IT CAREER HUB

Python

Инкапсуляция

Преподаватель

Портрет

Имя Фамилия

Текущая должность

Количество лет опыта

Какой у Вас опыт - ключевые кейсы

Самые яркие проекты

Дополнительная информация по вашему усмотрению

Корпоративный e-mail

Социальные сети (по желанию)

Важно

- Камера должна быть включена на протяжении всего занятия
- В течение занятия вопросы задавать в чате или когда преподаватель спрашивает, есть ли у Вас вопросы
- Вести себя уважительно и этично по отношению к остальным участникам занятия
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя

Повторение

- [x] Множественное наследование
- [x] Функция hasattr
- [x] Миксины
- [x] Порядок поиска методов при наследовании
- [x] Порядок разрешения методов (MRO)
- [x] Функция super в множественном наследовании
- [x] Композиция и агрегация

План занятия

- Инкапсуляция
- Уровни доступа
- Защищённые и приватные методы
- Посмотреть ответ
- Геттеры и сеттеры
- Декоратор @property

ICH IT CAREER HUB

ОСНОВНОЙ БЛОК

IT CAREER HUB

Инкапсуляция

Полезно знать

ICH
IT CAREER HUB

Инкапсуляция

Это подход, при котором внутреннее устройство объекта скрывается, а взаимодействие с ним осуществляется через понятный и контролируемый интерфейс.

Инкапсуляция помогает

- Защитить внутренние данные от прямого доступа и случайных изменений
- Контролировать поведение объекта через ограниченные точки входа
- Упростить использование объектов без знания их внутренней структуры

Уровни доступа
к атрибутам в Python

Публичные

Защищённые

Приватные

Python предлагает принятые соглашения по именованию, которые помогают соблюдать инкапсуляцию

Полезно знать

ICH
IT CAREER HUB

Публичные атрибуты

Это атрибуты, к которым можно свободно обращаться из любого места: и изнутри класса, и снаружи — через объект. Они не имеют специального префикса и являются полностью открытыми

Особенности публичных атрибутов

- Доступны для чтения и записи извне
- Используются как часть открытого интерфейса объекта
- Не скрывают детали реализации

Публичные атрибуты

Пример

```
class Book:
    def __init__(self, title, author):
        self.title = title      # публичное поле
        self.author = author    # публичное поле


book = Book("1984", "George Orwell")
print(book.title)           # доступ к публичному
полю
book.title = "Animal Farm"  # изменение публичного
поля
print(book.title)
```

Пояснение

- Если какое-то значение должно быть видно и доступно извне, его делают публичным

Полезно знать

ICH
IT CAREER HUB

Защищённые атрибуты

Такие атрибуты предназначены для внутреннего использования в классе и его наследниках, но всё ещё могут быть доступны извне.
В Python защищённые атрибуты обозначаются одним подчёркиванием перед именем: **`_name`**

Особенности защищенных атрибутов

- Условно считаются не для внешнего использования
- Не мешают доступу извне, но сигнализируют, что это внутренняя часть реализации
- Могут использоваться в дочерних классах

Защищённые атрибуты

Пример

```
class Book:
    def __init__(self, title):
        self._title = title  # защищённый атрибут

    def show_title(self):
        print(self._title)

class SpecialBook(Book):
    def show_title(self):
        print(self._title.upper())  # доступ из
наследника

book = SpecialBook("Brave New World")
print(book._title)        # доступ извне
технически возможен
book.show_title()
```

Пояснение

- Python не запрещает использовать `_title` напрямую, но появляется "warning" в виде подчёркивания

Полезно знать

ICH
IT CAREER HUB

Приватные атрибуты

Это атрибуты, которые предназначены только для использования внутри класса. Python не полностью запрещает доступ к приватным атрибутам, но автоматически меняет их имя внутри класса, чтобы затруднить случайное использование извне.
Приватные атрибуты обозначаются двумя подчёркиваниями перед именем: **`__name`**

Особенности приватных атрибутов

- Нельзя обратиться напрямую извне под тем же именем
- Используются для скрытия реализации и защиты важных данных
- Всё ещё технически доступны, но под изменённым именем

Пример: приватные атрибуты

```
class Book:
    def __init__(self, title):
        self.__title = title  # приватный атрибут

    def show_title(self):
        print(self.__title)

class SpecialBook(Book):
    def print_title(self):
        print(self.__title.upper())  # ошибка

book = SpecialBook("Brave New World")
book.show_title()     # доступ через метод базового класса
# book.shout_title()  # ошибка: такого атрибута с таким именем нет
# book.__title        # ошибка: такого атрибута с таким именем нет
```

Полезно знать

ICH
IT CAREER HUB

Манглирование имён
(от англ. name mangling)

Это механизм, с помощью которого Python автоматически изменяет имя приватных атрибутов, чтобы они не конфликтовали с атрибутами в дочерних классах и не были случайно переопределены или использованы извне

Важно

Если атрибут начинается с **двух подчёркиваний** и **не заканчивается** подчёркиваниями: `__value`, Python преобразует имя этого атрибута внутри класса в `_ClassName__value`

Манглирование имён

Пример

```
class Book:
    def __init__(self, title):
        self.__title = title  # приватный
атрибут


book = Book("Brave New World")
# print(book.__title)     #
AttributeError
print(book._Book__title)  # доступ через
mangled-имя
```

Пояснение

- Обращаться к атрибутам через `_ClassName__attr` **технически возможно**, но **делать так не рекомендуется** — это нарушает принцип инкапсуляции.

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Защищённые и приватные методы

Уровни доступа
к методам в Python

Публичные

Защищённые

Приватные

Главная цель использования защищённых и приватных методов — скрыть детали реализации, которые не предназначены для вызова снаружи

Защищённые методы

Защищённые методы используются, когда нужно **вынести часть логики в отдельный метод**, который **не должен вызываться напрямую извне**, но может быть **расширен в наследниках**

Пример

```
class Report:
    def __init__(self, data):
        self.data = data

    def generate(self):
        cleaned = self._prepare_data()
        print("Отчёт:")
        print("\n".join(cleaned))

    def _prepare_data(self):
        # Фильтрация пустых строк и обрезка
пробелов
        return [line.strip() for line in
self.data if line.strip()]
```

1

```
class SalesReport(Report):
    def _prepare_data(self):
        raw = super()._prepare_data()
        return [line for line in raw if not
line.startswith("#")]  # удаляем строки-
комментарии

data = [
    "  Продажи за январь  ",
    "",
    " # внутренний комментарий ",
    "  Доход: $5000 "
]

report = SalesReport(data)
report.generate()
```

2

Особенности защищённых методов

- `_prepare_data()` используется только внутри класса
- Его можно расширить в дочерних классах
- Но внешнему коду он не нужен — пользователь работает только с generate()

Приватные методы

Приватные методы применяются, когда часть логики должна быть **строго скрыта** и не должна вызываться извне или переопределяться в наследниках

Пример

```python
class User:
    def __init__(self, name):
        self.name = name
        self.__id = self.__generate_id()

    def show_info(self):
        print(f"{self.name} - ID: {self.__id}")

    def __generate_id(self):
        # приватная логика генерации идентификатора
        from random import randint
        return f"user-{randint(1000, 9999)}"

user = User("Alice")
user.show_info()
# print(user.__generate_id())       # AttributeError
print(user._User__generate_id())    # доступ возможен, но нарушает инкапсуляцию
```

Особенности приватных методов

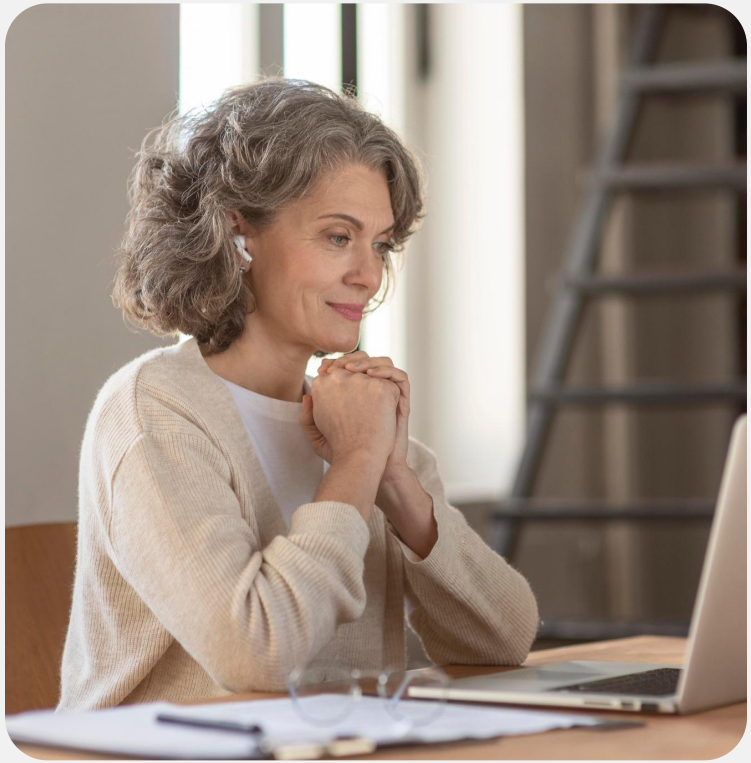
- `__generate_id()` — это **вспомогательная логика**, которая не нужна внешнему коду
- Метод **не может быть переопределён** в наследниках
- Пользователь взаимодействует только через `show_info()`, не зная деталей

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

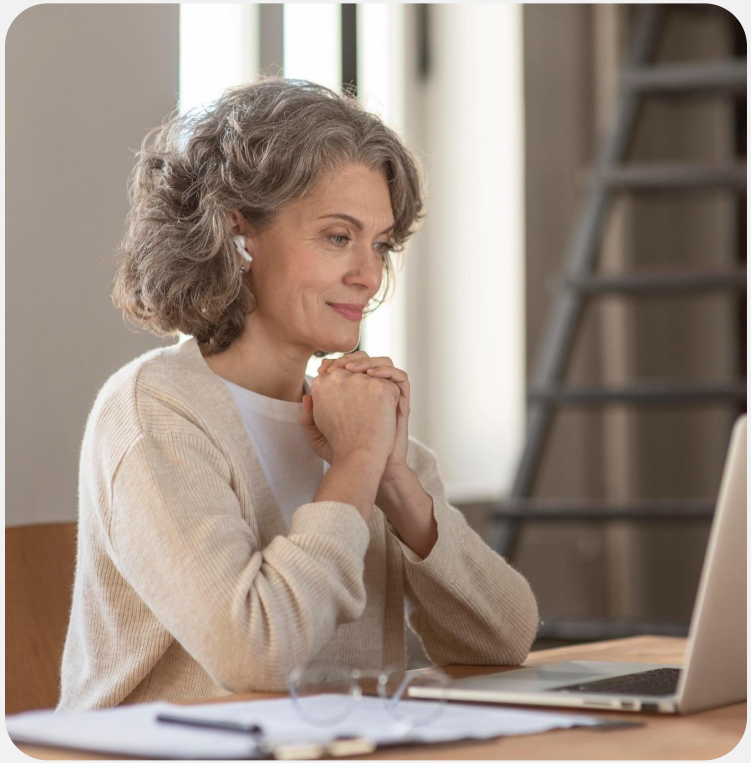
ЗАДАНИЯ



Выполните задание

Найдите ошибку в коде:

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

    def show_title(self):
        print(__title)

book = Book("1984")
book.show_title()
```

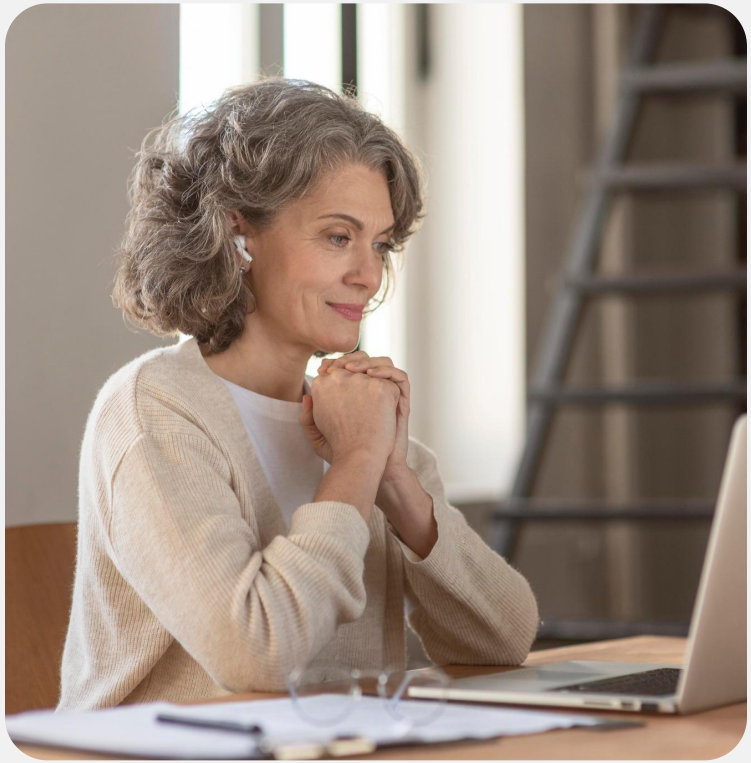


Выполните задание

Найдите ошибку в коде:

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

    def show_title(self):
        print(__title)

book = Book("1984")
book.show_title()
```

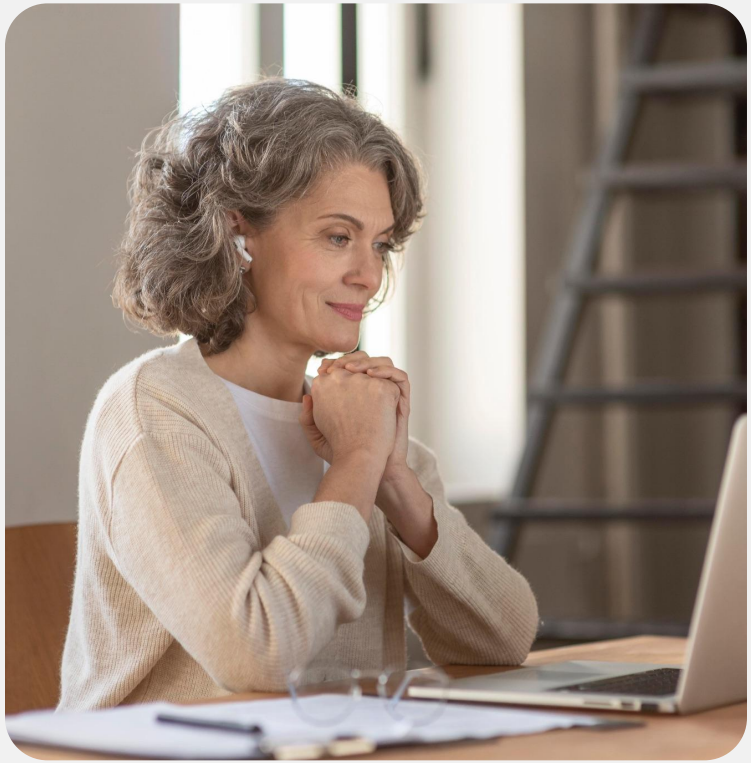
Ответ: в методе show_title нужно обращаться с помощью self.__title



Выберите верные варианты ответа

Укажите верные утверждения об уровнях доступа в Python:

a. Публичные атрибуты доступны везде
b. Защищённые атрибуты запрещено использовать вне класса
c. Приватные атрибуты создаются с одним подчёркиванием
d. Защищённые атрибуты могут использоваться в дочерних классах



Выберите верные варианты ответа

Укажите верные утверждения об уровнях доступа в Python:

a. Публичные атрибуты доступны везде
b. Защищённые атрибуты запрещено использовать вне класса
c. Приватные атрибуты создаются с одним подчёркиванием
d. Защищённые атрибуты могут использоваться в дочерних классах

IT CAREER HUB

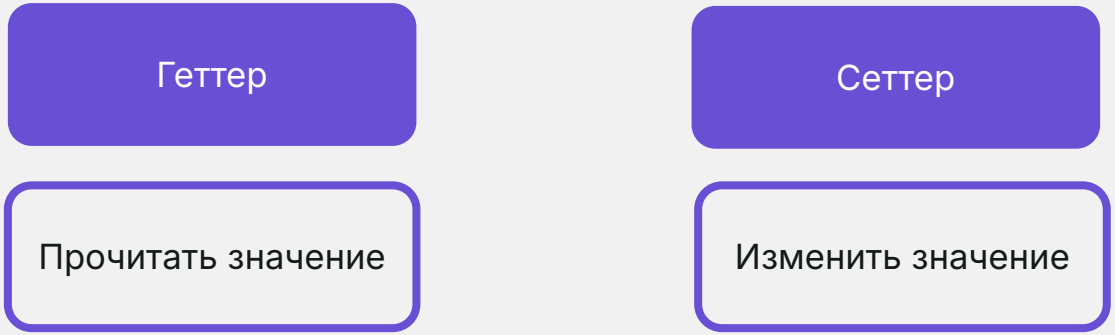
ВОПРОСЫ

IT CAREER HUB

Геттеры и сеттеры

Специальные методы

Для доступа к приватным атрибутам



Причины использования геттеров и сеттеров

- Безопасность
- Гибкость
- Поддержка инкапсуляции

Пример

```
class Temperature:
    def __init__(self):
        self.__celsius = 0  # приватное поле

    def get_celsius(self):
        return self.__celsius  # геттер

    def set_celsius(self, value):
        if value < -273.15:
            raise ValueError("Температура не может быть ниже абсолютного нуля")
        self.__celsius = value  # сеттер

temp = Temperature()
temp.set_celsius(25)       # установка значения
print(temp.get_celsius())  # чтение значения

# temp.set_celsius(-300)   # ValueError
```

Пример: установка значения при создании объекта

```
class Temperature:
    def __init__(self, value):
        self.__validate_celsius(value)
        self.__celsius = value

    def get_celsius(self):
        return self.__celsius

    def set_celsius(self, value):
        self.__validate_celsius(value)
        self.__celsius = value

    @staticmethod
    def __validate_celsius(value):
        if value < -273.15:
            raise ValueError("Температура не может быть ниже абсолютного нуля")

temp1 = Temperature(20)
print(temp1.get_celsius())
# temp2 = Temperature(-500)  # ValueError
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Декоратор @property

Важно

Когда необходимо контролировать доступ к полям, но при этом сохранить читаемый и естественный синтаксис, используется декоратор `@property`

Причины использования @property

- Чтобы скрыть реализацию, но предоставить понятный интерфейс
- Чтобы контролировать доступ к полям
- Чтобы сохранить возможность менять внутреннюю структуру класса, не меняя внешний синтаксис обращения

Декоратор @property

Синтаксис

```
class MyClass:
    def __init__(self, value):
        self.__attr = value

    @property  # всегда использовать property для
геттера
    def attr(self):        # геттер
        return self.__attr

    @attr.setter  # использовать
имя_геттера.setter для сеттера
    def attr(self, value):  # сеттер
        self.__attr = value
```

Пояснение

- `@property` используется для создания **геттера**
- `@attr.setter` используется для создания **геттера**
- Оба метода относятся к **одному "полю"** — `attr`

Пример: температура с валидацией

```python
class Temperature:
    def __init__(self, value):
        self.__celsius = 0
        self.celsius = value  # вызов сеттера внутри
__init__

    @property
    def celsius(self):
        return self.__celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Температура не может
быть ниже абсолютного нуля")
        self.__celsius = value
```

1

```python
t = Temperature(25)
print(t.celsius)    # вызов геттера

t.celsius = 15      # вызов сеттера
print(t.celsius)    # вызов геттера

# t.celsius = -500  # ValueError: Температура не
может быть ниже абсолютного нуля
```

2

Read-only свойства

Для того, чтобы разрешить **только чтение значения**, но **запретить его изменение извне**, создаётся **только геттер с** `@property`, без сеттера.

Пример

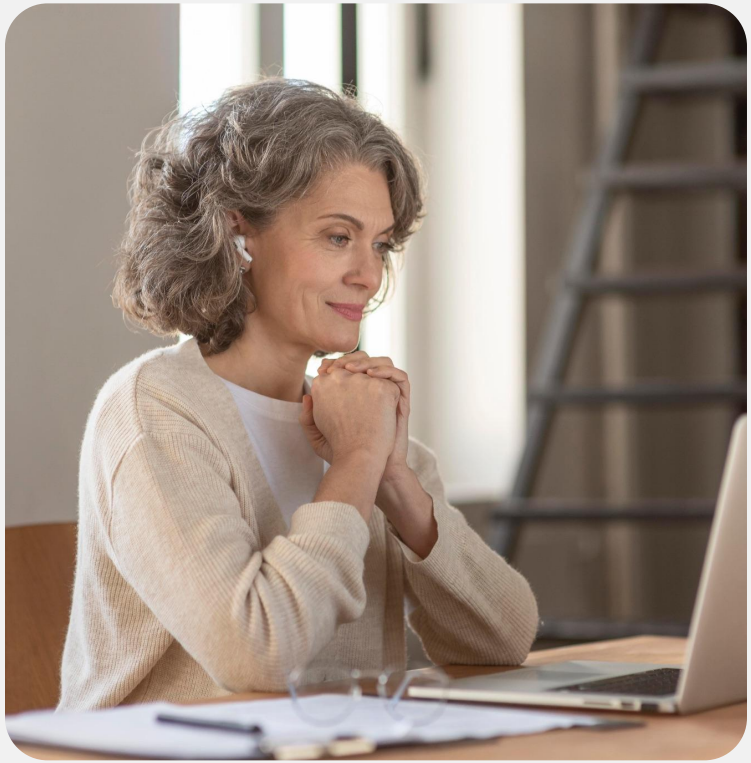
```
class Temperature:
    def __init__(self, celsius):
        self.__celsius = celsius  # приватное поле

    @property
    def celsius(self):
        # геттер для получения значения в цельсиях
        return self.__celsius

    @property
    def fahrenheit(self):
        # геттер вычисления значения в фаренгейтах
        return self.__celsius * 9 / 5 + 32

t = Temperature(25)
print(t.celsius)      # получение значения в фаренгейтах
print(t.fahrenheit)   # вычисление значения в фаренгейтах
# t.celsius = 100     # AttributeError
# t.fahrenheit = 100  # AttributeError
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

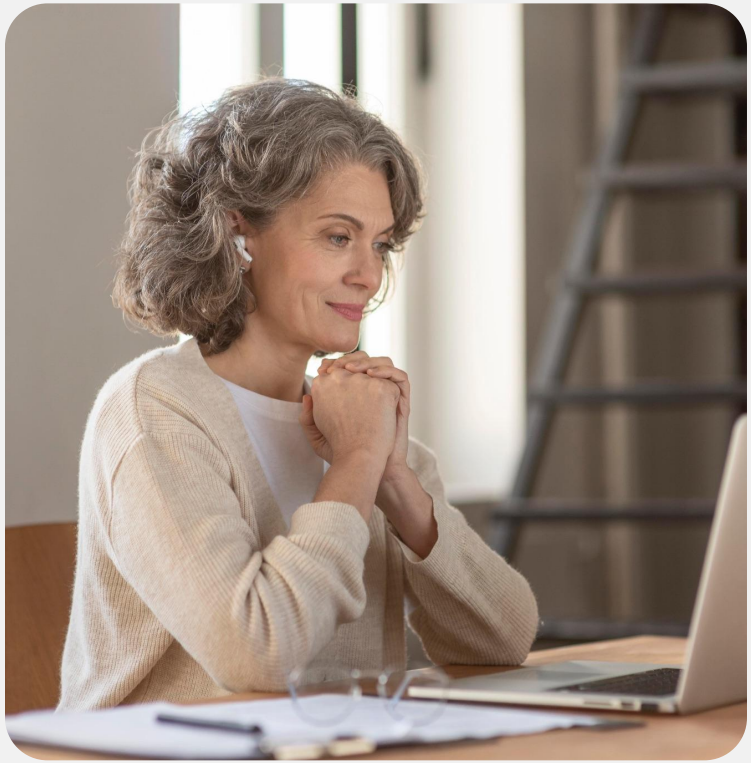
ЗАДАНИЯ



Выполните задание

Найдите ошибку в коде:

```
class Temperature:
    def __init__(self, value):
        self.__celsius = value

    @property
    def __celsius(self):
        return self.__celsius

t = Temperature(25)
print(t.celsius)
```



Выполните задание

Найдите ошибку в коде:

```
class Temperature:
    def __init__(self, value):
        self.__celsius = value

    @property
    def __celsius(self):
        return self.__celsius

t = Temperature(25)
print(t.celsius)
```

Ответ: ошибка в названии метода — должно быть `celsius`, а не `__celsius`.

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ПРАКТИЧЕСКАЯ РАБОТА

1. Безопасная флешка

Создайте класс `SecureUSB`, представляющий защищённую флешку.

- При создании передаётся **секретное содержимое** и пароль
- Метод `unlock(password)` — возвращает `True`, если пароль верный, и разблокирует флешку
- Метод `lock()` — блокирует устройство
- Метод `read()` — возвращает сохранённые данные, если устройство разблокировано, иначе выбрасывает ошибку `PermissionError`.

Продумайте, какие поля следует скрыть, а какие оставить доступными.

Пример вывода:

```
Device is locked.
Access denied.
Access granted.
Data: Secret plans
```

2. Данные через свойство

Доработайте класс `SecureUSB`:

- Переделайте метод `read()` в свойство `data`, используя `@property`.
- Теперь доступ к содержимому осуществляется через обращение к `usb.data`, а не вызов метода.

При попытке чтения в заблокированном состоянии должно по-прежнему выбрасываться `PermissionError`.

Пример вывода:

```
Device is locked. Access denied.
Access granted.
Data: Secret plans
```

IT CAREER HUB

ДОМАШНЕЕ ЗАДАНИЕ

Домашнее задание

1. **Банковский счёт**

Создайте класс `BankAccount`, описывающий банковский счёт.

- Объект должен хранить имя владельца и текущий баланс.
- Реализуйте методы:
  - пополнение счёта
  - снятие средств
  - отображение баланса
- При попытке снять больше, чем есть на счёте, операция не должна выполняться.

Продумайте, какие поля и методы следует скрыть от внешнего доступа, а какие оставить открытыми.

Пример вывода:

```
Current balance: 150
Error: Amount must be positive.
Current balance: 150
Error: Not enough funds.
Current balance: 150
```

Домашнее задание

2. История операций

Доработайте класс `BankAccount`.

- Каждая операция пополнения и снятия должна сохраняться в историю.
- История должна быть доступна через **`property`** `history` только для чтения.
- История представляется в виде списка строк (`"Deposit: 150"`, `"Withdraw: 100"` и т.д.).

Пример вывода:

```
Current balance: 50
Operation history:
        Deposit: 150
        Withdraw: 100
```

Заключение