📘 Справочные материалы к практической работе №5

Объектно-ориентированное программирование в Python

Содержание

---

1. Введение в ООП

1.1 Что такое объектно-ориентированное программирование?

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции "объектов", которые содержат данные (атрибуты) и методы.

1.2 Основные принципы ООП

Четыре столпа ООП:

1. **Инкапсуляция** — сокрытие внутренней реализации объекта
2. **Наследование** — создание новых классов на основе существующих
3. **Полиморфизм** — способность объектов разных типов реагировать на одни и те же сообщения
4. **Абстракция** — выделение существенных характеристик объекта

1.3 Преимущества ООП

- **Модульность** — код организован в логические блоки
- **Повторное использование** — код можно использовать многократно
- **Масштабируемость** — легко добавлять новую функциональность
- **Поддерживаемость** — проще находить и исправлять ошибки

---

2. Классы и объекты

2.1 Основные понятия

Класс — это шаблон или чертеж для создания объектов.
Объект — это экземпляр класса, конкретная реализация.

2.2 Создание первого класса

```
class Car:
    """Класс, представляющий автомобиль"""
    pass
```

```
# Создание объекта (экземпляра класса)
my_car = Car()
print(type(my_car))  # <class '__main__.Car'>
```

2.3 Добавление атрибутов

```
class Car:
    """Класс автомобиля с атрибутами"""

    def __init__(self, brand, model, year):
        self.brand = brand      # Атрибут экземпляра
        self.model = model      # Атрибут экземпляра
        self.year = year        # Атрибут экземпляра
        self.is_running = False # Атрибут по умолчанию

# Создание объектов
car1 = Car("Toyota", "Camry", 2020)
car2 = Car("BMW", "X5", 2019)

print(f"Машина 1: {car1.brand} {car1.model}")  # Toyota Camry
print(f"Машина 2: {car2.brand} {car2.model}")  # BMW X5
```

2.4 Атрибуты класса vs атрибуты экземпляра

```
class Car:
    # Атрибут класса (общий для всех экземпляров)
    wheels = 4
    total_cars = 0

    def __init__(self, brand, model):
        # Атрибуты экземпляра (уникальные для каждого объекта)
        self.brand = brand
        self.model = model
        Car.total_cars += 1  # Увеличиваем счетчик

# Доступ к атрибутам класса
print(Car.wheels)        # 4
print(Car.total_cars)    # 0

car1 = Car("Toyota", "Camry")
car2 = Car("BMW", "X5")

print(Car.total_cars)    # 2
print(car1.wheels)       # 4 (доступ через экземпляр)
print(car2.wheels)       # 4
```

---

3. Конструкторы и self

3.1 Метод **init** (конструктор)

`__init__` — это специальный метод, который автоматически вызывается при создании объекта.

```
class Person:
    def __init__(self, name, age, city="Неизвестно"):
        """
        Конструктор класса Person
```

```
        Args:
            name (str): Имя человека
            age (int): Возраст человека
            city (str): Город проживания (по умолчанию "Неизвестно")
        """
        self.name = name
        self.age = age
        self.city = city
        print(f"Создан объект: {name}")

# Создание объектов
person1 = Person("Алиса", 25)                # Создан объект: Алиса
person2 = Person("Боб", 30, "Москва")        # Создан объект: Боб
```

3.2 Параметр self

`self` — это ссылка на текущий экземпляр класса. Он должен быть первым параметром всех методов экземпляра.

```
class Calculator:
    def __init__(self, initial_value=0):
        self.value = initial_value

    def add(self, number):
        self.value += number
        return self  # Возвращаем self для цепочки вызовов

    def multiply(self, number):
        self.value *= number
        return self

    def get_result(self):
        return self.value

# Использование
calc = Calculator(10)
result = calc.add(5).multiply(2).get_result()
print(result)  # 30

# Цепочка вызовов работает, потому что методы возвращают self
```

3.3 Множественные конструкторы через classmethod

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

    @classmethod
    def from_age(cls, name, age):
        """Альтернативный конструктор: создание через возраст"""
        from datetime import datetime
        birth_year = datetime.now().year - age
        return cls(name, birth_year)

    @classmethod
    def from_string(cls, person_str):
        """Альтернативный конструктор: создание из строки"""
        name, birth_year = person_str.split('-')
```

```
        return cls(name, int(birth_year))

    def get_age(self):
        from datetime import datetime
        return datetime.now().year - self.birth_year

# Разные способы создания объектов
person1 = Person("Алиса", 1990)                  # Обычный конструктор
person2 = Person.from_age("Боб", 25)             # Из возраста
person3 = Person.from_string("Чарли-1985")       # Из строки

print(f"{person2.name}: {person2.get_age()} лет")  # Боб: 25 лет
```

4. Методы и поля

4.1 Типы методов

В Python существует три типа методов:

4.1.1 Методы экземпляра

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance
        self.transaction_history = []

    def deposit(self, amount):
        """Метод экземпляра: работает с конкретным счетом"""
        if amount > 0:
            self.balance += amount
            self.transaction_history.append(f"Пополнение: +{amount}")
            return True
        return False

    def withdraw(self, amount):
        """Метод экземпляра: снятие денег"""
        if 0 < amount <= self.balance:
            self.balance -= amount
            self.transaction_history.append(f"Снятие: -{amount}")
            return True
        return False

    def get_info(self):
        """Метод экземпляра: информация о счете"""
        return f"Счет {self.owner}: {self.balance} тг."
```

4.1.2 Методы класса (@classmethod)

```
class BankAccount:
    total_accounts = 0
    bank_name = "Мой Банк"

    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance
        BankAccount.total_accounts += 1

    @classmethod
```

```
    def get_total_accounts(cls):
        """Метод класса: работает с данными всего класса"""
        return cls.total_accounts

    @classmethod
    def set_bank_name(cls, name):
        """Метод класса: изменение названия банка"""
        cls.bank_name = name

    @classmethod
    def create_savings_account(cls, owner, initial_deposit):
        """Метод класса: создание сберегательного счета"""
        account = cls(owner, initial_deposit)
        account.account_type = "Сберегательный"
        return account

# Использование методов класса
account1 = BankAccount("Алиса", 1000)
account2 = BankAccount("Боб", 500)

print(BankAccount.get_total_accounts())  # 2
BankAccount.set_bank_name("Новый Банк")
print(BankAccount.bank_name)             # Новый Банк

savings = BankAccount.create_savings_account("Чарли", 5000)
print(savings.account_type)              # Сберегательный
```

4.1.3 Статические методы (@staticmethod)

```
class MathUtils:
    """Класс с утилитарными математическими функциями"""

    @staticmethod
    def is_prime(n):
        """Статический метод: проверка числа на простоту"""
        if n < 2:
            return False
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True

    @staticmethod
    def factorial(n):
        """Статический метод: вычисление факториала"""
        if n <= 1:
            return 1
        return n * MathUtils.factorial(n - 1)

    @staticmethod
    def gcd(a, b):
        """Статический метод: наибольший общий делитель"""
        while b:
            a, b = b, a % b
        return a

# Использование статических методов
print(MathUtils.is_prime(17))    # True
print(MathUtils.factorial(5))    # 120
```

```
print(MathUtils.gcd(48, 18))     # 6

# Статические методы можно вызывать и через экземпляр
utils = MathUtils()
print(utils.is_prime(13))        # True
```

4.2 Свойства (Properties)

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Геттер для температуры в Цельсиях"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Сеттер для температуры в Цельсиях"""
        if value < -273.15:
            raise ValueError("Температура не может быть ниже абсолютного нуля!")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Свойство: температура в Фаренгейтах"""
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Сеттер для температуры в Фаренгейтах"""
        self.celsius = (value - 32) * 5/9

    @property
    def kelvin(self):
        """Свойство: температура в Кельвинах"""
        return self._celsius + 273.15

# Использование свойств
temp = Temperature(25)
print(f"Цельсий: {temp.celsius}°C")      # 25°C
print(f"Фаренгейт: {temp.fahrenheit}°F") # 77.0°F
print(f"Кельвин: {temp.kelvin}K")        # 298.15K

# Изменение через свойство
temp.fahrenheit = 100
print(f"После изменения: {temp.celsius}°C")  # 37.77777777777778°C
```

---

5. Инкапсуляция

5.1 Основы инкапсуляции

Инкапсуляция — это сокрытие внутренней реализации объекта и предоставление контролируемого доступа к его состоянию.

5.2 Уровни доступа в Python

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand             # Публичный атрибут
        self._model = model            # Защищенный атрибут (условно приватный)
        self.__serial_number = "12345"  # Приватный атрибут (name mangling)

    def get_info(self):
        return f"{self.brand} {self._model}"

    def _internal_method(self):
        """Защищенный метод (условно приватный)"""
        return "Внутренний метод"

    def __private_method(self):
        """Приватный метод"""
        return "Приватный метод"

    def access_private(self):
        """Публичный метод для доступа к приватному"""
        return self.__private_method()

# Создание объекта
car = Car("Toyota", "Camry")

# Доступ к атрибутам
print(car.brand)          # Toyota (работает)
print(car._model)         # Camry (работает, но не рекомендуется)

# Попытка доступа к приватному атрибуту
try:
    print(car.__serial_number)
except AttributeError as e:
    print(f"Ошибка: {e}")  # 'Car' object has no attribute '__serial_number'

# Доступ через name mangling
print(car._Car__serial_number)  # 12345 (работает, но крайне не рекомендуется)

# Доступ к методам
print(car._internal_method())   # Работает, но не рекомендуется
print(car.access_private())     # Приватный метод (через публичный интерфейс)
```

5.3 Практический пример инкапсуляции

```
class BankAccount:
    """Пример правильной инкапсуляции"""

    def __init__(self, owner, initial_balance=0):
        self.owner = owner
        self.__balance = initial_balance  # Приватный атрибут
        self.__pin = None
        self.__is_locked = False

    def set_pin(self, pin):
        """Установка PIN-кода"""
        if len(str(pin)) == 4 and str(pin).isdigit():
            self.__pin = pin
            return True
        raise ValueError("PIN должен состоять из 4 цифр")
```

```
def verify_pin(self, pin):
    """Проверка PIN-кода"""
    return self.__pin == pin

@property
def balance(self):
    """Только чтение баланса"""
    if self.__is_locked:
        return "Счет заблокирован"
    return self.__balance

def deposit(self, amount, pin):
    """Пополнение счета"""
    if not self.verify_pin(pin):
        self.__lock_account()
        raise ValueError("Неверный PIN")

    if amount <= 0:
        raise ValueError("Сумма должна быть положительной")

    self.__balance += amount
    return f"Пополнено на {amount}. Баланс: {self.__balance}"

def withdraw(self, amount, pin):
    """Снятие денег"""
    if not self.verify_pin(pin):
        self.__lock_account()
        raise ValueError("Неверный PIN")

    if amount <= 0:
        raise ValueError("Сумма должна быть положительной")

    if amount > self.__balance:
        raise ValueError("Недостаточно средств")

    self.__balance -= amount
    return f"Снято {amount}. Баланс: {self.__balance}"

def __lock_account(self):
    """Приватный метод блокировки счета"""
    self.__is_locked = True
    print("ВНИМАНИЕ: Счет заблокирован из-за неверного PIN")

# Использование
account = BankAccount("Алиса", 1000)
account.set_pin(1234)

print(account.balance)                  # 1000
print(account.deposit(500, 1234))       # Пополнено на 500. Баланс: 1500

# Попытка прямого доступа к балансу не сработает
try:
    account.__balance = 999999  # Не изменит реальный баланс
    print(account.balance)      # По-прежнему правильный баланс
except:
    pass

# Неверный PIN заблокирует счет
try:
```

```
    account.withdraw(100, 5678)  # Неверный PIN
except ValueError as e:
    print(e)                     # Неверный PIN

print(account.balance)           # Счет заблокирован
```

---

6. Наследование

6.1 Основы наследования

Наследование позволяет создавать новые классы на основе существующих, наследуя их атрибуты и методы.

```
class Animal:
    """Базовый класс животного"""

    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.is_alive = True

    def eat(self):
        return f"{self.name} ест"

    def sleep(self):
        return f"{self.name} спит"

    def make_sound(self):
        return f"{self.name} издает звук"

    def info(self):
        status = "жив" if self.is_alive else "не жив"
        return f"{self.name} ({self.species}) - {status}"

class Dog(Animal):
    """Класс собаки, наследующий от Animal"""

    def __init__(self, name, breed):
        super().__init__(name, "Собака")  # Вызов конструктора родителя
        self.breed = breed
        self.tricks = []

    def make_sound(self):
        """Переопределение метода родителя"""
        return f"{self.name} гавкает: Гав-гав!"

    def learn_trick(self, trick):
        """Дополнительный метод для собаки"""
        self.tricks.append(trick)
        return f"{self.name} выучил трюк: {trick}"

    def perform_tricks(self):
        if not self.tricks:
            return f"{self.name} не знает трюков"
        return f"{self.name} выполняет: {', '.join(self.tricks)}"

class Cat(Animal):
    """Класс кота, наследующий от Animal"""
```

```
    def __init__(self, name, color):
        super().__init__(name, "Кот")
        self.color = color
        self.lives = 9

    def make_sound(self):
        return f"{self.name} мяукает: Мяу!"

    def purr(self):
        """Уникальный метод для кота"""
        return f"{self.name} мурлычет"

    def lose_life(self):
        if self.lives > 0:
            self.lives -= 1
            if self.lives == 0:
                self.is_alive = False
            return f"{self.name} потерял жизнь. Осталось: {self.lives}"
        return f"{self.name} уже мертв"

# Создание объектов
dog = Dog("Рекс", "Лабрадор")
cat = Cat("Мурзик", "Рыжий")

# Использование унаследованных методов
print(dog.info())       # Рекс (Собака) - жив
print(cat.info())       # Мурзик (Кот) - жив

# Использование переопределенных методов
print(dog.make_sound())  # Рекс гавкает: Гав-гав!
print(cat.make_sound())  # Мурзик мяукает: Мяу!

# Использование специфичных методов
print(dog.learn_trick("Сидеть"))       # Рекс выучил трюк: Сидеть
print(dog.learn_trick("Дать лапу"))    # Рекс выучил трюк: Дать лапу
print(dog.perform_tricks())            # Рекс выполняет: Сидеть, Дать лапу

print(cat.purr())       # Мурзик мурлычет
print(cat.lose_life())  # Мурзик потерял жизнь. Осталось: 8
```

6.2 Функция super()

`super()` предоставляет доступ к методам родительского класса.

```
class Vehicle:
    """Базовый класс транспортного средства"""

    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.fuel_level = 100

    def start_engine(self):
        return f"{self.brand} {self.model}: Двигатель запущен"

    def info(self):
        return f"{self.year} {self.brand} {self.model}"

class Car(Vehicle):
```

```
    """Класс автомобиля"""

    def __init__(self, brand, model, year, doors):
        super().__init__(brand, model, year)  # Вызов родительского конструктора
        self.doors = doors
        self.trunk_open = False

    def start_engine(self):
        """Расширение функциональности родительского метода"""
        base_result = super().start_engine()  # Вызов родительского метода
        return f"{base_result}. Автомобиль готов к поездке"

    def open_trunk(self):
        self.trunk_open = True
        return "Багажник открыт"

    def info(self):
        """Расширение информации"""
        base_info = super().info()  # Получаем базовую информацию
        return f"{base_info}, {self.doors} двери"

class ElectricCar(Car):
    """Класс электромобиля"""

    def __init__(self, brand, model, year, doors, battery_capacity):
        super().__init__(brand, model, year, doors)
        self.battery_capacity = battery_capacity
        self.charge_level = 100

    def start_engine(self):
        """Полное переопределение для электромобиля"""
        return f"{self.brand} {self.model}: Электромотор активирован"

    def charge_battery(self, amount):
        old_level = self.charge_level
        self.charge_level = min(100, self.charge_level + amount)
        return f"Заряжено с {old_level}% до {self.charge_level}%"

    def info(self):
        base_info = super().info()  # Получаем информацию от Car
        return f"{base_info}, Батарея: {self.battery_capacity}kWh"

# Демонстрация наследования
regular_car = Car("Toyota", "Camry", 2020, 4)
electric_car = ElectricCar("Tesla", "Model 3", 2021, 4, 75)

print(regular_car.start_engine())
# Toyota Camry: Двигатель запущен. Автомобиль готов к поездке

print(electric_car.start_engine())
# Tesla Model 3: Электромотор активирован

print(regular_car.info())
# 2020 Toyota Camry, 4 двери

print(electric_car.info())
# 2021 Tesla Model 3, 4 двери, Батарея: 75kWh
```

6.3 Множественное наследование

Python поддерживает множественное наследование с помощью Method Resolution Order (MRO).

```
class Flyer:
    """Миксин для летающих объектов"""

    def fly(self):
        return f"{self.__class__.__name__} летит"

    def land(self):
        return f"{self.__class__.__name__} приземляется"

class Swimmer:
    """Миксин для плавающих объектов"""

    def swim(self):
        return f"{self.__class__.__name__} плывет"

    def dive(self):
        return f"{self.__class__.__name__} ныряет"


class Bird(Animal, Flyer):
    """Птица: животное + может летать"""

    def __init__(self, name, wingspan):
        super().__init__(name, "Птица")
        self.wingspan = wingspan

    def make_sound(self):
        return f"{self.name} поет"

class Fish(Animal, Swimmer):
    """Рыба: животное + может плавать"""

    def __init__(self, name, depth_limit):
        super().__init__(name, "Рыба")
        self.depth_limit = depth_limit

    def make_sound(self):
        return f"{self.name} булькает"

class Duck(Animal, Flyer, Swimmer):
    """Утка: животное + может летать + может плавать"""

    def __init__(self, name):
        super().__init__(name, "Утка")

    def make_sound(self):
        return f"{self.name} крякает: Кря-кря!"

# Создание объектов
eagle = Bird("Орел", 2.0)
salmon = Fish("Лосось", 100)
duck = Duck("Кряква")

# Демонстрация возможностей
print(eagle.fly())          # Bird летит
print(salmon.swim())        # Fish плывет
print(duck.fly())           # Duck летит
print(duck.swim())          # Duck плывет
print(duck.make_sound())    # Кряква крякает: Кря-кря!
```

```
# Проверка MRO (Method Resolution Order)
print(Duck.__mro__)
# (<class '__main__.Duck'>, <class '__main__.Animal'>,
#  <class '__main__.Flyer'>, <class '__main__.Swimmer'>, <class 'object'>)
```

---

7. Полиморфизм

7.1 Основы полиморфизма

Полиморфизм позволяет объектам разных классов реагировать на одни и те же методы по-разному.

```
class Shape:
    """Базовый класс фигуры"""

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

    def area(self):
        # Формула Герона
        s = self.perimeter() / 2
        return (s * (s - self.side1) * (s - self.side2) * (s - self.side3)) ** 0.5

    def perimeter(self):
        return self.side1 + self.side2 + self.side3

# Демонстрация полиморфизма
def process_shapes(shapes):
    """Функция, демонстрирующая полиморфизм"""
    total_area = 0
    total_perimeter = 0

    print("Обработка фигур:")
    for shape in shapes:
        print(f"  {shape.info()}")
        total_area += shape.area()
        total_perimeter += shape.perimeter()

    print(f"\nОбщая площадь: {total_area:.2f}")
    print(f"Общий периметр: {total_perimeter:.2f}")

# Создание различных фигур
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Triangle(3, 4, 5),
    Rectangle(2, 8),
    Circle(2.5)
]

# Полиморфная обработка - все объекты имеют одинаковый интерфейс
process_shapes(shapes)
```

7.2 Duck Typing

Python использует "утиную типизацию": "Если что-то ходит как утка и крякает как утка, то это утка".

```
class FileWriter:
    """Класс для записи в файл"""
```

```
    def __init__(self, filename):
        self.filename = filename

    def write(self, data):
        with open(self.filename, 'w') as f:
            f.write(str(data))
        return f"Записано в файл {self.filename}"

class DatabaseWriter:
    """Класс для записи в базу данных"""

    def __init__(self, connection_string):
        self.connection_string = connection_string

    def write(self, data):
        # Имитация записи в БД
        return f"Записано в БД: {data}"

class ConsoleWriter:
    """Класс для вывода в консоль"""

    def write(self, data):
        print(f"КОНСОЛЬ: {data}")
        return "Выведено в консоль"

class Logger:
    """Логгер, использующий duck typing"""

    def __init__(self, writer):
        self.writer = writer  # Любой объект с методом write()

    def log(self, message):
        return self.writer.write(f"[LOG] {message}")

# Использование duck typing
file_logger = Logger(FileWriter("app.log"))
db_logger = Logger(DatabaseWriter("sqlite://db.sqlite"))
console_logger = Logger(ConsoleWriter())

# Все логгеры работают одинаково, несмотря на разные writer'ы
loggers = [file_logger, db_logger, console_logger]

for logger in loggers:
    result = logger.log("Система запущена")
    if "КОНСОЛЬ" not in result:
        print(result)
```

7.3 Оператор isinstance() и type()

```
class Vehicle:
    pass

class Car(Vehicle):
    pass

class ElectricCar(Car):
    pass
```

```
# Создание объектов
vehicle = Vehicle()
car = Car()
electric_car = ElectricCar()

# Проверка типов
print("=== Проверка с isinstance() ===")
print(f"isinstance(electric_car, ElectricCar): {isinstance(electric_car, ElectricCar)}")  # True
print(f"isinstance(electric_car, Car): {isinstance(electric_car, Car)}")            # True
print(f"isinstance(electric_car, Vehicle): {isinstance(electric_car, Vehicle)}")     # True

print(f"isinstance(car, ElectricCar): {isinstance(car, ElectricCar)}")            # False
print(f"isinstance(car, Car): {isinstance(car, Car)}")                            # True

print("\n=== Проверка с type() ===")
print(f"type(electric_car) == ElectricCar: {type(electric_car) == ElectricCar}")    # True
print(f"type(electric_car) == Car: {type(electric_car) == Car}")                    # False

# Полиморфная функция с проверкой типов
def service_vehicle(vehicle):
    """Полиморфная функция обслуживания"""
    if isinstance(vehicle, ElectricCar):
        return "Проверка батареи электромобиля"
    elif isinstance(vehicle, Car):
        return "Замена масла в двигателе"
    elif isinstance(vehicle, Vehicle):
        return "Общее техобслуживание"
    else:
        return "Неизвестный тип транспорта"

# Демонстрация
vehicles = [vehicle, car, electric_car]
for v in vehicles:
    print(f"{type(v).__name__}: {service_vehicle(v)}")
```

8. Абстракция

8.1 Абстрактные базовые классы (ABC)

```
from abc import ABC, abstractmethod

class Animal(ABC):
    """Абстрактный базовый класс животного"""

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @abstractmethod
    def make_sound(self):
        """Абстрактный метод - должен быть реализован в подклассах"""
        pass

    @abstractmethod
    def move(self):
        """Абстрактный метод движения"""
        pass
```

```
    def eat(self):
        """Конкретный метод - наследуется всеми подклассами"""
        return f"{self.name} ест"

    @abstractmethod
    def get_info(self):
        """Абстрактный метод для получения информации"""
        pass

class Dog(Animal):
    """Конкретная реализация собаки"""

    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def make_sound(self):
        return f"{self.name} гавкает: Гав-гав!"

    def move(self):
        return f"{self.name} бежит на четырех лапах"

    def get_info(self):
        return f"Собака {self.name}, {self.age} лет, порода: {self.breed}"

class Bird(Animal):
    """Конкретная реализация птицы"""

    def __init__(self, name, age, can_fly=True):
        super().__init__(name, age)
        self.can_fly = can_fly

    def make_sound(self):
        return f"{self.name} поет: Чирик-чирик!"

    def move(self):
        if self.can_fly:
            return f"{self.name} летит"
        return f"{self.name} прыгает"

    def get_info(self):
        fly_status = "умеет летать" if self.can_fly else "не умеет летать"
        return f"Птица {self.name}, {self.age} лет, {fly_status}"

# Нельзя создать экземпляр абстрактного класса
try:
    animal = Animal("Абстракт", 1)
except TypeError as e:
    print(f"Ошибка: {e}")

# Создание конкретных объектов
dog = Dog("Рекс", 3, "Лабрадор")
bird = Bird("Чиж", 1, True)

# Использование полиморфизма с абстракцией
animals = [dog, bird]

for animal in animals:
    print(animal.get_info())
    print(animal.make_sound())
```

```
    print(animal.move())
    print(animal.eat())
    print("-" * 30)
```

8.2 Абстрактные свойства и статические методы

```
from abc import ABC, abstractmethod

class Shape(ABC):
    """Абстрактный класс геометрической фигуры"""

    def __init__(self, color="black"):
        self._color = color

    @property
    @abstractmethod
    def area(self):
        """Абстрактное свойство площади"""
        pass

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, value):
        self._color = value

    @abstractmethod
    def draw(self):
        """Абстрактный метод отрисовки"""
        pass

    @staticmethod
    @abstractmethod
    def validate_dimensions(*args):
        """Абстрактный статический метод валидации"""
        pass

    def info(self):
        """Конкретный метод"""
        return f"{self.__class__.__name__}(цвет: {self.color}, площадь: {self.area:.2f})"

class Circle(Shape):
    """Конкретная реализация круга"""

    def __init__(self, radius, color="black"):
        self.validate_dimensions(radius)  # Валидация перед инициализацией
        super().__init__(color)
        self._radius = radius

    @property
    def area(self):
        import math
        return math.pi * self._radius ** 2

    @property
    def radius(self):
        return self._radius
```

```
    @radius.setter
    def radius(self, value):
        self.validate_dimensions(value)
        self._radius = value

    def draw(self):
        return f"Рисую круг радиусом {self._radius} цветом {self.color}"

    @staticmethod
    def validate_dimensions(radius):
        if radius <= 0:
            raise ValueError("Радиус должен быть положительным числом")

class Rectangle(Shape):
    """Конкретная реализация прямоугольника"""

    def __init__(self, width, height, color="black"):
        self.validate_dimensions(width, height)
        super().__init__(color)
        self._width = width
        self._height = height

    @property
    def area(self):
        return self._width * self._height

    def draw(self):
        return f"Рисую прямоугольник {self._width}x{self._height} цветом {self.color}"

    @staticmethod
    def validate_dimensions(width, height):
        if width <= 0 or height <= 0:
            raise ValueError("Ширина и высота должны быть положительными")

# Использование абстракции
try:
    # Валидация работает
    circle = Circle(5, "red")
    rectangle = Rectangle(3, 4, "blue")

    shapes = [circle, rectangle]

    for shape in shapes:
        print(shape.info())
        print(shape.draw())
        print()

    # Проверка валидации
    try:
        invalid_circle = Circle(-1)  # Ошибка валидации
    except ValueError as e:
        print(f"Ошибка валидации: {e}")

except Exception as e:
    print(f"Общая ошибка: {e}")
```

9. Специальные методы (магические методы)

9.1 Основные магические методы

```
class Book:
    """Класс книги с магическими методами"""

    def __init__(self, title, author, pages, price=0):
        self.title = title
        self.author = author
        self.pages = pages
        self.price = price

    def __str__(self):
        """Строковое представление для пользователя"""
        return f'"{self.title}" by {self.author}'

    def __repr__(self):
        """Строковое представление для разработчика"""
        return f'Book("{self.title}", "{self.author}", {self.pages}, {self.price})'

    def __len__(self):
        """Длина объекта (количество страниц)"""
        return self.pages

    def __eq__(self, other):
        """Сравнение на равенство"""
        if isinstance(other, Book):
            return (self.title == other.title and
                    self.author == other.author)
        return False

    def __lt__(self, other):
        """Сравнение "меньше чем" (по цене)"""
        if isinstance(other, Book):
            return self.price < other.price
        return NotImplemented

    def __add__(self, other):
        """Сложение книг (создание коллекции)"""
        if isinstance(other, Book):
            return BookCollection([self, other])
        return NotImplemented

    def __getitem__(self, page):
        """Доступ по индексу (имитация страниц)"""
        if isinstance(page, int):
            if 1 <= page <= self.pages:
                return f"Страница {page} книги '{self.title}'"
            raise IndexError("Номер страницы вне диапазона")
        return NotImplemented

    def __contains__(self, word):
        """Проверка наличия слова в названии"""
        return word.lower() in self.title.lower()

class BookCollection:
    """Коллекция книг"""

    def __init__(self, books=None):
        self.books = books or []
```

```
    def __len__(self):
        return len(self.books)

    def __iter__(self):
        """Делаем объект итерируемым"""
        return iter(self.books)

    def __getitem__(self, index):
        return self.books[index]

    def __str__(self):
        titles = [book.title for book in self.books]
        return f"Коллекция: {', '.join(titles)}"

    def add(self, book):
        if isinstance(book, Book):
            self.books.append(book)

# Демонстрация магических методов
book1 = Book("1984", "George Orwell", 328, 15.99)
book2 = Book("Brave New World", "Aldous Huxley", 268, 12.99)
book3 = Book("1984", "George Orwell", 328, 15.99)  # Дубликат

print("=== __str__ и __repr__ ===")
print(f"str(book1): {str(book1)}")      # "1984" by George Orwell
print(f"repr(book1): {repr(book1)}")    # Book("1984", "George Orwell", 328, 15.99)

print("\n=== __len__ ===")
print(f"len(book1): {len(book1)} страниц")  # 328 страниц

print("\n=== __eq__ и __lt__ ===")
print(f"book1 == book3: {book1 == book3}")  # True (одинаковые)
print(f"book1 == book2: {book1 == book2}")  # False (разные)
print(f"book2 < book1: {book2 < book1}")    # True (дешевле)

print("\n=== __add__ ===")
collection = book1 + book2
print(f"Коллекция: {collection}")  # Коллекция: 1984, Brave New World

print("\n=== __getitem__ ===")
print(book1[1])    # Страница 1 книги '1984'
print(book1[100])  # Страница 100 книги '1984'

print("\n=== __contains__ ===")
print(f"'1984' in book1: {'1984' in book1}")      # True
print(f"'Python' in book1: {'Python' in book1}")  # False

print("\n=== Итерация по коллекции ===")
collection.add(Book("Fahrenheit 451", "Ray Bradbury", 194, 13.50))
for book in collection:
    print(f"- {book}")
```

9.2 Контекстные менеджеры

```
class FileManager:
    """Контекстный менеджер для работы с файлами"""

    def __init__(self, filename, mode='r'):
        self.filename = filename
```

```
        self.mode = mode
        self.file = None

    def __enter__(self):
        """Вход в контекст - открытие файла"""
        print(f"Открываем файл {self.filename}")
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Выход из контекста - закрытие файла"""
        if self.file:
            print(f"Закрываем файл {self.filename}")
            self.file.close()

        if exc_type is not None:
            print(f"Произошла ошибка: {exc_val}")
            return False  # Не подавляем исключение
        return True


class DatabaseConnection:
    """Контекстный менеджер для соединения с БД"""

    def __init__(self, connection_string):
        self.connection_string = connection_string
        self.connected = False

    def __enter__(self):
        print(f"Подключение к БД: {self.connection_string}")
        self.connected = True
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.connected:
            print("Закрытие соединения с БД")
            self.connected = False

        if exc_type:
            print(f"Ошибка в БД операции: {exc_val}")
            # Откат транзакции
            self.rollback()

    def execute(self, query):
        if not self.connected:
            raise RuntimeError("Нет соединения с БД")
        return f"Выполнен запрос: {query}"

    def rollback(self):
        print("Откат транзакции")

# Использование контекстных менеджеров
print("=== Работа с файлом ===")
try:
    with FileManager("test.txt", "w") as f:
        f.write("Привет, мир!")
        # Файл автоматически закроется при выходе
except FileNotFoundError:
    print("Файл не найден")

print("\n=== Работа с БД ===")
```

```
with DatabaseConnection("sqlite://memory") as db:
    result1 = db.execute("SELECT * FROM users")
    print(result1)
    result2 = db.execute("UPDATE users SET active = 1")
    print(result2)
    # Соединение автоматически закроется
```

9.3 Дескрипторы

```
class ValidatedAttribute:
    """Дескриптор для валидации атрибутов"""

    def __init__(self, validator, name=None):
        self.validator = validator
        self.name = name

    def __set_name__(self, owner, name):
        """Автоматически вызывается при присвоении дескриптора атрибуту"""
        self.name = name
        self.private_name = f'_{name}'

    def __get__(self, obj, objtype=None):
        """Получение значения атрибута"""
        if obj is None:
            return self
        return getattr(obj, self.private_name, None)

    def __set__(self, obj, value):
        """Установка значения с валидацией"""
        if self.validator(value):
            setattr(obj, self.private_name, value)
        else:
            raise ValueError(f"Некорректное значение для {self.name}: {value}")

class Person:
    """Класс с валидируемыми атрибутами"""

    # Дескрипторы с различными валидаторами
    name = ValidatedAttribute(lambda x: isinstance(x, str) and len(x) > 0)
    age = ValidatedAttribute(lambda x: isinstance(x, int) and 0 <= x <= 150)
    email = ValidatedAttribute(lambda x: '@' in str(x))

    def __init__(self, name, age, email):
        self.name = name    # Используется дескриптор
        self.age = age      # Используется дескриптор
        self.email = email  # Используется дескриптор

    def __str__(self):
        return f"Person(name='{self.name}', age={self.age}, email='{self.email}')"

# Использование дескрипторов
try:
    person = Person("Алиса", 25, "alice@email.com")
    print(person)  # Person(name='Алиса', age=25, email='alice@email.com')

    # Валидация работает
    person.age = 30  # OK
    print(f"Новый возраст: {person.age}")
```

```
    # Попытка установить некорректное значение
    person.age = -5   # Ошибка!
except ValueError as e:
    print(f"Ошибка валидации: {e}")

try:
    person.email = "неправильный_email"  # Ошибка!
except ValueError as e:
    print(f"Ошибка валидации email: {e}")
```

---

10. Декораторы в классах

10.1 Property декораторы

```
import math

class Circle:
    """Класс круга с использованием property"""

    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """Геттер радиуса"""
        return self._radius

    @radius.setter
    def radius(self, value):
        """Сеттер радиуса с валидацией"""
        if value <= 0:
            raise ValueError("Радиус должен быть положительным")
        self._radius = value

    @property
    def diameter(self):
        """Вычисляемое свойство диаметра"""
        return self._radius * 2

    @property
    def area(self):
        """Вычисляемое свойство площади"""
        return math.pi * self._radius ** 2

    @property
    def circumference(self):
        """Вычисляемое свойство длины окружности"""
        return 2 * math.pi * self._radius

# Использование свойств
circle = Circle(5)
print(f"Радиус: {circle.radius}")          # 5
print(f"Диаметр: {circle.diameter}")       # 10
print(f"Площадь: {circle.area:.2f}")       # 78.54
print(f"Длина окружности: {circle.circumference:.2f}")  # 31.42

# Изменение радиуса
```

```
circle.radius = 3
print(f"Новая площадь: {circle.area:.2f}")  # 28.27
```

10.2 Классовые декораторы

```
def singleton(cls):
    """Декоратор для создания класса-синглтона"""
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

def log_methods(cls):
    """Декоратор для логирования вызовов методов"""
    for name, method in cls.__dict__.items():
        if callable(method) and not name.startswith('_'):
            setattr(cls, name, log_method_calls(method, name))
    return cls

def log_method_calls(method, method_name):
    """Декоратор для логирования отдельного метода"""
    def wrapper(*args, **kwargs):
        print(f"Вызов метода {method_name} с аргументами {args[1:]} {kwargs}")
        result = method(*args, **kwargs)
        print(f"Метод {method_name} завершен, результат: {result}")
        return result
    return wrapper

@singleton
class Database:
    """Класс-синглтон для подключения к базе данных"""

    def __init__(self, connection_string="sqlite://memory"):
        self.connection_string = connection_string
        self.connected = False
        print(f"Создание подключения к БД: {connection_string}")

    def connect(self):
        if not self.connected:
            self.connected = True
            return "Подключено к БД"
        return "Уже подключено"

    def disconnect(self):
        if self.connected:
            self.connected = False
            return "Отключено от БД"
        return "Уже отключено"

@log_methods
class Calculator:
    """Класс калькулятора с логированием методов"""

    def __init__(self):
        self.result = 0
```

```
    def add(self, value):
        self.result += value
        return self.result

    def multiply(self, value):
        self.result *= value
        return self.result

# Демонстрация синглтона
print("=== Тест синглтона ===")
db1 = Database()
db2 = Database()
print(f"db1 is db2: {db1 is db2}")  # True - один и тот же объект

# Демонстрация логирования
print("\n=== Тест логирования методов ===")
calc = Calculator()
calc.add(10)
calc.multiply(5)
```

10.3 Декораторы методов

```
import functools
import time
from typing import Any, Callable

def timer(func: Callable) -> Callable:
    """Декоратор для измерения времени выполнения метода"""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Метод {func.__name__} выполнен за {end_time - start_time:.4f} секунд")
        return result
    return wrapper

def cache_result(func: Callable) -> Callable:
    """Декоратор для кэширования результатов методов"""
    cache = {}

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # Создаем ключ для кэша
        cache_key = str(args) + str(sorted(kwargs.items()))

        if cache_key in cache:
            print(f"Результат {func.__name__} взят из кэша")
            return cache[cache_key]

        result = func(*args, **kwargs)
        cache[cache_key] = result
        print(f"Результат {func.__name__} сохранен в кэш")
        return result

    return wrapper

def validate_positive(func: Callable) -> Callable:
```

```
    """Декоратор для валидации положительных аргументов"""
    @functools.wraps(func)
    def wrapper(self, *args, **kwargs):
        for arg in args:
            if isinstance(arg, (int, float)) and arg <= 0:
                raise ValueError(f"Аргумент должен быть положительным: {arg}")
        return func(self, *args, **kwargs)
    return wrapper

class MathOperations:
    """Класс с декорированными методами"""

    @timer
    @cache_result
    def fibonacci(self, n):
        """Вычисление числа Фибоначчи (с кэшированием и таймером)"""
        if n <= 1:
            return n
        return self.fibonacci(n - 1) + self.fibonacci(n - 2)

    @validate_positive
    def factorial(self, n):
        """Факториал с валидацией"""
        if n <= 1:
            return 1
        return n * self.factorial(n - 1)

    @timer
    def slow_operation(self, seconds=1):
        """Имитация медленной операции"""
        time.sleep(seconds)
        return f"Операция завершена за {seconds} секунд"

# Использование декорированных методов
math_ops = MathOperations()

print("=== Fibonacci с кэшированием ===")
print(f"fibonacci(10): {math_ops.fibonacci(10)}")  # Первый вызов
print(f"fibonacci(10): {math_ops.fibonacci(10)}")  # Из кэша
print(f"fibonacci(12): {math_ops.fibonacci(12)}")  # Частично из кэша

print("\n=== Валидация аргументов ===")
try:
    print(f"factorial(5): {math_ops.factorial(5)}")   # OK
    print(f"factorial(-3): {math_ops.factorial(-3)}") # Ошибка
except ValueError as e:
    print(f"Ошибка: {e}")

print("\n=== Измерение времени ===")
math_ops.slow_operation(0.5)
```