurs = []

    def add_dinosaur(self, dinosaur):
        self.dinosaurs.append(dinosaur)


    def list_dinosaurs(self):
        return [(dinosaur.get_personal_name(), dinosaur.get_breed(),
dinosaur.get_height(), dinosaur.get_weight(), dinosaur.get_diet()) for
dinosaur in self.dinosaurs]


    def list_carnivores(self):
```

```
        return [(dinosaur.get_personal_name(), dinosaur.get_breed(),
dinosaur.get_height(), dinosaur.get_weight()) for dinosaur in self.dinosaurs
if isinstance(dinosaur, Carnivore)]


    def list_herbivores(self):
        return [(dinosaur.get_personal_name(), dinosaur.get_breed(),
dinosaur.get_height(), dinosaur.get_weight()) for dinosaur in self.dinosaurs
if isinstance(dinosaur, Herbivore)]
```

Пример 15

Для учета музыкальных инструментов в оркестре нужно создать абстрактный класс **Instrument** с методами **get_name()**, **get_type()**, **get_sound()** и **play()**. Два подкласса **StringedInstrument** (струнные) и **PercussionInstrument** (ударные) наследуют методы Instrument и реализуют свои собственные версии методов get_name(), get_type(), get_sound() и play(). Кроме того, необходимо реализовать класс **Orchestra**: он добавляет новые инструменты и имеет методы **list_instruments()**, **list_stringed_instruments()**, **list_percussion_instruments()**, которые выводят списки a) всех инструментов, b) ударных, c) струнных.

Пример использования:

```
chello = StringedInstrument("виолончель", "струнный инструмент", "Strum")
maracas = PercussionInstrument("маракасы", "ударный инструмент", "Maracas")
violin = StringedInstrument("скрипка", "струнный инструмент", "Virtuso")
drums = PercussionInstrument("барабан", "ударный инструмент", "Beat")

orchestra = Orchestra()
orchestra.add_instrument(chello)
orchestra.add_instrument(maracas)
orchestra.add_instrument(violin)
orchestra.add_instrument(drums)

print("В оркестре есть инструменты:", ',
'.join(orchestra.list_instruments()))
print("Струнные инструменты:", ',
'.join(orchestra.list_stringed_instruments()))
print("Ударные инструменты:", ',
'.join(orchestra.list_percussion_instruments()))

print(chello.play())
print(drums.play())
```

Вывод:

```
В оркестре есть инструменты: виолончель, маракасы, скрипка, барабан
Струнные инструменты: виолончель, скрипка
Ударные инструменты: маракасы, барабан
Звучит струнный инструмент виолончель
Звучит ударный инструмент барабан
```

Решение:

```
from abc import ABC, abstractmethod

def print_decorator(func):
```

```
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result
    return wrapper

class Instrument(ABC):
    @abstractmethod
    def get_name(self):
        pass

    @abstractmethod
    def get_type(self):
        pass

    @abstractmethod
    def get_sound(self):
        pass

    @abstractmethod
    def play(self):
        pass

class StringedInstrument(Instrument):
    def __init__(self, name, type, sound):
        self.name = name
        self.type = type
        self.sound = sound

    def get_name(self):
        return self.name

    def get_type(self):
        return self.type

    def get_sound(self):
        return self.sound


    def play(self):
        return f"Звучит {self.type} {self.name}"

class PercussionInstrument(Instrument):
    def __init__(self, name, type, sound):
        self.name = name
        self.type = type
        self.sound = sound

    def get_name(self):
        return self.name

    def get_type(self):
        return self.type

    def get_sound(self):
        return self.sound

    def play(self):
        return f"Звучит {self.type} {self.name}"

class Orchestra:
```

```
    def __init__(self):
        self.instruments = []

    def add_instrument(self, instrument):
        self.instruments.append(instrument)

    @print_decorator
    def list_instruments(self):
        return [instrument.get_name() for instrument in self.instruments]

    @print_decorator
    def list_stringed_instruments(self):
        return [instrument.get_name() for instrument in self.instruments if
isinstance(instrument, StringedInstrument)]

    @print_decorator
    def list_percussion_instruments(self):
        return [instrument.get_name() for instrument in self.instruments if
isinstance(instrument, PercussionInstrument)]
```

Пример 16

Напишите класс **FilmCatalogue** (каталог фильмов), который отвечает за ведение фильмотеки. FilmCatalogue должен поддерживать различные типы кинокартин, чтобы пользователи могли искать фильмы по определенному жанру. Для этого необходимо создать новые классы для различных жанров (например, **Horror**, **Action**, **Romance**), которые наследуют класс **Movie** и переопределяют метод **play()** для вывода информации о том, к какому жанру относится фильм.

Пример использования:

```
my_catalogue = FilmCatalogue()

my_catalogue.add_movie(Drama("Крестный отец", "Френсис Ф. Коппола"))
my_catalogue.add_movie(Comedy("Ночные игры", "Джон Фрэнсис Дейли, Джонатан М.
Голдштейн"))
my_catalogue.add_movie(Horror("Дракула Брэма Стокера", "Френсис Ф. Коппола"))
my_catalogue.add_movie(Action("Крушение", "Жан-Франсуа Рише"))
my_catalogue.add_movie(Romance("Честная куртизанка", "Маршалл Херсковиц"))

my_catalogue.play_all_movies()

print(f"\nНайдены фильмы ужасов:")
for movie in my_catalogue.search_movies_by_genre(Horror):
    print(movie.title)

print(f"\nЗапускаем фильм из жанра 'Мелодрамы':")
my_catalogue.play_movies_by_genre(Romance)
```

Вывод:

```
Включаем драму 'Крестный отец' реж. Френсис Ф. Коппола.
Включаем комедию 'Ночные игры' реж. Джон Фрэнсис Дейли, Джонатан М.
Голдштейн.
Включаем фильм ужасов 'Дракула Брэма Стокера' реж. Френсис Ф. Коппола.
Включаем боевик 'Крушение' реж. Жан-Франсуа Рише.
Включаем мелодраму 'Честная куртизанка' реж. Маршалл Херсковиц.

Найдены фильмы ужасов:
```

```
Дракула Брэма Стокера

Запускаем фильм из жанра 'Мелодрамы':
Включаем мелодраму 'Честная куртизанка' реж. Маршалл Херсковиц.
```

Решение:

```
class Movie:
    def __init__(self, title, director):
        self.title = title
        self.director = director

    def play(self):
        print(f"Собираемся смотреть фильм '{self.title}' реж.
{self.director}.")

class Comedy(Movie):
    def play(self):
        print(f"Включаем комедию '{self.title}' реж. {self.director}.")

class Drama(Movie):
    def play(self):
        print(f"Включаем драму '{self.title}' реж. {self.director}.")

class Horror(Movie):
    def play(self):
        print(f"Включаем фильм ужасов '{self.title}' реж. {self.director}.")

class Action(Movie):
    def play(self):
        print(f"Включаем боевик '{self.title}' реж. {self.director}.")

class Romance(Movie):
    def play(self):
        print(f"Включаем мелодраму '{self.title}' реж. {self.director}.")

class FilmCatalogue:
    def __init__(self):
        self.movies = []

    def add_movie(self, movie):
        self.movies.append(movie)

    def play_all_movies(self):
        for movie in self.movies:
            movie.play()

    def search_movies_by_genre(self, genre):
        return [movie for movie in self.movies if isinstance(movie, genre)]

    def play_movies_by_genre(self, genre):
        movies = self.search_movies_by_genre(genre)
        for movie in movies:
            movie.play()
```

Пример 17

Для CRM винодельни нужно написать модуль, отвечающий за учет красных, белых и розовых вин, каждое из которых имеет свое название, сорт винограда, год и температуру подачи. Создайте базовый класс **Wine** с

атрибутами **name**, **grape** и **year**. Затем создайте три подкласса **RedWine**, **WhiteWine** и **RoseWine**, которые наследуют методы и атрибуты от Wine и реализуют свои собственные версии метода **serve()**. Кроме того, создайте класс **Winery**, который ведет список вин и имеет метод **serve_wines()**, вызывающий метод **serve()** для каждого вина.

Пример использования:

```
winery = Winery()
winery.add_wine(RedWine("Cabernet Sauvignon", "Каберне Совиньон", 2015))
winery.add_wine(WhiteWine("Chardonnay", "Шардоне", 2018))
winery.add_wine(RoseWine("Grenache", "Гренаш", 2020))
winery.serve_wines()
```

Вывод:

```
Красное вино 'Cabernet Sauvignon', сделанное из винограда сорта Каберне
Совиньон в 2015 году, рекомендуем подавать комнатной температуры.
Белое вино 'Chardonnay', сделанное из винограда сорта Шардоне в 2018 году,
рекомендуем подавать хорошо охлажденным.
Розовое вино 'Grenache', сделанное из винограда сорта Гренаш в 2020 году,
рекомендуем подавать слегка охлажденным.
```

Решение:

```
class Wine:
    def __init__(self, name, grape, year):
        self.name = name
        self.grape = grape
        self.year = year

class RedWine(Wine):
    def serve(self):
        print(f"Красное вино '{self.name}', сделанное из винограда сорта
{self.grape} в {self.year} году, рекомендуем подавать комнатной
температуры.")

class WhiteWine(Wine):
    def serve(self):
        print(f"Белое вино '{self.name}', сделанное из винограда сорта
{self.grape} в {self.year} году, рекомендуем подавать хорошо охлажденным.")

class RoseWine(Wine):
    def serve(self):
        print(f"Розовое вино '{self.name}', сделанное из винограда сорта
{self.grape} в {self.year} году, рекомендуем подавать слегка охлажденным.")

class Winery:
    def __init__(self):
        self.wines = []

    def add_wine(self, wine):
        self.wines.append(wine)

    def serve_wines(self):
        for wine in self.wines:
            wine.serve()
```

Пример 18

Для ПО аэропорта нужно разработать модуль, отслеживающий пассажирские и грузовые самолеты, которые отличаются моделью, производителем, вместимостью и грузоподъемностью. Создайте базовый класс **Aircraft** (воздушное судно) с атрибутами **model**, **manufacturer** и **capacity**. Затем создайте два подкласса **PassengerAircraft** и **CargoAircraft**, которые наследуют атрибуты и методы от Aircraft и реализуют свои собственные версии метода **fly()**. В дополнение создайте класс Airport, который содержит список самолетов и имеет метод **takeoff()**, вызывающий метод **fly()** для каждого самолета.

Пример использования:

```
airport = Airport()
airport.add_aircraft(PassengerAircraft("Boeing 747", "Боинг", 416))
airport.add_aircraft(CargoAircraft("Airbus A330", "Эйрбас", 70))
airport.add_aircraft(PassengerAircraft("Boeing 777", "Боинг", 396))
airport.takeoff()
```

Вывод:

```
Пассажирский самолет 'Boeing 747' вместимостью 416 человек, произведенный
компанией Боинг, поднимается в воздух с пассажирами на борту.
Грузовой самолет 'Airbus A330' с грузоподъемностью 70 т, произведенный
компанией Эйрбас, поднимается в воздух с грузом на борту.
Пассажирский самолет 'Boeing 777' вместимостью 396 человек, произведенный
компанией Боинг, поднимается в воздух с пассажирами на борту.
```

Решение:

```
class Aircraft:
    def __init__(self, model, manufacturer, capacity):
        self.model = model
        self.manufacturer = manufacturer
        self.capacity = capacity

class PassengerAircraft(Aircraft):
    def fly(self):
        print(f"Пассажирский самолет '{self.model}' вместимостью
{self.capacity} человек, произведенный компанией {self.manufacturer},
поднимается в воздух с пассажирами на борту.")

class CargoAircraft(Aircraft):
    def fly(self):
        print(f"Грузовой самолет '{self.model}' с грузоподъемностью
{self.capacity} т, произведенный компанией {self.manufacturer}, поднимается в
воздух с грузом на борту.")

class Airport:
    def __init__(self):
        self.aircrafts = []

    def add_aircraft(self, aircraft):
        self.aircrafts.append(aircraft)

    def takeoff(self):
        for aircraft in self.aircrafts:
            aircraft.fly()
```

Пример 19

Необходимо реализовать модуль, отвечающий за обработку данных о тестировании конфигурации настольных компьютеров и ноутбуков, каждый из которых отличается моделью, процессором, памятью и производительностью. Создайте базовый класс **Computer** с атрибутами **model**, **processor** и **memory**. Затем создайте два подкласса **Desktop** и **Laptop**, которые наследуют атрибуты и методы Computer и реализуют свои собственные версии метода **run()**. В дополнение, создайте класс **ComputerStore**, который содержит список компьютеров и имеет метод **run_tests()**, вызывающий метод **run()** для каждого компьютера. Используйте декораторы для вывода результатов.

Пример использования:

```
store = ComputerStore()
store.add_computer(Desktop("HP Legion", "Intel Core i9-10900K", "64 Гб"))
store.add_computer(Laptop("Dell Xtra", "Intel Core i5 13600K", "32 Гб"))
store.add_computer(Desktop("Lenovo SuperPad", "AMD Ryzen 7 2700X", "16 Гб"))
store.run_tests()
```

Вывод:

```
Начинаем тест производительности...
Запускаем настольный компьютер 'HP Legion' с процессором Intel Core i9-10900K
и 64 Гб RAM.
Тест производительности завершен.
Начинаем тест производительности...
Запускаем ноутбук 'Dell Xtra' с процессором Intel Core i5 13600K и 32 Гб RAM.
Тест производительности завершен.
Начинаем тест производительности...
Запускаем настольный компьютер 'Lenovo SuperPad' с процессором AMD Ryzen 7
2700X и 16 Гб RAM.
Тест производительности завершен.
```

Решение:

```
def performance_test(func):
    def wrapper(*args, **kwargs):
        print("Начинаем тест производительности...")
        result = func(*args, **kwargs)
        print("Тест производительности завершен.")
        return result
    return wrapper

class Computer:
    def __init__(self, model, processor, memory):
        self.model = model
        self.processor = processor
        self.memory = memory

    @performance_test
    def run(self):
        pass

class Desktop(Computer):
    @performance_test
    def run(self):
        print(f"Запускаем настольный компьютер '{self.model}' с процессором
{self.processor} и {self.memory} RAM.")
```

```
class Laptop(Computer):
    @performance_test
    def run(self):
        print(f"Запускаем ноутбук '{self.model}' с процессором
{self.processor} и {self.memory} RAM.")

class ComputerStore:
    def __init__(self):
        self.computers = []

    def add_computer(self, computer):
        self.computers.append(computer)

    def run_tests(self):
        for computer in self.computers:
            computer.run()
```

Пример 20

Определите базовый класс **Cryptocurrency**, имеющий атрибуты:

- **name** – название;
- **symbol** – символ-тикер;
- **minable** – возможность добычи майнингом;
- **rate_to_usd** – текущий курс к доллару;
- **anonymous** – наличие анонимных транзакций.

Затем создайте три подкласса **Nano**, **Iota** и **Stellar**, которые наследуют атрибуты и методы родительского класса **Cryptocurrency**, и обладают дополнительными свойствами, влияющими на размер вознаграждения за майнинг:

- атрибут **block_lattice** у Nano;
- **tangle** у Iota.

Кроме того, нужно реализовать:

1. Декоратор **minable_required**, который проверяет, можно ли майнить криптовалюту перед вызовом метода **mining_reward()**, и выводит сообщение, если ее майнить нельзя.
2. Функцию **print_info**, которая принимает на вход экземпляр криптовалюты и выводит информацию о монете, включая название, символ, возможность добычи, курс к доллару США, анонимность и наличие блок-решетки.

Пример использования:

```
cryptocurrencies = [Nano(block_lattice=True, rate_to_usd=6, anonymous=False),
                    Iota(tangle=True, rate_to_usd=0.4, anonymous=False),
                    Stellar(distributed=False, rate_to_usd=0.15,
anonymous=True)]

for crypto in cryptocurrencies:
    print_info(crypto)
    if crypto.minable:
        print(f"Награда за майнинг: {crypto.mining_reward()} {crypto.symbol}\
n")
```

Вывод:

```
Nano (NANO): добывают майнингом, курс к USD: 6, только публичные транзакции,
блок-решетка
Награда за майнинг: 0.02 NANO

Iota (IOTA): добывают майнингом, курс к USD: 0.4, только публичные транзакции
Награда за майнинг: 0.001 IOTA

Stellar (XLM): не майнится, курс к USD: 0.15, анонимные транзакции
```

Решение:

```
class Cryptocurrency:
    def __init__(self, name, symbol, minable=True, rate_to_usd=1,
anonymous=False):
        self.name = name
        self.symbol = symbol
        self.minable = minable
        self.rate_to_usd = rate_to_usd
        self.anonymous = anonymous

    def __str__(self):
        return f"{self.name} ({self.symbol})"

    def mining_reward(self):
        return None

def minable_required(func):
    def wrapper(crypto):
        if not crypto.minable:
            print("Эту криптовалюту не майнят")
            return None
        return func(crypto)
    return wrapper

class Nano(Cryptocurrency):
    def __init__(self, block_lattice=False, rate_to_usd=5, anonymous=True):
        super().__init__("Nano", "NANO", True, rate_to_usd, anonymous)
        self.block_lattice = block_lattice

    @minable_required
    def mining_reward(self):
        return 0.02 if self.block_lattice else 0.01

class Iota(Cryptocurrency):
    def __init__(self, tangle=False, rate_to_usd=0.3, anonymous=False):
        super().__init__("Iota", "IOTA", True, rate_to_usd, anonymous)
        self.tangle = tangle

    @minable_required
    def mining_reward(self):
        return 0.001 if self.tangle else 0.002

class Stellar(Cryptocurrency):
    def __init__(self, distributed=False, rate_to_usd=0.1, anonymous=True):
        super().__init__("Stellar", "XLM", False, rate_to_usd, anonymous)
        self.distributed = distributed

    def mining_reward(self):
        print("Stellar is not minable")
        return None
```

```
def print_info(crypto):
    minable_str = 'добывают майнингом' if crypto.minable else 'не майнится'
    anonymity_str = 'анонимные транзакции' if crypto.anonymous else 'только
публичные транзакции'
    block_lattice_str = 'блок-решетка' if hasattr(crypto, 'block_lattice')
and crypto.block_lattice else ''

    if block_lattice_str:
        print(f"{crypto}: {minable_str}, курс к USD: {crypto.rate_to_usd},
{anonymity_str}, {block_lattice_str}")
    else:
        print(f"{crypto}: {minable_str}, курс к USD: {crypto.rate_to_usd},
{anonymity_str}")
```

3. Ход работы

3.1. Установка и настройка

3.1.1. Python

Для установки Python необходимо выполнить ряд действий в зависимости от ОС:

- определить разрядность ОС (32- или 64-битная, инструкции для Windows, Mac OS X или Linux);
- открыть страницу загрузки (общая) и загрузить соответствующий дистрибутив;
- выполнить установку (при установке на ОС Windows необходимо установить флажок *Add python.exe to PATH* во время установки).

В дистрибутивах Linux с пакетным менеджером, удобнее использовать его команды, например:

```
sudo apt-get install python3
sudo apt-get install python3-pip
sudo apt-get install idle3
```

После установки запустите терминал и убедитесь, что необходимая версия Python установлена.

3.1.2. IDE

В рамках курса возможности разных сред разработки не отличаются, поэтому выбор остается за Вами.

3.2. Выполнение заданий включает несколько этапов

1. ***Решение***.

Задания предусматривают написание программы на Python (программирование) на базе заготовок в соответствии с заданием, инструкциями и подсказками.

Для решения задания по программированию необходимо:

- открыть заготовку и выполнить решение;
- запустить программу и исправить ошибки (до их отсутствия);
- проверить соответствие вывода на экран примеру.

2. ***Проверка***.

Задания должны проходить проверку. В процессе проверки программа автоматически проверяется на правильность решения, соответствие стандарту оформления и др.

Для проверки задачи:

- настройте среду разработки;
- запустите команду для проверки и исправьте ошибки (до их отсутствия).

3. ***Защита***.

Задания, которые были выполнены и проверены защищаются очно, включая демонстрацию и ответы на дополнительные вопросы.

3.3. Оборудование и материалы

Для выполнения данной лабораторной работы необходим компьютер с установленной операционной системой.

3.4. Указания по технике безопасности.

К выполнению лабораторных работ допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

4. Задания

При выполнении заданий используйте заготовки решений: https://github.com/wegolev/oop

Предупреждение

В данном разделе НЕ должны использоваться сторонние модули для реализации заданной функциональности. Например, при реализации класса РимскоеЧисло не нужно использовать модуль ру-romanify и т.д.

4.3.1. Римское число

Создайте класс Roman (РимскоеЧисло), представляющий римское число и поддерживающий операции +, -, *, /.

Совет

При реализации класса следуйте рекомендациям:

- операции +, -, *, / реализуйте как специальные методы (__add__ и др.);
- методы преобразования имеет смысл реализовать как статические методы, позволяя не создавать экземпляр объекта в случае, если необходимо выполнить только преобразования чисел.

При выполнении задания необходимо построить UML-диаграмма классов приложения

4.3.2. Пицерия

Пиццерия предлагает клиентам три вида пиццы: Пепперони, Барбекю и Дары Моря, каждая из которых определяется тестом, соусом и начинкой.

Требуется спроектировать и реализовать приложение для терминала, позволяющее обеспечить обслуживание посетителей.

Дополнительная информация

В бизнес-процессе работы пиццерии в контексте задачи можно выделить 3 сущности (объекта):

- Терминал: отвечает за взаимодействие с пользователем:
  - вывод меню на экран;
  - прием команд от пользователя (выбор пиццы, подтверждение заказа, оплата и др.);
- Заказ: содержит список заказанных пицц, умеет подсчитывать свою стоимость;

- Пицца: содержит заявленные характеристики пиццы, а также умеет себя подготовить (замесить тесто, собрать ингредиенты и т.д.), испечь, порезать и упаковать.

Т.к. пиццерия реализует несколько видов пиццы, которые различаются характеристиками, логично будет сделать общий класс Пицца, а в дочерних классах (например, классе ПиццаБарбекю) уточнить характеристики конкретной пиццы.

Диаграмма указанных классов в нотации UML приведена на Рисунке 1.

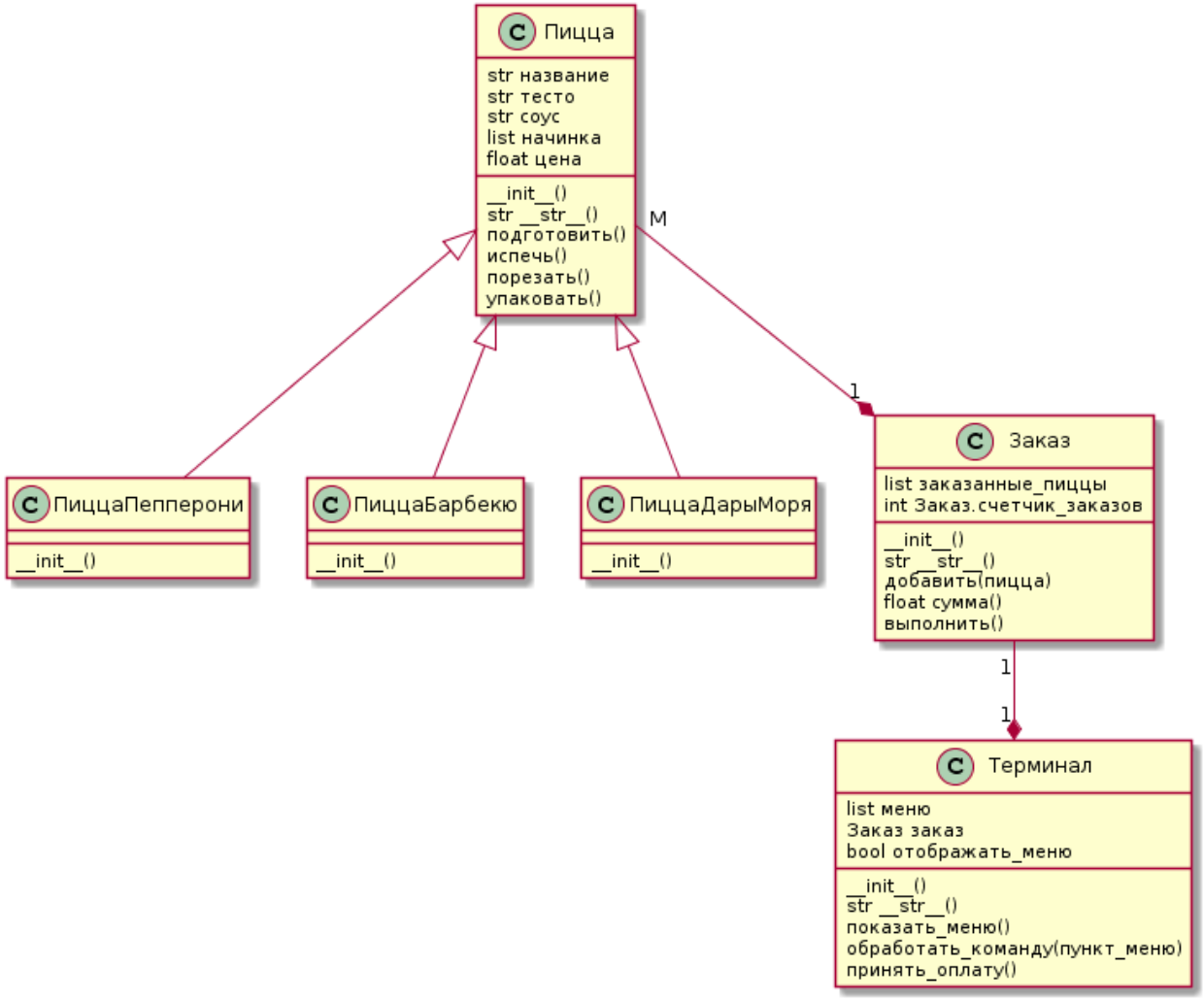

Рисунок 1 - UML-диаграмма классов приложения

Алгоритм работы пользователя с терминалом может выглядеть следующим образом:

1. Терминал отображает список меню.
2. Терминал создает новый заказ.
3. Клиент вводит номер пиццы из меню.
4. Заказ добавляет в список выбранную пиццу.
5. Действия 3-4 повторяются до подтверждения или отмены.
6. Клиент подтверждает заказ (или отменяет).
7. Терминал выставляет счет, отображая информацию о заказе.
8. Терминал принимает оплату.
9. Заказ отдается на выполнение.

4.3.3. Банковские вклады

Банк предлагает ряд вкладов для физических лиц:

- Срочный вклад: расчет прибыли осуществляется по формуле простых процентов;
- Бонусный вклад: бонус начисляется в конце периода как % от прибыли, если вклад больше определенной суммы;
- Вклад с капитализацией процентов.

Реализуйте приложение, которое бы позволило подобрать клиенту вклад по заданным параметрам.

При выполнении задания необходимо построить UML-диаграмма классов приложения

4.3.4. Простой класс

Выберите класс под номером № (Таблица 1), где № Ваш порядковый номер в журнале. При превышении порядка номера отчет ведется сначала по циклу.

Таблица 1 - Классы и их описание (простой класс)

№ п/п	Наименование класса	Описание
1	Vector	Геометрический вектор на плоскости
2	LineSegment	Математический интервал
3	Complex	Комплексное число
4	Time	Время
5	DateTime	Дата/Время
6	Money	Денежная единица
7	Stack	Стек
8	Queue	Очередь
9	Date	Дата
10	Fraction	Обыкновенная дробь

Прежде чем перейти к написанию кода:

- изучите предметную область объекта и доступные операции;
- для каждого поля и метода продумайте его область видимости, а также необходимость использования свойств.

При реализации класс должен содержать:

- специальные методы:
  - __init__(self, ...) - инициализация с необходимыми параметрами;

    - __str__(self) - представление объекта в удобном для человека виде;
    - специальные методы для возможности сложения, разности и прочих операций, которые класс должен поддерживать;
- методы класса:
    - from_string(cls, str_value) - создает объект на основании строки str_value;
- поля, методы, свойства:
    - поля, необходимые для выбранного класса;
    - метод save(self, filename) - сохраняет объект в JSON-файл filename;
    - метод load(self, filename) - загружает объект из JSON-файла filename;
    - прочие методы (не менее 3-х) и свойства, выявленные на этапе изучения класса.

Реализуйте класс в отдельном модуле, а также создайте main.py, который бы тестировал все его возможности.

При выполнении задания необходимо построить UML-диаграмма классов приложения

4.3.5. Класс-контейнер

Создайте класс-контейнер, который будет содержать набор объектов из предыдущей задачи. Например, класс VectorCollection будет содержать объекты класса Vector.

Для класса-контейнера предусмотрите:

- специальные методы:
    - __init__(self, ...) - инициализация с необходимыми параметрами;
    - __str__(self) - представление объекта в удобном для человека виде;
    - __getitem__() - индексация и срез для класса-контейнера.
- поля, методы, свойства:
    - поле _data - содержит набор данных;
    - метод add(self, value) - добавляет элемент value в контейнер;
    - метод remove(self, index) - удаляет элемент из контейнера по индексу index;
    - метод save(self, filename) - сохраняет объект в JSON-файл filename;
    - метод load(self, filename) - загружает объект из JSON-файла filename.

При выполнении задания необходимо построить UML-диаграмму классов приложения

4.3.6. Иерархия классов

Выберите класс под номером № (Таблица 2), где № Ваш порядковый номер в журнале. При превышении порядка номера отчет ведется сначала по циклу.

Таблица 2 - Иерархия классов

№ п/п	Классы	Методы базового класса
1	Плеер, АудиоПлеер, ВидеоПлеер, DvdПлеер	запустить(), остановить()
2	ПищущаяПринадлежность, Карандаш, Ручка, ГелеваяРучка	писать()
3	ТранспортноеСредство, ВодноеТС, КолесноеТС, Автомобиль	ехать()
4	ДенежныйПеревод, ПочтовыйПеревод, БанковскийПеревод, ВалютныйПеревод	выполнить()
5	ПроезднойБилет, БезлимитныйБилет, БилетСОграничением, БилетСОграничениемПоездок	списать_поездку()

Далее:

- выстройте классы в иерархию, продумайте их общие и отличительные характеристики и действия;
- добавьте собственную реализацию методов базового класса в каждый из классов, предусмотрев:
  - необходимые параметры для базовых методов (например, в метод воспроизведения в Dvd-плеере можно передать абстрактный DVD-диск);
  - необходимые поля для функционирования базовых методов (например, при остановке Dvd-плеера имеет смысл сохранить текущую позицию воспроизведения); классы должны содержать как минимум по одному общедоступному, не общедоступному и закрытому полю/методу;
  - вывод на экран работы метода (например, вызов метода остановки в Dvd-плеере должен сообщать на экране, что плеер установлен на определенной позиции).
- по желанию добавьте собственные методы в классы иерархии.

Реализуйте все классы в отдельном модуле, а также создайте main.py, который бы тестировал все его возможности.

По согласованию иерархия может быть расширена или выбрана самостоятельная индивидуальная тема для данной задачи.

При выполнении задания необходимо построить UML-диаграмма классов приложения

5. Содержание отчета

Отчет по лабораторной работе должен быть выполнен в текстовом редакторе и оформлен согласно требованиям. На проверку необходимо сдать следующие файлы:

1. Исходный файл, содержащий готовый отчет и набранный с помощью текстового редактора (например *.doc, *.docx, *.rtf и др.), а также готовый отчет в открытом формате электронных документов *.pdf.
2. Zip-архив, содержащий файлы программного кода выполненных заданий.

Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выравнен по ширине.

Отчет должен содержать титульный лист с темой лабораторной работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

6. Контрольные вопросы:

1. Проблемы императивного (процедурного) подхода в программировании.
2. Понятие объекта и черного ящика (примеры). Парадигма объектно-ориентированного программирования. Цель объектно-ориентированного подхода. Как выглядит разработка приложения в объектно-ориентированном стиле?
3. Понятия класс, объект, поле, метод. Область видимости, виртуальность метода.
4. Основные принципы ООП. Содержание и примеры.
5. Поддержка ООП в Python. Определение простого класса (инициализация, строковое представление, специальные методы).
6. Определение операторов в классах, проверка типов. Случаи, в которых добавление такой функциональности является целесообразным.
7. Атрибуты объекта и атрибуты класса: ключевые различия и варианты использования.
8. Инкапсуляция в Python: особенности, общедоступные (Public), не общедоступные (Non-Public) и закрытые (Private) атрибуты. Принцип универсального доступа, геттеры/сеттеры, свойства.
9. Наследование в Python: особенности, два вида наследования (ключевые моменты, разница, примеры использования). Проверка принадлежности к классу/типу.
10. Полиморфизм в Python: особенности, примеры использования.

11. Множественное наследование: возможности и недостатки. Использование класса Python как структуры из Си.

7. Список рекомендуемых источников по данной теме:

7.1. Официальная документация.

1. Classes. URL: https://docs.python.org/3/tutorial/classes.html.
2. Data model. URL: https://docs.python.org/3/reference/datamodel.html.

7.2. Python.

1. Петров Ю. Программирование на языке высокого уровня (Python). URL: https://www.yuripetrov.ru/edu/python/index.html.
2. Object Oriented Programming. URL: http://anandology.com/python-practice-book/object_oriented_programming.html.
3. Заметки об объектной системе языка Python (часть 1). URL: http://habrahabr.ru/post/114576/.
4. Заметки об объектной системе языка Python (часть 2). URL: http://habrahabr.ru/post/114585/.
5. Improve Your Python: Python Classes and Object Oriented Programming. URL: https://www.jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/.
6. Modules, Classes, and Objects. URL: http://learnpythonthehardway.org/book/ex40.html.
7. A Guide to Python’s Magic Methods. URL: http://www.rafekettler.com/magicmethods.html.
8. Static class variables in Python. URL: http://stackoverflow.com/a/27568860, 8. Advanced Python : Static and Class Methods. URL: http://www.slideshare.net/papaJee/static-and-class-methods. 9. Разница между ``__repr__` и __str__. URL: http://stackoverflow.com/questions/1436703/difference-between-str-and-repr-in-python.

7.3. Прочее.

1. Объектно-ориентированное программирование. URL: https://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование.
2. Коротко об истории объектно-ориентированного программирования. URL: http://habrahabr.ru/post/107940/.
3. ООП с примерами (часть 1). URL: https://megamozg.ru/post/6908/.
4. ООП с примерами (часть 2). URL: https://megamozg.ru/post/6910/.
5. Really Brief Introduction to Object Oriented Design. URL: http://www.fincher.org/tips/General/SoftwareEngineering/ObjectOrientedDesign.shtml.
6. Java Programming Tutorial. Object-oriented Programming (OOP) Basics. URL:

https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html.
7. Define static methods in the following scenarios only. URL: http://stackoverflow.com/a/5313383/396619.
8. Practical UML: A Hands-On Introduction for Developers. URL: http://edn.embarcadero.com/article/31863.
9. Building Skills in Object-Oriented Design. URL: http://www.itmaybeahack.com/homepage/books/oodesign.html.