IT CAREER HUB

Python

Декораторы

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] Вложенные функции
- [x] Область видимости Enclosing
- [x] Вложенные функции
- [x] Область видимости Enclosing
- [x] Ключевое слово nonlocal
- [x] Замыкание
- [x] Функция как объект
- [x] Декораторы
- [x] Синтаксис @decorator

План занятия

- Где применяются декораторы
- Декораторы для функций с аргументами
- Возврат результата из вложенной функции
- Декоратор functools.wraps
- Декораторы с аргументами
- Использование нескольких декораторов

ICH IT CAREER HUB

ОСНОВНОЙ БЛОК

IT CAREER HUB

Где применяются декораторы

Основные области применения декораторов

- Логирование вызовов функций
- Измерение времени выполнения
- Ограничение частоты вызовов (Throttling)
- Проверка и валидация входных данных
- Автоматическое повторение выполнения (Retry)
- Ограничение доступа
- Кеширование результатов
- Автоматическое изменение данных

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Декораторы для функций с аргументами

Аргументы функций

Часто декорируемые функции принимают аргументы. Чтобы декоратор мог работать с такими функциями, его вложенная функция должна уметь принимать и передавать аргументы.

Пример

```python
import logging

# Настраиваем логирование: записи будут сохраняться в файл "functions.log"
logging.basicConfig(
    filename="functions.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    encoding="utf-8"
)

def log_decorator(func):
    def wrapper(*args, **kwargs):  # Принимаем все аргументы функции
        logging.info(f"Функция {func.__name__} вызвана с аргументами: {args}, {kwargs}")
        result = func(*args, **kwargs)  # Передаём аргументы в функцию
        logging.info(f"Функция {func.__name__} вернула: {result}")
        return result  # Возвращаем результат
    return wrapper
```

1

```python
@log_decorator
def add(a, b):
    return a + b

@log_decorator
def say_hello():
    print("Привет!")

print(add(3, 5))  # Декорируемая функция принимает аргументами
say_hello()  # Декорируемая функция без аргументов
```

2

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Возврат результата из вложенной функции

Важно

Если оригинальная функция **возвращает значение**, декоратор должен либо **вернуть его**, либо **заменить на другой результат**

Примеры

С потерей результата

```
def upper_decorator(func):
    def wrapper(*args, **kwargs):
        print("Выполняем функцию, но ничего не
возвращаем")
        # Результат теряется
        func(*args, **kwargs).upper()
    return wrapper

@upper_decorator
def get_text():
    return "hello"

result = get_text()
print("Результат:", result)
```

С возвратом результата

```
def upper_decorator(func):
    def wrapper(*args, **kwargs):
        print("Выполняем функцию и возвращаем
результат")
        # Преобразуем результат в верхний
регистр и возвращаем
        return func(*args, **kwargs).upper()
    return wrapper

@upper_decorator
def get_text():
    return "hello"

result = get_text()
print("Результат:", result)
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Декоратор functools.wraps

Важно

При создании декораторов оригинальная функция **теряет своё имя и документацию**, так как заменяется вложенной функцией (`wrapper`). Декоратор `functools.wraps` помогает **сохранить метаданные** декорируемой функции.

Проблема без functools.wraps

```
def simple_decorator(func):
    def wrapper(*args, **kwargs):
        """Вложенная функция wrapper"""
        print("\nДекорированная функция: ")
        print(f"Оригинальное имя функции: {func.__name__}")
        print(f"Оригинальная документация: {func.__doc__}")
        return func(*args, **kwargs)
    return wrapper

@simple_decorator
def example_function():
    """Это оригинальная функция."""
    print("Привет!")

print("Имя декорированной функции:", example_function.__name__)
print("Документация декорированной функции:", example_function.__doc__)
example_function()
```

Решение с functools.wraps

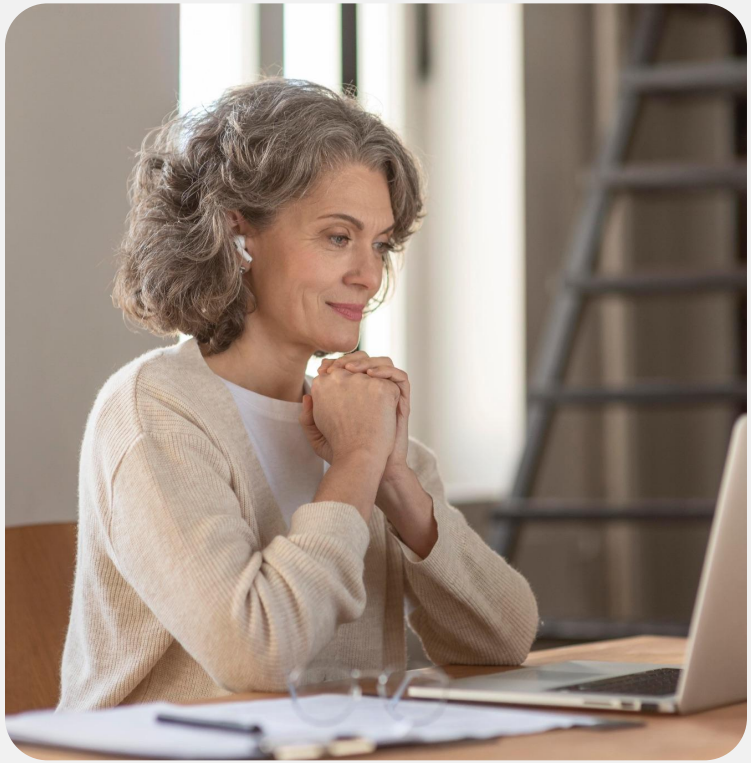
```python
import functools

def simple_decorator(func):
    @functools.wraps(func)  # Сохраняет имя и документацию оригинальной функции
    def wrapper(*args, **kwargs):
        print("\nДекорированная функция")
        return func(*args, **kwargs)
    return wrapper

@simple_decorator
def example_function():
    """Это оригинальная функция."""
    print("Привет!")

print("Имя декорированной функции:", example_function.__name__)
print("Документация декорированной функции:", example_function.__doc__)
example_function()
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

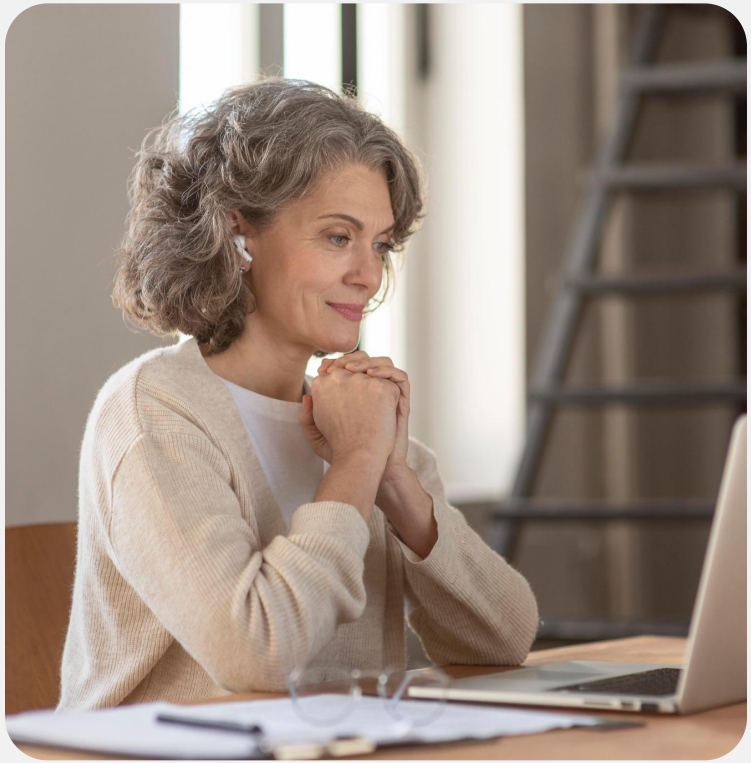
ЗАДАНИЯ



Выберите верный вариант ответа

1. **Что делает functools.wraps?**

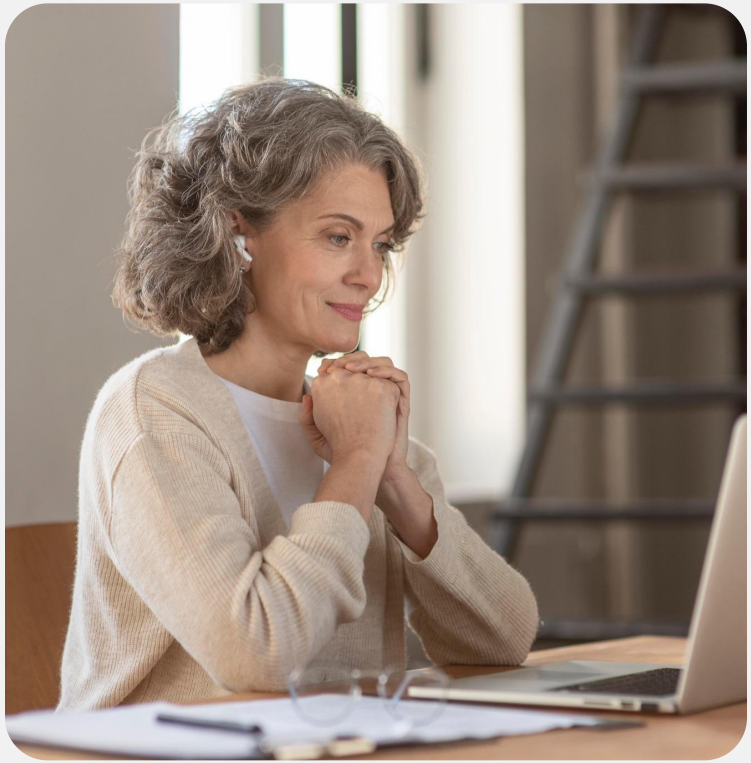
a. Удаляет лишние аргументы
b. Автоматически вызывает декорируемую функцию
c. Сохраняет имя и документацию исходной функции



Выберите верный вариант ответа

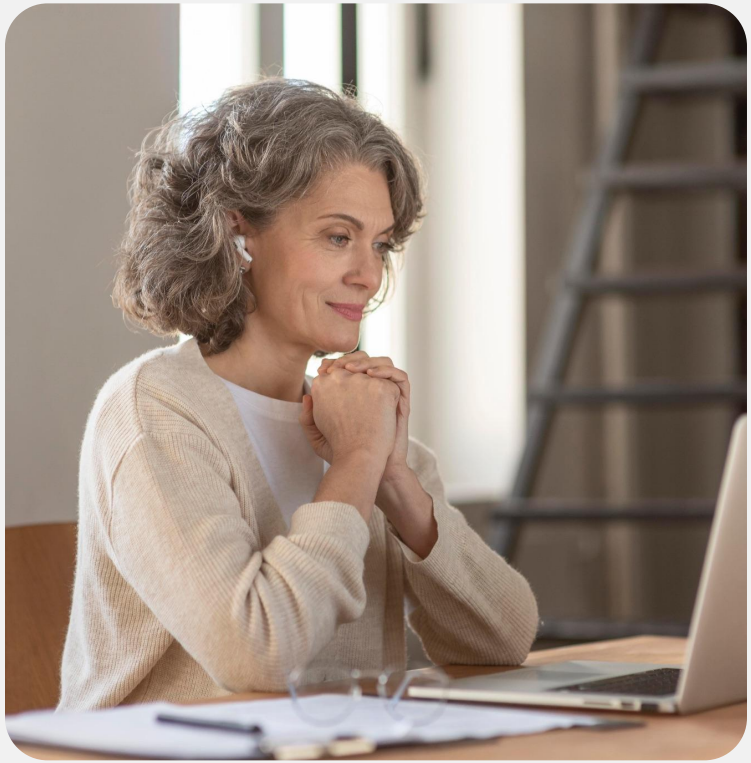
1. **Что делает functools.wraps?**

a. Удаляет лишние аргументы
b. Автоматически вызывает декорируемую функцию
c. Сохраняет имя и документацию исходной функции



Ответьте на вопрос

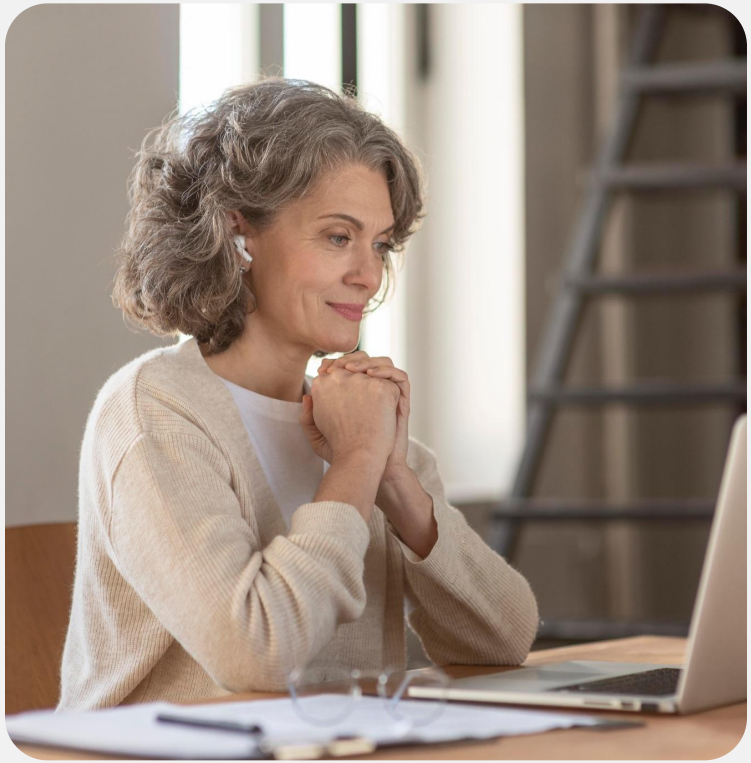
2. Почему без functools.wraps теряется документация функции?



Ответьте на вопрос

2. Почему без functools.wraps теряется документация функции?

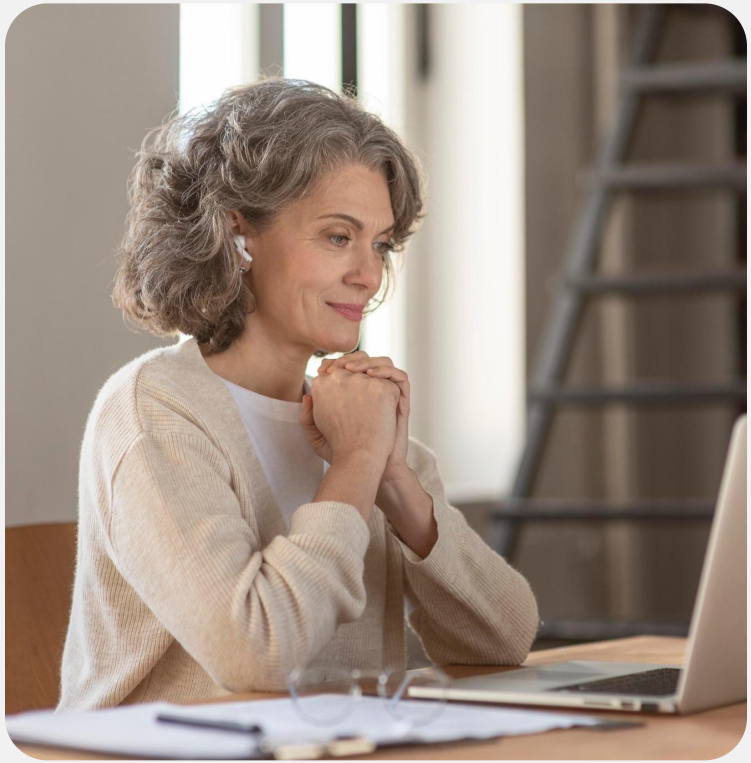
Ответ: потому что `wrapper` заменяет функцию, и метаданные не переносятся



Выберите верный вариант ответа

3. Для чего нужны *args и **kwargs в декораторах?

a. Для совместимости с функциями без аргументов
b. Для совместимости с функциями с любыми аргументами
c. Для сокращения кода



Выберите верный вариант ответа

3. Для чего нужны *args и **kwargs в декораторах?

a. Для совместимости с функциями без аргументов
b. Для совместимости с функциями с любыми аргументами
c. Для сокращения кода

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Декораторы с аргументами

Важно

Чтобы декоратор принимал аргументы, создаётся функция **декоратор-фабрика**, которая возвращает сам декоратор.

Декоратор с настраиваемым сообщением

```
def message_decorator(message):
    def decorator(func):
        def wrapper():
            print(message)  # Используем переданный аргумент
            return func()
        return wrapper
    return decorator

@message_decorator("Начинаем выполнение")
def analise_data():
    print("Данные проанализированы")

@message_decorator("Загрузка данных...")
def load_data():
    print("Данные загружены")

analise_data()
print()
load_data()
```

Автоматическое повторение функции при ошибке

```python
import time

def retry(attempts):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for i in range(attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Попытка {i+1} не
удалась: {e}")
                    time.sleep(5)  # Подождём перед
новой попыткой
            print("Все попытки исчерпаны.")
        return wrapper
    return decorator
```

1

```python
@retry(3)
def get_data(filename):
    """Читает данные из файла"""
    with open(filename, "r", encoding="utf-8") as
file:
        return file.read()

# Пример использования
data = get_data("data.txt")
if data:
    print("Содержимое файла:")
    print(data)
```

2

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Использование нескольких декораторов

Особенности применения нескольких декораторов

- Декораторы выполняются снизу вверх
- Можно комбинировать несколько независимых декораторов
- Важно учитывать порядок выполнения, так как один декоратор может изменить результат перед следующим

Пример сочетания декораторов

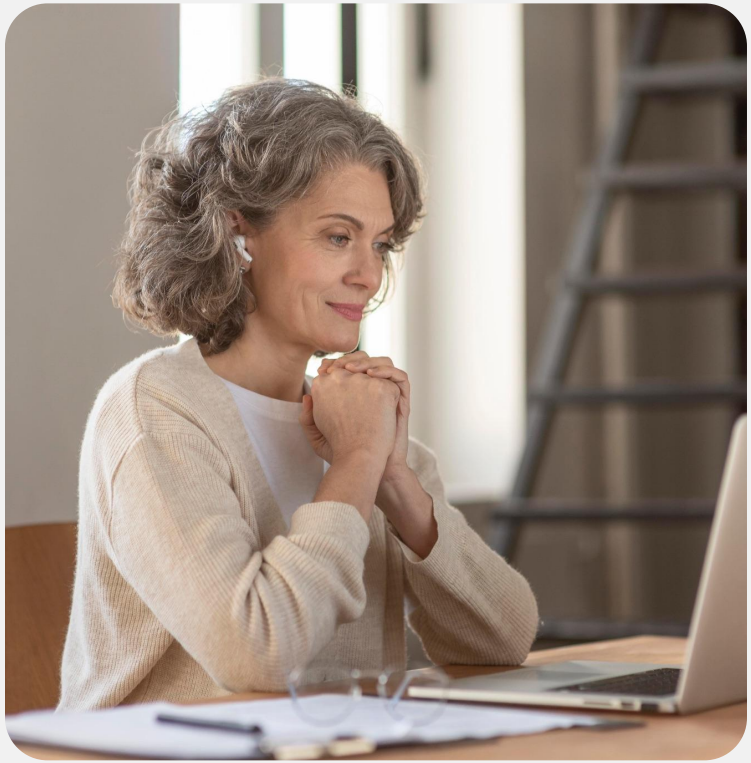
```
def border_decorator(func):
    def wrapper():
        print("*" * 100)  # Верхняя граница
        func()
        print("*" * 100)  # Нижняя граница
    return wrapper

def repeat_decorator(func):
    def wrapper():
        for _ in range(3):  # Повторяем вызов трижды
            func()
    return wrapper

@border_decorator  # Применяется вторым
@repeat_decorator  # Применяется первым
def print_line():
    print("-" * 100)

print_line()
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

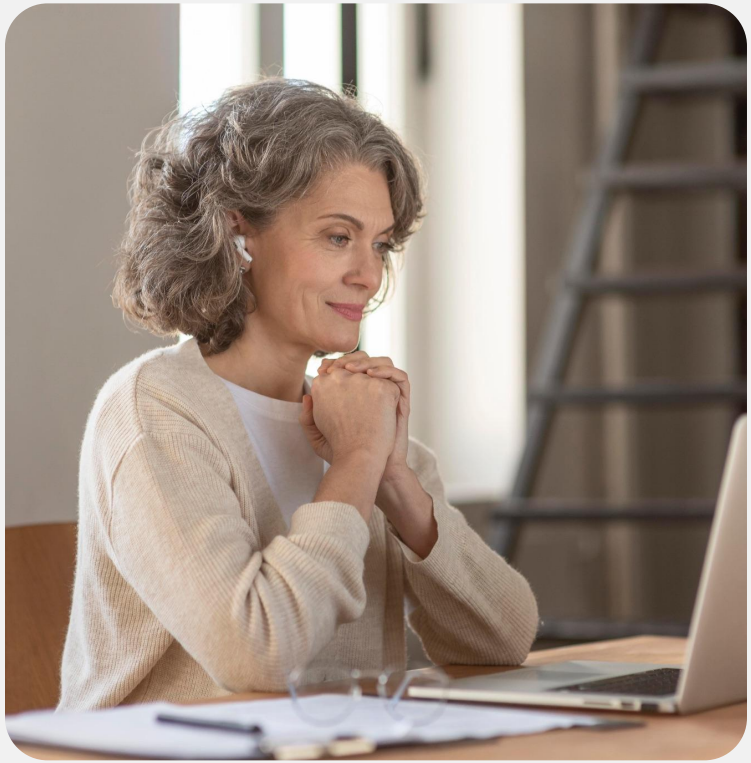
ЗАДАНИЯ



Ответьте на вопрос

1. **Что делает следующий декоратор?**

```
def custom_decorator(message):
    def decorator(func):
        def wrapper():
            try:
                return func()
            except Exception:
                print(message)
        return wrapper
    return decorator
```

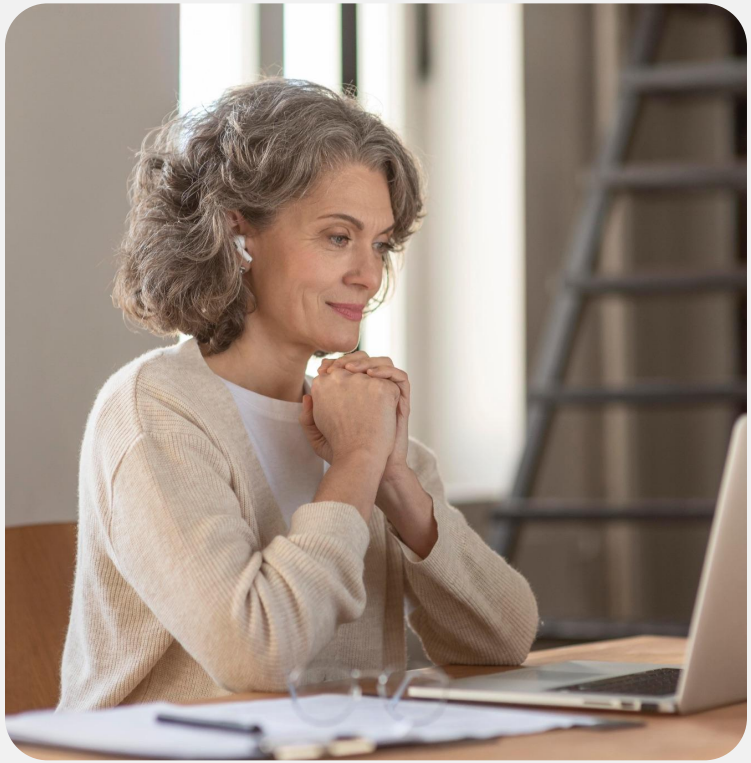


Ответьте на вопрос

1. **Что делает следующий декоратор?**

```
def custom_decorator(message):
    def decorator(func):
        def wrapper():
            try:
                return func()
            except Exception:
                print(message)
        return wrapper
    return decorator
```

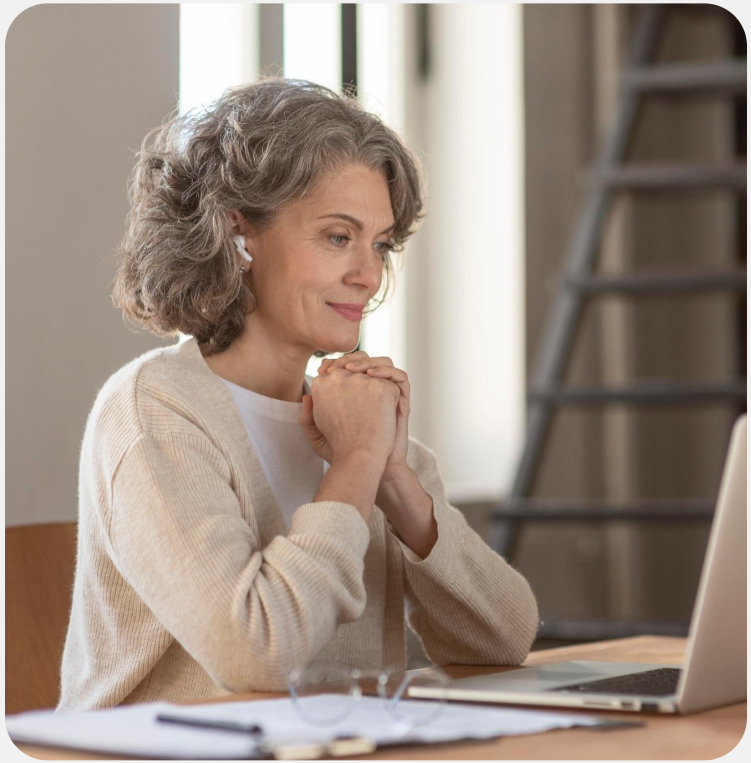
Ответ: при возникновении ошибки выводит указанное сообщение



Выберите верный вариант ответа

2. Какая конструкция позволит задать аргументы декоратору?

a. `@decor("info")`
b. `@decor["info"]`
c. `@decor = "info"`
d. `@decor: "info"`



Выберите верный вариант ответа

2. Какая конструкция позволит задать аргументы декоратору?

a. `@decor("info")`
b. `@decor["info"]`
c. `@decor = "info"`
d. `@decor: "info"`

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ПРАКТИЧЕСКАЯ РАБОТА

1. Рамка-обводка

Создайте декоратор `framed`, который **оборачивает результат** в рамку из символов = длиной 40.

Пример применения:

```
@framed
def show_title():
    print("== Menu ==")
```

Пример вывода:

```
========================================
== Menu ==
========================================
```

2. Настраиваемая рамка-обводка

Доработайте декоратор `framed`, чтобы он принимал параметр `width`, определяющий ширину рамки, и параметр `symbol`, определяющий символ для рамки (по умолчанию `"="`).

Пример применения:

```
@framed(30, "-")
def show_title():
    print("== Menu ==")
```

Пример вывода:

```
------------------------------
== Menu ==
------------------------------
```

IT CAREER HUB

ДОМАШНЕЕ ЗАДАНИЕ

Домашнее задание

1. Среднее время выполнения

Создайте декоратор `measure_time`, который измеряет и выводит **среднее время выполнения** функции за 5 вызовов. Функция может быть любой: например, сортировка списка, чтение из файла или расчёты.

Пример применения:

```
@measure_time
def compute():
    total = 0
    for i in range(10_000_000):
        total += i
    return total
```

Пример вывода:

```
Среднее время выполнения для 5 вызовов:
0.21 секунд
Результат: 49999995000000
```

Домашнее задание

2. Среднее время выполнения с количеством вызовов

Доработайте декоратор `measure_time`, чтобы он принимал параметр `repeats` — количество вызовов функции. Декоратор должен выполнять функцию указанное число раз и выводить **среднее время выполнения**.

Пример применения:

```
@measure_time(10)
def compute():
    total = 0
    for i in range(10_000_000):
        total += i
    return total
```

Пример вывода:

```
Среднее время выполнения для 10 вызовов:
0.21 секунд
Результат: 49999995000000
```

Заключение