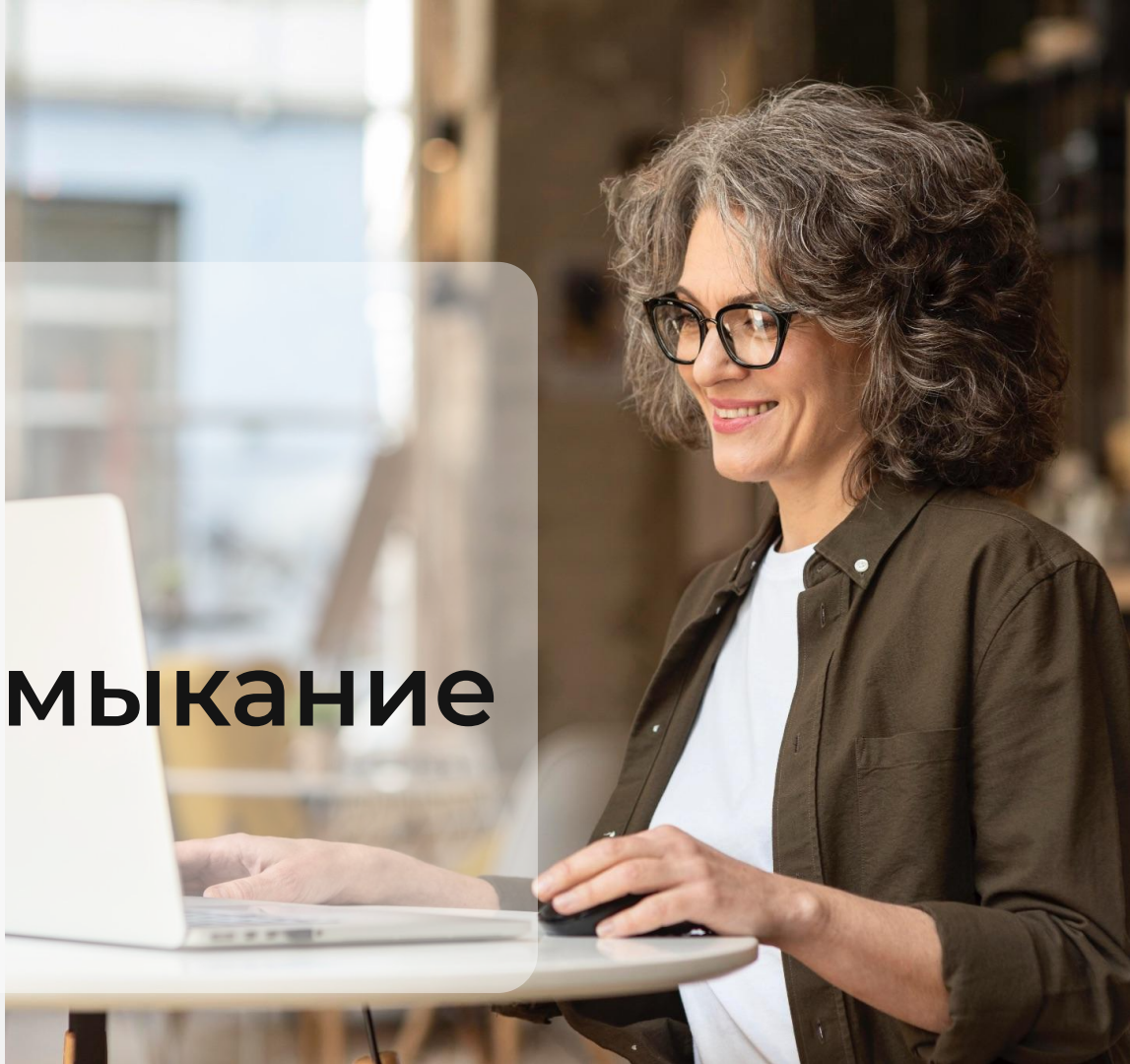
IT CAREER HUB

Python

Вложенные функции. Замыкание



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

- Регулярные выражения
- Модуль re
- Функция findall
- Основные символы в регулярных выражениях
- Символ 'r' перед строкой шаблона
- Классы символов
- Квантификаторы
- Жадные и ленивые квантификаторы
- Экранирование специальных символов
- Якоря
- Альтернативы
- Группы
- Функции модуля

План занятия

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

ICH IT CAREER HUB

ОСНОВНОЙ БЛОК

ICH IT CAREER HUB

Вложенные функции

Важно

В Python одна функция может быть определена внутри другой. Такие функции называются **вложенными**. Это полезно, когда внутренняя функция выполняет вспомогательные задачи, которые не должны быть доступны за пределами внешней функции.

Назначение вложенных функций

- Инкапсуляция логики
- Локальная область видимости
- Избегание дублирования кода

Пример простой вложенной функции

```
def outer_function():
    print("Внутри внешней функции")

    def inner_function():
        print("Внутри вложенной функции")

    inner_function()  # Вызов вложенной функции

outer_function()
# inner_function()  # Вызовет ошибку
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Область видимости Enclosing

Полезно знать

IT CAREER HUB

Enclosing

`Enclosing` (охватывающая) область – это область, содержащая вложенную функцию

Очередность областей видимости

LEGB

- `Local`
- `Enclosing`
- `Global`
- `Built-in`

Пояснения

Если во вложенной функции используется переменная, но она не объявлена в ней, Python ищет её в **Enclosing** области

Пример

```
def outer_function(repeat):
    message = "Внешняя функция\n"

    def inner_function():
        print(message * repeat)  # Переменные внешней функции

    inner_function()

outer_function(3)
```

Локальные переменные во вложенных функциях

Когда во вложенной функции создается переменная с таким же именем, как во внешней, это **не изменяет** значение внешней переменной. Вместо этого во вложенной функции создается **новая локальная переменная**.

Пример

```
def outer_function():
    message = "Внешняя функция"

    def inner_function():
        message = "Вложенная функция"  # Создается новая локальная
переменная

    inner_function()
    print(message)  # Выведет неизмененное значение внешней переменной

outer_function()
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Ключевое слово nonlocal

Важно

Если во вложенной функции нужно изменить переменную, объявленную во внешней функции, используется ключевое слово `nonlocal`. Оно указывает, что переменная **принадлежит не локальной, а внешней области видимости**

Пример

```
def outer_function():
    message = "Внешняя функция"

    def inner_function():
        nonlocal message  # Указываем, что message принадлежит внешней
функции
        message = "Изменено во вложенной функции"

    inner_function()
    print(message)  # Теперь message изменилось

outer_function()
```

Особенности nonlocal

- `nonlocal` работает только для ближайшей внешней функции
- Если вложенность больше одного уровня, `nonlocal` изменяет переменную **из ближайшей внешней функции**, но не из глобальной области

Пример

```
def outer_function():
    message = "Внешняя функция"

    def middle_function():
        def inner_function():
            nonlocal message
            message = "Изменено во вложенной функции"

        inner_function()

    middle_function()
    print(message)

outer_function()
```

Вспомогательные функции внутри основной

Пример

```
def process_data(data):
    def clean_text(text):
        return text.strip().lower()

    cleaned_data = [clean_text(item) for item
in data]
    return cleaned_data

data = ["  Apple  ", " BaNaNa ", " CHERRY "]
print(process_data(data))
```

Пояснения

Если определенная часть кода используется только внутри одной функции, её можно оформить как вложенную. Это делает код более читаемым и упрощает структуру программы

Разделение кода на логические части

Пример

```
def analyze_text(text):
    def count_words():
        return len(text.split())

    def count_letters():
        return sum(1 for char in text if
char.isalpha())

    print(f"Слов: {count_words()}")
    print(f"Букв: {count_letters()}")

analyze_text("Пример текста!")
```

Пояснения

Иногда вложенные функции помогают разделить код на более понятные части, особенно если внутри основной функции есть несколько шагов обработки

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Замыкание

Полезно знать

ICH
IT CAREER HUB

Замыкание (closure)

Это объект, содержащий функцию и сохранённое окружение (переменные из охватывающей области Enclosing), которые остаются доступными даже после завершения внешней функции

Работа замыкания

Когда внешняя функция возвращает вложенную функцию, эта вложенная функция **запоминает** переменные из внешней функции и может использовать их при последующих вызовах

Пример замыкания

```
def outer_function(text):
    def inner_function():
        print(text)  # Запоминает переменную text

    return inner_function  # Возвращаем незапущенную функцию

closure = outer_function("Переданный текст")  # Объект замыкания
print(closure)

closure()  # Вызываем внутреннюю функцию после завершения внешней
```

Замыкание с изменением переменной

Если нужно изменять переменную внешней функции, используется `nonlocal`

```
def counter():
    count = 0

    def increment():
        nonlocal count  # Используем count из enclosing-области
        count += 1
        return count

    return increment  # Возвращаем функцию

counter_function = counter()
print(counter_function())
print(counter_function())
print(counter_function())
```

Фильтрация данных с параметром

Можно создать функцию, которая возвращает фильтр с предустановленным значением

```
def create_filter(border):
    def filter_value(value):
        return value > border  # Использует сохранённый border
    return filter_value

greater_than_five = create_filter(5)  # Объект замыкания
print(greater_than_five)
print(greater_than_five(7))
print(greater_than_five(3))
```

Настраиваемые математические операции

С помощью замыкания можно создавать функции с разными коэффициентами

```
def multiplier(factor):
    def multiply(number):
        return number * factor  # Использует сохранённый factor
    return multiply

double = multiplier(2)  # Сохраняет 2 в переменной factor
triple = multiplier(3)  # Сохраняет 3 в переменной factor

print(double(4))
print(triple(4))
```

Создание замыкания для кеширования

Если вычисление занимает много времени, можно сохранить результаты в замыкании

1

```
import time

def long_function(num):
    time.sleep(3)
    return list(range(num))

def memoize():
    cache = {}

    def get_or_compute(key,
compute_function):
        if key not in cache:
            cache[key] =
compute_function(key)
        return cache[key]
```

2

```
return get_or_compute

cached_computation = memoize()

start = time.time()
print(cached_computation(10,
long_function))  # Долгая операция
print("Время расчёта:", time.time() -
start)

start = time.time()
print(cached_computation(10,
long_function))  # Берёт из кеша
(быстро)
print("Время получения из кэша:",
time.time() - start)
```

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Функция как объект

Важно

В Python функция — это объект, у которого есть **атрибуты**, хранящие служебную информацию. К ним можно получить доступ напрямую

Пример: получение имени и документации

```
def greet():
    """Функция приветствия"""
    print("Привет!")

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

Пример: информация о вложенной функции

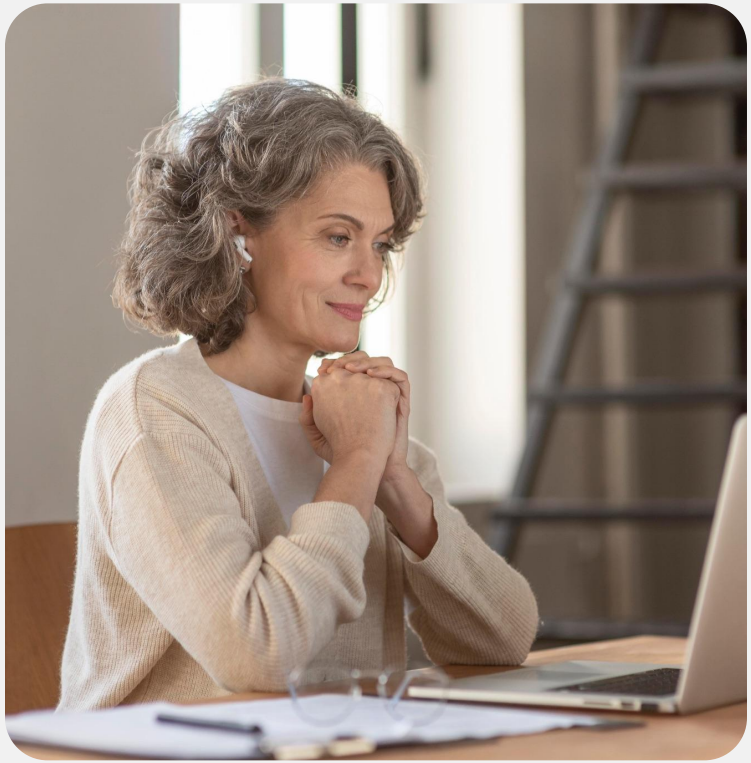
```
def outer():
    def inner():
        """Вложенная функция"""
        pass
    return inner

func = outer()
print("Имя функции:", func.__name__)
print("Документация:", func.__doc__)
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ

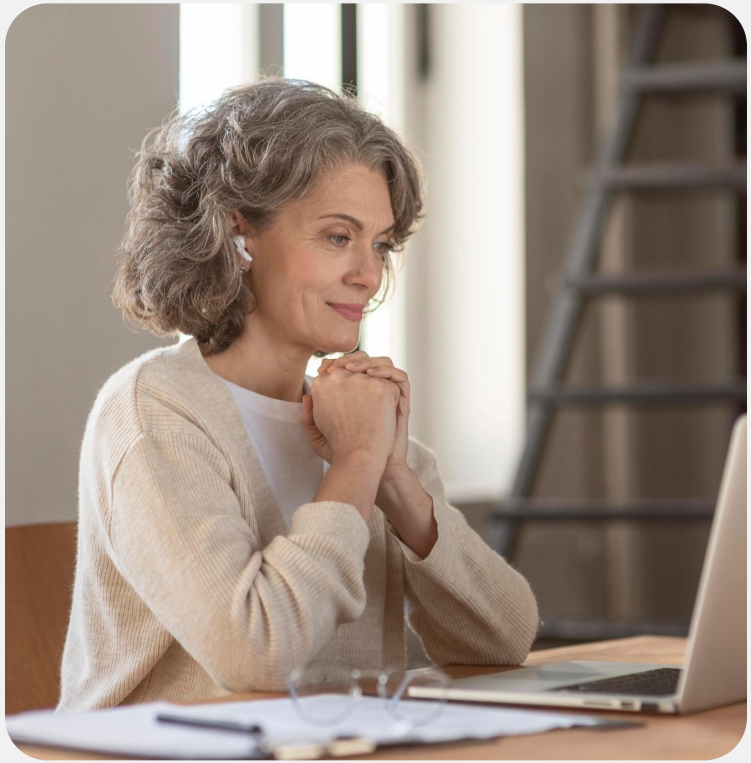


Выберите верный вариант ответа

1. **Что произойдёт при запуске следующего кода?**

```
def outer():
    def inner():
        print("Hi")
    return inner()

result = outer()
result()
```

a. Будет выведено: `Hi`
b. Будет выведено: `outer`
c. Будет ошибка: `inner is not defined`
d. Будет ошибка: `NoneType object is not callable`

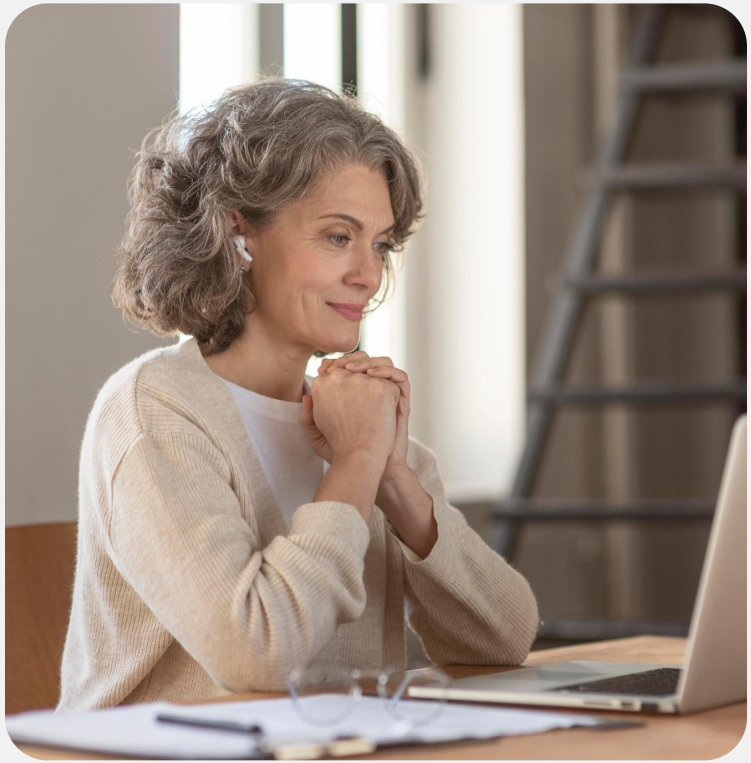


Выберите верный вариант ответа

1. **Что произойдёт при запуске следующего кода?**

```
def outer():
    def inner():
        print("Hi")
    return inner()

result = outer()
result()
```

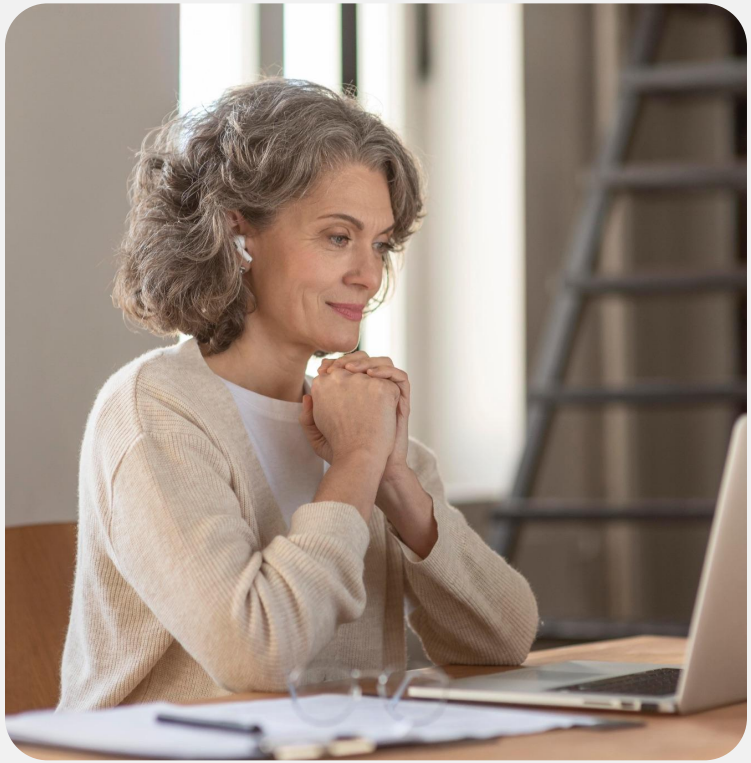
a. Будет выведено: `Hi`
b. Будет выведено: `outer`
c. Будет ошибка: `inner is not defined`
d. Будет ошибка: `NoneType object is not callable`



Сопоставьте понятие с его описанием

1. Вложенная функция
2. nonlocal
3. Enclosing
4. Замыкание

a. Ключевое слово
b. Функция, определённая внутри другой
c. Функция, возвращаемая с сохранёнными переменными внешней области
d. Область, в которой определены переменные для вложенной функции



Сопоставьте понятие с его описанием

1. Вложенная функция
2. nonlocal
3. Enclosing
4. Замыкание

a. Ключевое слово
b. Функция, определённая внутри другой
c. Функция, возвращаемая с сохранёнными переменными внешней области
d. Область, в которой определены переменные для вложенной функции

Ответ: 1-b, 2-a, 3-d, 4-c

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Декораторы

Полезно знать

ICH
IT CAREER HUB

Декораторы

Это способ изменить поведение функции, не изменяя её код. Декоратор принимает функцию, добавляет к ней новую логику и возвращает изменённую версию

Важно

В Python функции являются объектами, поэтому их можно передавать и возвращать из других функций. Декораторы используют это свойство

Как работает декоратор

Создаётся функция-декоратор, принимающая другую функцию

1

Внутри декоратора объявляется **вложенная функция**, которая выполняет дополнительный код перед и/или после вызова исходной функции

2

Декоратор возвращает эту вложенную функцию

3

Пример

```
def simple_decorator(func):    # Функция-декоратор, принимает другую
функцию
    def wrapper():  # Вложенная функция-обертка, добавляющая
дополнительное поведение
        print("Перед вызовом функции")
        func()  # Вызываем переданную функцию
        print("После вызова функции")
    return wrapper  # Возвращаем изменённую функцию

def say_hello():
    print("Привет!")

decorated = simple_decorator(say_hello)  # Вызываем декоратор, теперь
decorated = wrapper
print(decorated)
decorated()  # Теперь вызов say_hello() происходит через wrapper
```

Важно

После применения декоратора результат (`wrapper`) можно сохранить в переменной с тем же именем, что и декорируемая функция.
Тогда вместо вызова оригинальной функции будет вызываться функция `wrapper`, добавляющая дополнительный функционал

Пример

```
def simple_decorator(func):    # Функция-декоратор, принимает другую
функцию
    def wrapper():  # Вложенная функция-обертка, добавляющая
дополнительное поведение
        print("Перед вызовом функции")
        func()  # Вызываем переданную функцию
        print("После вызова функции")
    return wrapper  # Возвращаем изменённую функцию

def say_hello():
    print("Привет!")

say_hello = simple_decorator(say_hello)  # Вызываем декоратор, теперь
decorated = wrapper
print(say_hello)
say_hello()  # Теперь вызов say_hello() происходит через wrapper
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Синтаксис @decorator

Важно

Вместо явного вызова `decorated = simple_decorator(say_hello)` можно использовать `@` с именем декоратора перед определением функции

Пример

```
def simple_decorator(func):    # Функция-декоратор, принимает другую функцию
    def wrapper():  # Вложенная функция, добавляющая дополнительное поведение
        print("Перед вызовом функции")
        func()  # Вызываем переданную функцию
        print("После вызова функции")
    return wrapper  # Возвращаем изменённую функцию

@simple_decorator  # Эквивалентно say_hello = simple_decorator(say_hello)
def say_hello():
    print("Привет!")

say_hello()
```

Важно

Декораторы полезны, когда нужно добавлять дополнительное поведение к функциям. Один из распространённых примеров — декоратор для измерения времени выполнения

Пример

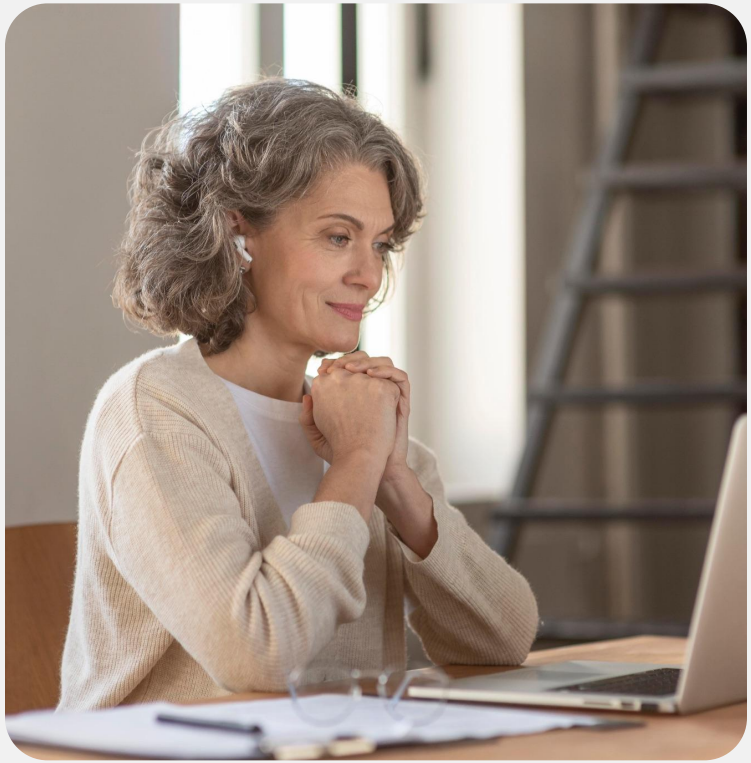
```
import time

def timing_decorator(func):
    def wrapper():
        start_time = time.time()  # Засекаем время перед выполнением
функции
        func()  # Вызываем декорируемую функцию
        end_time = time.time()  # Засекаем время после выполнения
        print(f"Функция {func.__name__} выполнялась {end_time -
start_time:.5f} секунд")
    return wrapper
@timing_decorator  # Применение декоратора
def slow_function():
    time.sleep(2)  # Имитация долгой операции
    print("Функция выполнена")

slow_function()
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

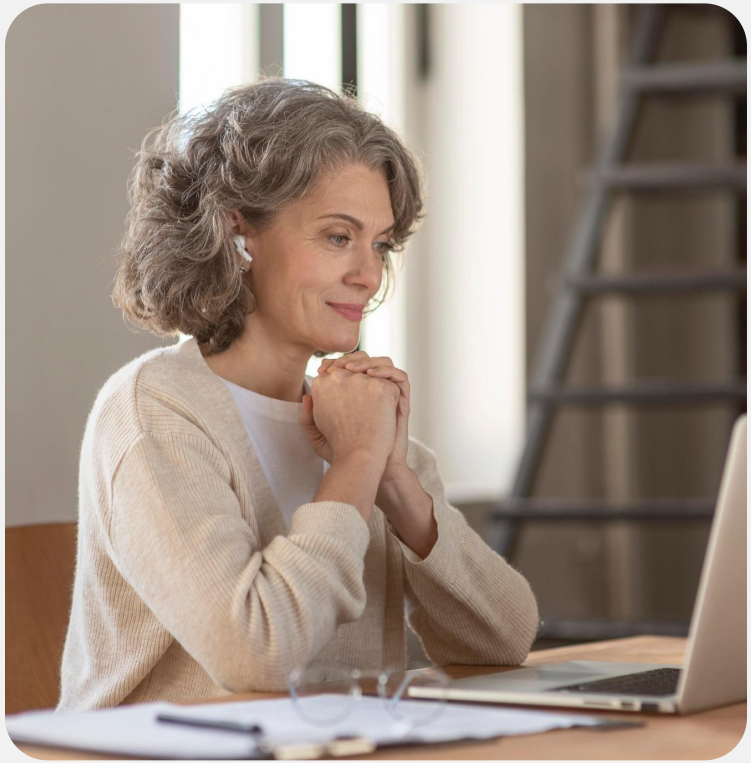
ЗАДАНИЯ



Выберите верный вариант ответа

Что делает декоратор?

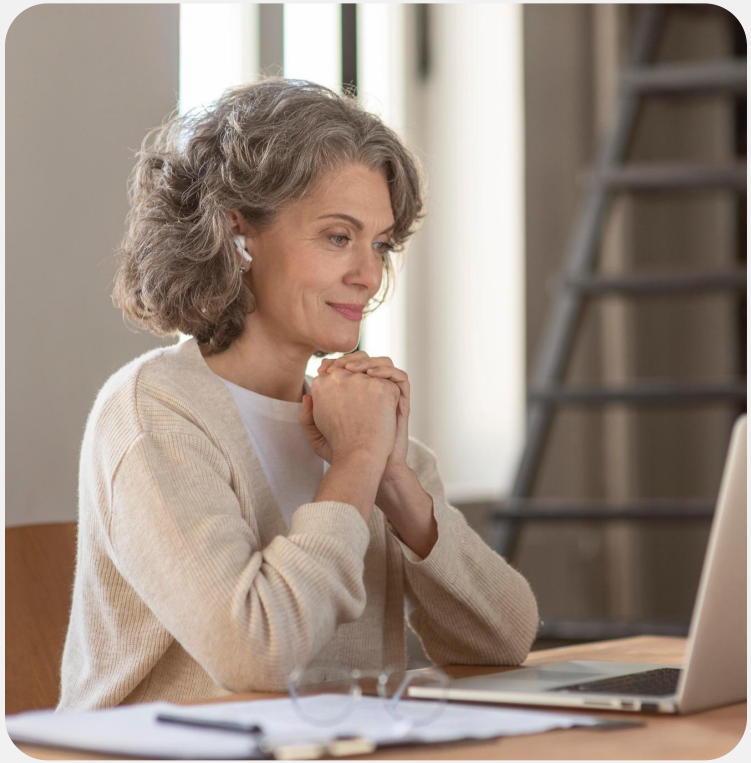
a. Изменяет код функции внутри её тела
b. Добавляет дополнительную логику к функции без изменения её тела
c. Копирует поведение функции в другую переменную



Выберите верный вариант ответа

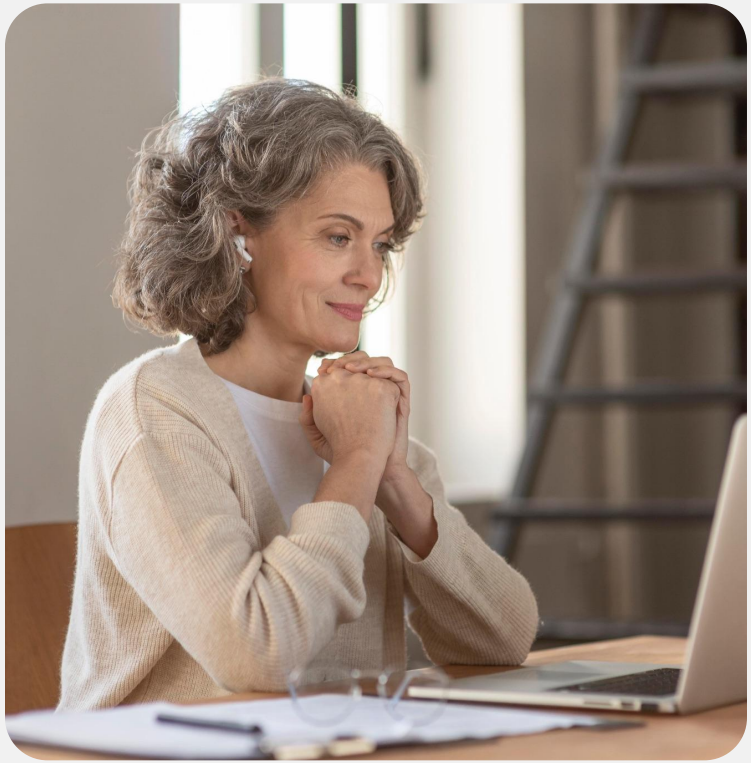
Что делает декоратор?

a. Изменяет код функции внутри её тела
b. Добавляет дополнительную логику к функции без изменения её тела
c. Копирует поведение функции в другую переменную



Ответьте на вопрос

Что делает конструкция `@decorator_name`?



Ответьте на вопрос

Что делает конструкция

`@decorator_name`?

Ответ: Заменяет функцию на результат работы декоратора

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ПРАКТИЧЕСКАЯ РАБОТА

Фабрика функций расчёта НДС

Создайте функцию `vat_calculator(rate)`, которая принимает ставку НДС и возвращает другую функцию. Полученная функция должна принимать сумму и возвращать цену **с учётом НДС** по переданной ставке.

Пример вызова:

```
print(vat_20(100))
print(vat_10(200))
```

Пример вывода:

```
120.0
220.0
```

Калькулятор скидок по категориям

Создайте функцию, которая возвращает другую функцию для расчёта скидки. Внешняя функция принимает словарь скидок, например `{"food": 0.1}` — 10% на еду. Если категория не найдена — цена не меняется.

Данные:

```
discounts = {"food": 0.1, "clothes": 0.2}
```

Пример вызова:

```
discounts = {"food": 0.1, "clothes": 0.2}

print(friday_discount("food", 100))
print(friday_discount("clothes", 250))
print(friday_discount("electronics", 500))
```

Пример вывода:

```
90.0
200.0
500
```

Настроенная функция вывода

Создайте функцию `custom_printer(sep, end)`, которая возвращает **новую функцию печати**, использующую указанные значения `sep` и `end` по умолчанию.

Пример вызова:

```
printer = custom_printer(sep=' | ', end=' -->\n')

printer('Hello', 'World')
printer('Python', 'Java', 'C++')
```

Пример вывода:

```
Hello | World -->
Python | Java | C++ -->
```

Нумерация вызовов функции

Создайте декоратор `call_counter`, который выводит **имя и номер вызова функции** каждый раз, когда она вызывается.
Номер должен увеличиваться при каждом вызове.

Пример декорируемой функции:

```
def greet():
    print("Привет!")
```

Пример вывода:

```
Вызов функции 'greet' №1:
Привет!
Вызов функции 'greet' №2:
Привет!
Вызов функции 'greet' №3:
Привет!
```

IT CAREER HUB

ДОМАШНЕЕ ЗАДАНИЕ

Домашнее задание

1. Фабрика функций округления

Создайте функцию `make_rounder()`, которая принимает количество знаков для округления и возвращает другую функцию. Полученная функция должна принимать число и возвращать его, округлённое до указанного ранее количества знаков после запятой.

Пример вызова:

```
print(round2(3.14159))
print(round2(2.71828))
print(round0(9.999))
```

Пример вывода:

```
3.14
2.72
10.0
```

Домашнее задание

2. Расширяемый логгер событий

Создайте функцию, которая возвращает вложенный **логгер событий**. Каждый вызов логгера должен **сохранять событие с текущим временем** (если оно передано) и **возвращать весь список событий**.

Пример вызова:

```
log("Загрузка данных")
log("Обработка завершена")
log("Сохранение файла")
for event in log():
    print(event)
```

Пример вывода:

```
Загрузка данных: 2025-03-24 14:06:29
Обработка завершена: 2025-03-24 14:06:29
Сохранение файла: 2025-03-24 14:06:29
```

Домашнее задание

3. Рамка вокруг вывода

Создайте декоратор `frame`, который **оборачивает результат функции рамкой** из 50 символов `-`, выводя **по строке до и после** вызова функции.

Пример декорируемой функции:

```
def say_hello():
    print("Привет, игрок!")
```

Пример вывода:

```
--------------------------------------------------
Привет, игрок!
--------------------------------------------------
```

Заключение