

Практика 11.1 - Наследование, инкапсуляция, полиморфизм

Помимо классов и объектов, ООП содержит дополнительные три основные концепции: наследование, инкапсуляция и полиморфизм.

Наследование

Наследование является одним из ключевых понятий ООП. За счёт наследования можно создать один общий класс (класс родитель) и создать множество других классов (классы наследники), что будут наследовать все поля, методы и конструкторы из главного класса.

Зачем использовать наследование?

Предположим что у нас есть один большой класс «Транспорт». В классе описываются базовые характеристики для всех транспортных средств:

- поля: скорость, вес, запас хода и тому подобное;
- методы: получение информации из полей, установка новых значений;
- конструктор: пустой и по установке всех полей.

На основе класса мы спокойно можем создать объект легковой машины, объект грузовика, объект самолета и так далее. У всех объектов будут одинаковые характеристики и методы.

Мы явно понимаем, что у объекта машина и самолёт будут разные поля и характеристики. Как можно поступить:

Можно создать два отдельных класса: «Car» и «Airplane». В каждом классе будут все методы, поля и конструкторы повторно переписанные из класса «Транспорт», а также будут новые методы, что важны только для конкретного класса;

Можно создать два класса наследника: «Car» и «Airplane». Оба класса будут наследовать всё от класса «Транспорт» и при этом будут содержать свои дополнительные функции. Таким образом повторения кода не будет и код станет меньше и чище.

Создание классов наследников

Для создания класса наследника требуется создать класс и указать наследование от главного класса.

Пример класса наследника:

```
Приветствие classCars.py X Welcome to Settings Sync
Users > arailymtleubayeva > classCars.py > ...
1 class Cars:
2     wheels = 4 # Общее значение для всех объектов,
3     # так как все машины имеют колеса
4
5 class BMW (Cars):
6     is_m_series = True # Является ли модель "М" серии?
7     # Переменная нужна только в классе BMW
```

ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ **ТЕРМИНАЛ** ПОРТЫ КОММЕНТАРИИ + Python

```
/usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
● arailymtleubayeva@MacBook-Air-Arilym ~ % /usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
○ arailymtleubayeva@MacBook-Air-Arilym ~ %
```

Наследование позволяет одному классу (наследнику) наследовать свойства и методы другого класса (родителя). Это помогает избежать дублирования кода и упрощает расширение функциональности.

Пример класса наследника:

```
← → Поиск
Приветствие classCars.py X Welcome to Settings Sync
Users > arailymtleubayeva > classCars.py > ...
1 # Определение основного класса Transport
2 class Transport:
3     def __init__(self, speed, weight, range):
4         self.speed = speed # Скорость транспортного средства
5         self.weight = weight # Вес транспортного средства
6         self.range = range # Дальность поездки транспортного средства
7
8 # Класс Car, наследующий от Transport
9 class Car(Transport):
10     def __init__(self, speed, weight, range, wheels):
11         super().__init__(speed, weight, range) # Инициализация атрибутов родительского класса
12         self.wheels = wheels # Количество колёс автомобиля
13
14 # Класс Airplane, наследующий от Transport
15 class Airplane(Transport):
16     def __init__(self, speed, weight, range, wingspan):
17         super().__init__(speed, weight, range) # Инициализация атрибутов родительского класса
18         self.wingspan = wingspan # Размах крыльев самолёта
19
```

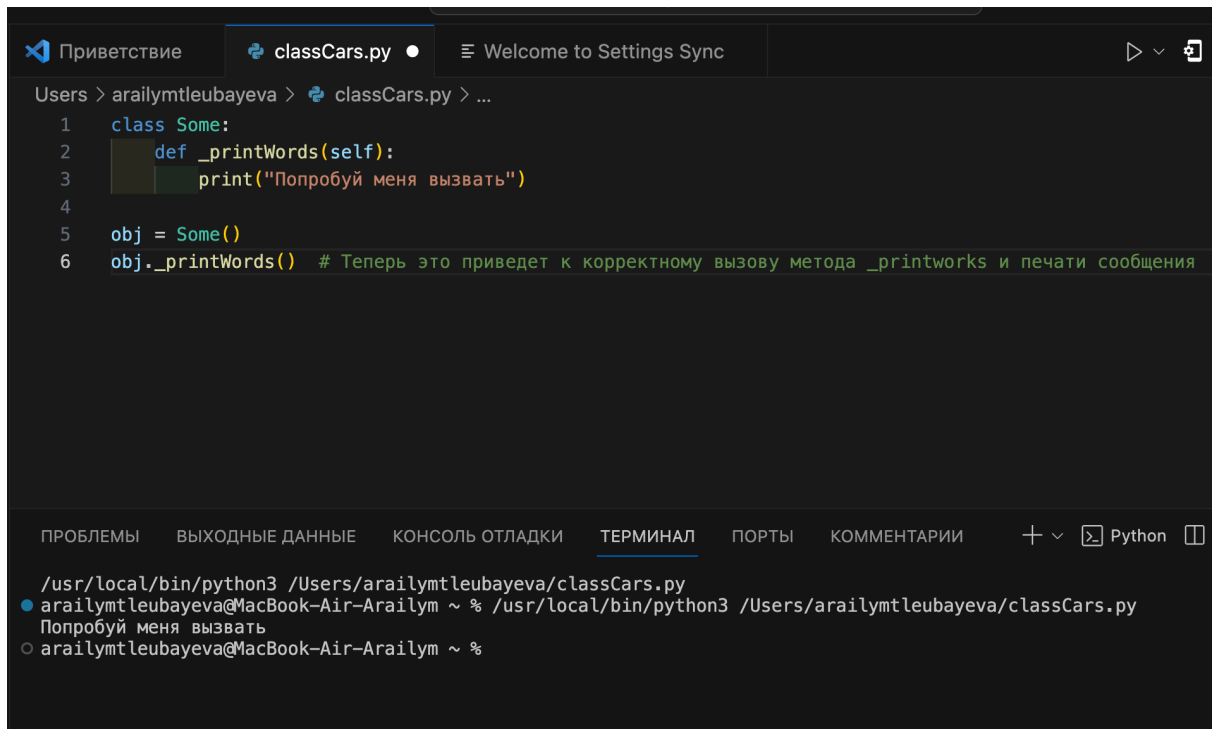
ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ **ТЕРМИНАЛ** ПОРТЫ КОММЕНТАРИИ

```
/usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
● arailymtleubayeva@MacBook-Air-Arilym ~ % /usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
○ arailymtleubayeva@MacBook-Air-Arilym ~ %
```

Инкапсуляция

Инкапсуляция позволяет ограничить доступ к какой-либо функции в классе. Благодаря такому подходу злоумышленники или же мы сами не сможем случайно или намеренно вызвать или изменить метод.

Пример:



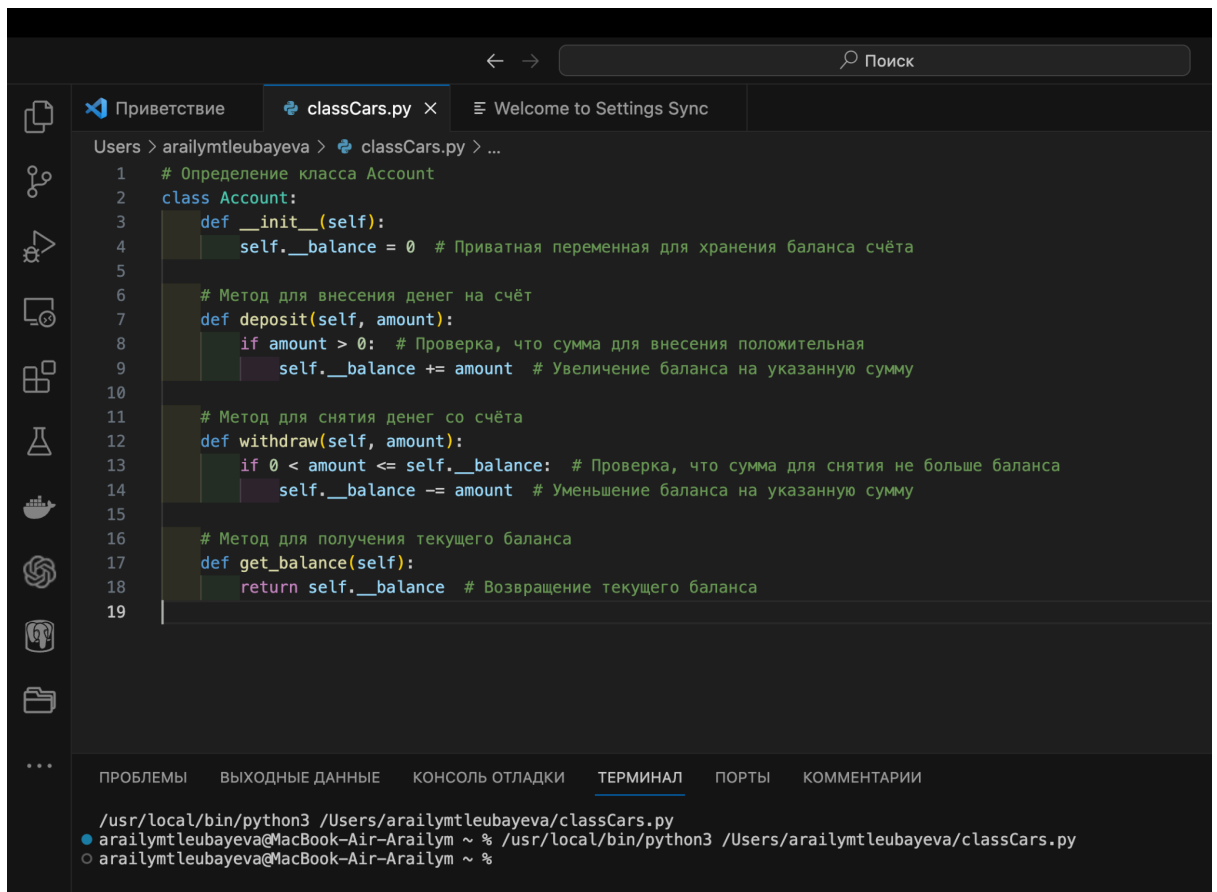
```
classCars.py
class Some:
    def _printWords(self):
        print("Попробуй меня вызвать")

obj = Some()
obj._printWords() # Теперь это приведет к корректному вызову метода _printworks и печати сообщения
```

ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ ПОРТЫ КОММЕНТАРИИ + ∨ Python

```
/usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
● arailymtleubayeva@MacBook-Air-Arilym ~ % /usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
  Попробуй меня вызвать
○ arailymtleubayeva@MacBook-Air-Arilym ~ %
```

При помощи двойного подчеркивания эффект защиты усиливается, поэтому вызвать функцию будет невозможным.



```
1 # Определение класса Account
2 class Account:
3     def __init__(self):
4         self.__balance = 0 # Приватная переменная для хранения баланса счёта
5
6     # Метод для внесения денег на счёт
7     def deposit(self, amount):
8         if amount > 0: # Проверка, что сумма для внесения положительная
9             self.__balance += amount # Увеличение баланса на указанную сумму
10
11    # Метод для снятия денег со счёта
12    def withdraw(self, amount):
13        if 0 < amount <= self.__balance: # Проверка, что сумма для снятия не больше баланса
14            self.__balance -= amount # Уменьшение баланса на указанную сумму
15
16    # Метод для получения текущего баланса
17    def get_balance(self):
18        return self.__balance # Возвращение текущего баланса
19
```

В этом классе:

`__balance` - это приватная переменная, которая используется для хранения баланса счёта. Она доступна только внутри класса `Account`, что предотвращает прямой доступ к ней извне и защищает от нежелательных изменений.

Метод `deposit` позволяет вносить деньги на счёт. Он проверяет, что вносимая сумма положительна, и только тогда увеличивает баланс.

Метод `withdraw` позволяет снимать деньги со счёта. Он проверяет, что запрашиваемая сумма находится в пределах доступного баланса, прежде чем уменьшить баланс.

Метод `get_balance` предоставляет доступ к текущему балансу, позволяя его только читать, но не изменять напрямую.

Этот подход к инкапсуляции обеспечивает безопасность данных и контроль над тем, как эти данные могут быть изменены или доступны.

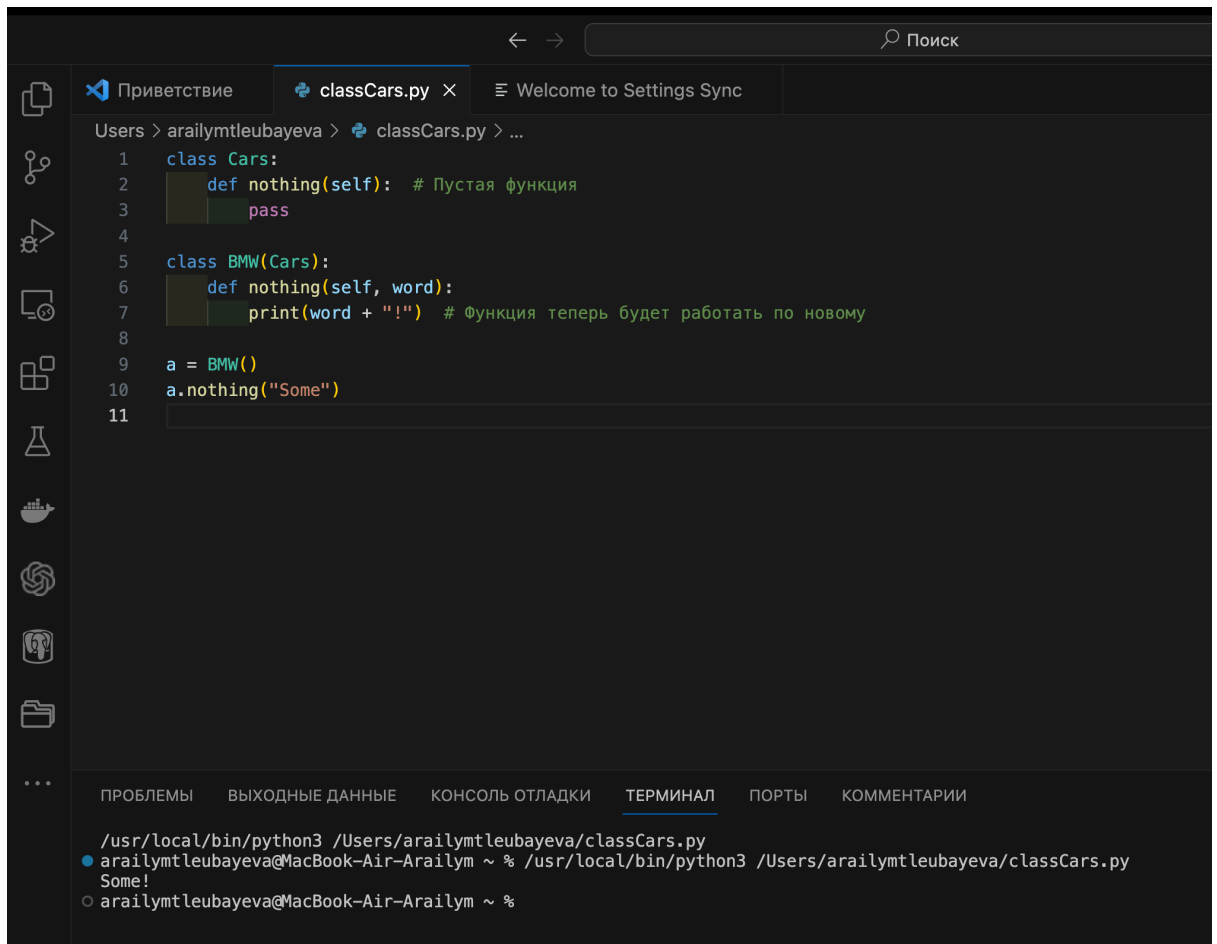
Итого, Инкапсуляция заключается в ограничении прямого доступа к некоторым компонентам объекта и предоставлении методов для безопасного взаимодействия с ними.

Полиморфизм

Полиморфизм

Полиморфизм — это способность методов обрабатывать данные разных типов. В ООП это часто реализуется через переопределение методов в классах-наследниках.

Полиморфизм позволяет изменять функции в классах наследниках. Пример:



The screenshot shows a code editor with a dark theme. The top bar includes a search icon and the text 'Поиск'. Below the top bar, there are tabs for 'Приветствие', 'classCars.py', and 'Welcome to Settings Sync'. The main editor area displays the following Python code:

```
1 class Cars:
2     def nothing(self): # Пустая функция
3         pass
4
5 class BMW(Cars):
6     def nothing(self, word):
7         print(word + "!") # Функция теперь будет работать по новому
8
9 a = BMW()
10 a.nothing("Some")
11
```

At the bottom of the editor, there is a terminal window with the following output:

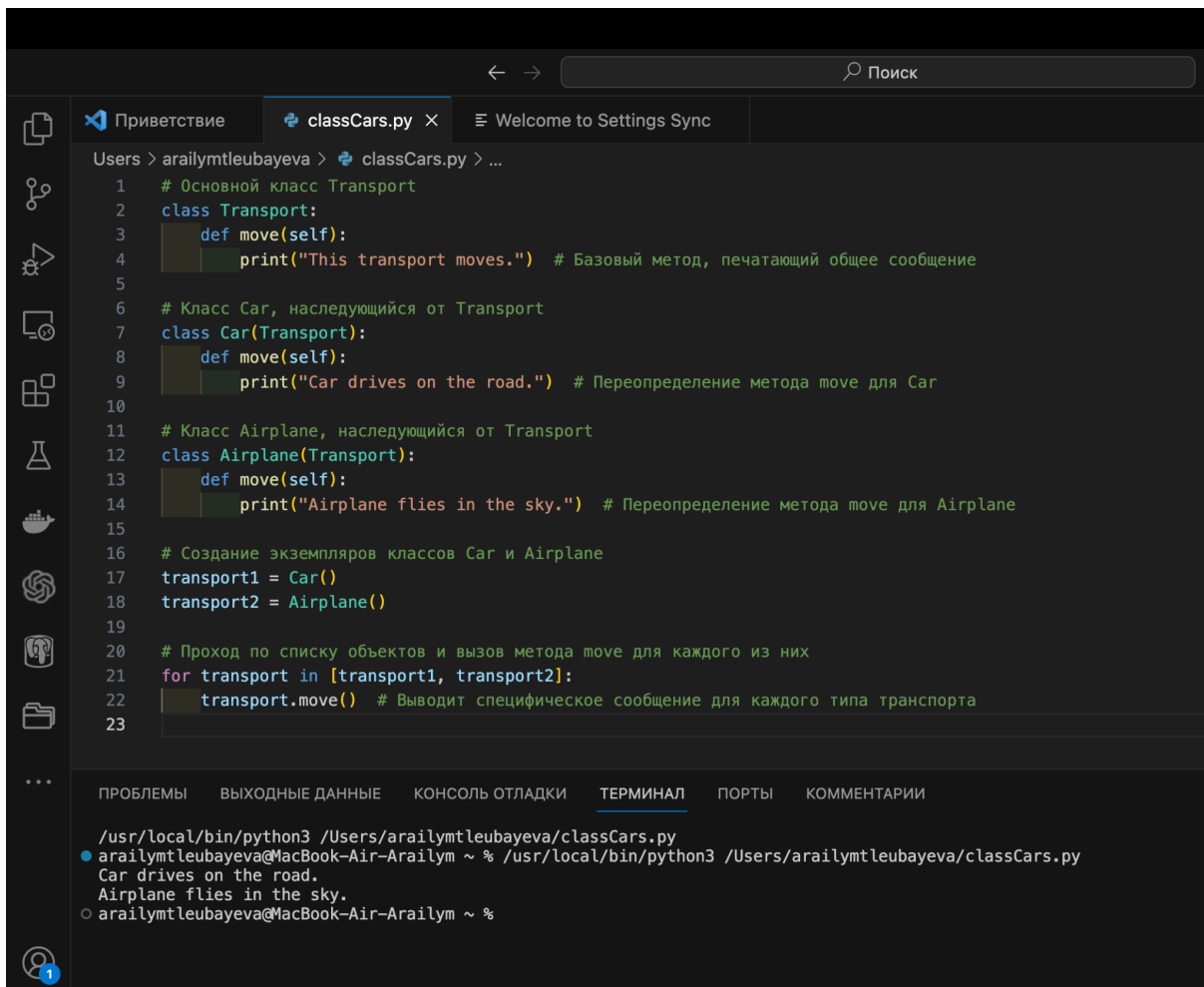
```
/usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
arailymtleubayeva@MacBook-Air-Arilym ~ % /usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
Some!
```

В этом примере:

Класс Cars содержит метод nothing, который ничего не делает (оператор pass).

Класс BMW, который является наследником класса Cars, переопределяет метод nothing. В этой версии метод принимает аргумент word и выводит его с восклицательным знаком на конце.

Это демонстрация полиморфизма: метод nothing в классе BMW имеет ту же самую название, что и в родительском классе Cars, но выполняет другую функцию.



```
1 # Основной класс Transport
2 class Transport:
3     def move(self):
4         print("This transport moves.") # Базовый метод, печатающий общее сообщение
5
6 # Класс Car, наследующийся от Transport
7 class Car(Transport):
8     def move(self):
9         print("Car drives on the road.") # Переопределение метода move для Car
10
11 # Класс Airplane, наследующийся от Transport
12 class Airplane(Transport):
13     def move(self):
14         print("Airplane flies in the sky.") # Переопределение метода move для Airplane
15
16 # Создание экземпляров классов Car и Airplane
17 transport1 = Car()
18 transport2 = Airplane()
19
20 # Проход по списку объектов и вызов метода move для каждого из них
21 for transport in [transport1, transport2]:
22     transport.move() # Выводит специфическое сообщение для каждого типа транспорта
23
```

PROBLEMY ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ **ТЕРМИНАЛ** ПОРТЫ КОММЕНТАРИИ

```
/usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
arailymtleubayeva@MacBook-Air-Arilym ~ % /usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
Car drives on the road.
Airplane flies in the sky.
arailymtleubayeva@MacBook-Air-Arilym ~ %
```

В этом примере:

Класс `Transport` определяет метод `move`, который печатает общее сообщение.

Классы `Car` и `Airplane` наследуются от `Transport` и переопределяют метод `move`, предоставляя свою уникальную реализацию. Это показывает, как один и тот же метод (`move`) может выполнять разные функции в зависимости от класса, к которому он принадлежит.

В конце кода создаются экземпляры классов `Car` и `Airplane`, и через цикл вызывается метод `move` для каждого из них. Здесь проявляется полиморфизм: одинаковый интерфейс (`move`) для разных типов объектов (`Car` и `Airplane`) приводит к различному поведению.

Таким образом, полиморфизм позволяет объектам разных классов отвечать на одинаковые вызовы методов разными способами, что упрощает управление кодом и повышает его гибкость.

Каждая из этих концепций играет ключевую роль в создании гибких, масштабируемых и поддерживаемых программных систем.

Аудиторное задание

Простое наследование
Есть класс Автомобиль:

```
class Car: # Создание класса

    wheels = 4 # Несколько полей

    model = "Some"

    speed = 123.5

    def set(self, wheels, model, speed): # Метод для установки значений

        self.wheels = wheels

        self.model = model

        self.speed = speed

    def getAll (self): # Метод для вывода значений

        print ("Автомобиль ", self.model, " может ехать со скоростью ", self.speed,
" на всех ", self.wheels, " колесах!")

        pass
```

Создайте класс-наследник Мотоцикл с параметрами:

- поле с описанием двигателя - Engine;
- метод для установки значения в поле Engine.
-

Создание объектов:

- Создайте два объекта для класс Автомобиль и используйте оба метода для каждого из объектов.
- Создайте объект на основе класса-наследника и используйте методы из класса Мотоцикл, а также из класса Автомобиль.

Декораторы функций

Декораторы в языке Питон позволяют добавить функционал до и после выполнения определенной функции.

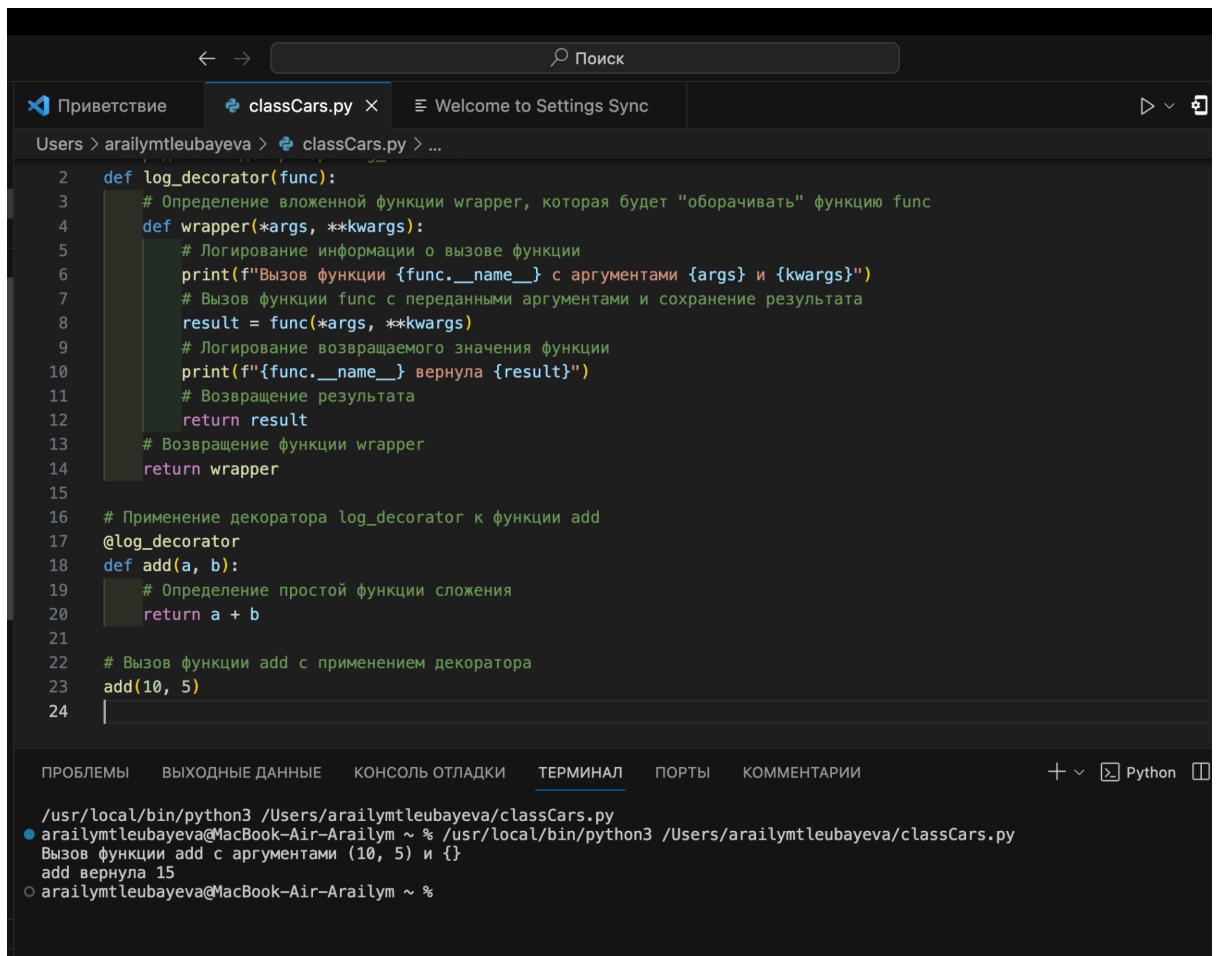
Декораторы бывают очень удобны во многих случаях. Вы можете задать вопросы, а зачем оборачивать какую-либо функцию с добавлением нового кода до и после её выполнения, если можно просто вписать это же в саму функцию?

Давайте на секунду представим что у нас большая программа и мы используем какую-либо функцию множество раз. Нам не нужно её менять, так как её использует много других подпрограмм. Тем не менее, нам нужно добавить к ней код, который должен сработать до или после её вызова. Тут на помощь и приходят декораторы, которые способны выполнить эту задачу.

Декораторы — это мощный инструмент, который позволяет модифицировать поведение функций или методов без изменения их кода. Это особенно полезно в больших и сложных системах, где изменение существующей функции может повлиять на множество других частей программы. Вот несколько примеров сценариев, в которых декораторы могут быть особенно полезны:

- **Логирование:** Добавление логирования к функциям для отслеживания их вызовов и параметров, что может быть полезно для отладки и мониторинга.
- **Кэширование:** Автоматическое сохранение результатов дорогостоящих вычислений для ускорения повторных вызовов с теми же аргументами.
- **Проверка авторизации:** Проверка прав пользователя перед выполнением определенных действий в веб-приложениях.
- **Валидация входных данных:** Проверка корректности входных данных перед выполнением функции.
- **Измерение времени выполнения:** Использование декораторов для замера времени выполнения функций, что может помочь в оптимизации производительности.
- **Обработка исключений:** Обеспечение стандартного способа обработки ошибок и исключений в функциях.

Вот пример простого декоратора, который логирует вызовы функции:



```
2 def log_decorator(func):
3     # Определение вложенной функции wrapper, которая будет "оборачивать" функцию func
4     def wrapper(*args, **kwargs):
5         # Логирование информации о вызове функции
6         print(f"Вызов функции {func.__name__} с аргументами {args} и {kwargs}")
7         # Вызов функции func с переданными аргументами и сохранение результата
8         result = func(*args, **kwargs)
9         # Логирование возвращаемого значения функции
10        print(f"{func.__name__} вернула {result}")
11        # Возвращение результата
12        return result
13    # Возвращение функции wrapper
14    return wrapper
15
16 # Применение декоратора log_decorator к функции add
17 @log_decorator
18 def add(a, b):
19     # Определение простой функции сложения
20     return a + b
21
22 # Вызов функции add с применением декоратора
23 add(10, 5)
24
```

PROБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ **ТЕРМИНАЛ** ПОРТЫ КОММЕНТАРИИ + - Python

```
/usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
● arailymtleubayeva@MacBook-Air-Arilym ~ % /usr/local/bin/python3 /Users/arailymtleubayeva/classCars.py
Вызов функции add с аргументами (10, 5) и {}
add вернула 15
○ arailymtleubayeva@MacBook-Air-Arilym ~ %
```

Как это работает:

- Декоратор `log_decorator` принимает функцию `func` и возвращает новую функцию `wrapper`.
- `wrapper` принимает любые аргументы и ключевые слова (`*args` и `**kwargs`), логирует их, затем вызывает `func` с этими аргументами и логирует результат.
- При использовании `@log_decorator` перед определением функции `add`, вызов `add(10, 5)` фактически становится вызовом `wrapper(10, 5)`.
- `wrapper` логирует вызов и результат функции `add`, а затем возвращает результат, который в данном случае является суммой 10 и 5.

Этот пример демонстрирует, как декораторы могут быть использованы для добавления дополнительного функционала (в данном случае, логирования) к существующим функциям без изменения их первоначального кода.

Самостоятельное задание

Простые задачи

1) Класс для управления банковским счетом:

- Создание класса `BankAccount` с базовыми операциями: внесение и снятие средств, проверка баланса.

2) Симуляция зоопарка:

- Разработка класса Animal и нескольких производных классов (например, Lion, Tiger, Elephant) с основными методами, такими как speak.

3) Декоратор функций

Создайте декоратор, который обернет любую функцию и будет выполнять:

- вывод строки до выполнения функции: «Код до выполнения функции»;
- выполнение переданной функции;
- вывод строки после выполнения функции: «Код после выполнения функции».

Средние задачи

1) Иерархия транспортных средств:

Разработка иерархии классов для различных видов транспортных средств с общими атрибутами в базовом классе и уникальными в подклассах.

2) Учетная система для сотрудников:

Создание класса Employee и его наследников для разных ролей в компании, таких как Manager, Engineer, SalesPerson.

3) Графический интерфейс пользователя:

Использование библиотеки для графического интерфейса для создания базового приложения с интерактивными элементами.

Сложные задачи

1) Система управления библиотекой:

Создание классов Book, Author, Library с возможностями управления книгами, поиска и предоставления информации.

2) Игра "Крестики-нолики":

Разработка игры "Крестики-нолики" с полным набором правил, логикой игры и интерфейсом для двух игроков.