IT CAREER HUB

Урок 32.
Вложенные функции. Замыкание

IT CAREER HUB

Вложенные функции

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.

Зачем нужны вложенные функции?

- **Инкапсуляция логики** – скрывают вспомогательный код внутри основной функции.
- **Локальная область видимости** – переменные вложенной функции недоступны извне, что предотвращает конфликты имен.
- **Избегание дублирования кода** – можно повторно использовать логику внутри одной функции.

Пример простой вложенной функции

```python
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

Область видимости Enclosing

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

Очередность областей видимости (**LEGB**):

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

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

Пример

```python
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**.

IT CAREER HUB

Пример

Python

```python
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

Ключевое слово nonlocal

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

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

Пример

```python
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()
```

Примеры использования вложенных функций

1. Вспомогательные функции внутри основной

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

Python
```
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))
```

2. Разделение кода на логические части

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

IT CAREER HUB

```python
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!")
```

IT CAREER HUB

Замыкание

Замыкание (closure) — это объект, содержащий функцию и сохранённое окружение (переменные из охватывающей области Enclosing), которые остаются доступными даже после завершения внешней функции.

Как работает замыкание?

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

Пример замыкания

```python
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
```

Примеры применения замыканий

1. Замыкание с изменением переменной

Если нужно изменять переменную внешней функции, используется `nonlocal`:

```python
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())
```

2. Фильтрация данных с параметром

Можно создать функцию, которая возвращает фильтр с предустановленным значением.

```python
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))
```

3. Настраиваемые математические операции

С помощью замыкания можно создавать функции с разными коэффициентами.

```python
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))
```

4. Создание замыкания для кеширования

Если вычисление занимает много времени, можно сохранить результаты в замыкании.

```python
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]

    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

Функция как объект

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

Пример: получение имени и документации

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

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

Пример: информация о вложенной функции

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

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

IT CAREER HUB

Задания для закрепления 1

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

```python
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`

Посмотреть ответ

2. 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

Посмотреть ответ

IT CAREER HUB

Декораторы

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

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

Как работает декоратор?

1. Создаётся **функция-декоратор**, принимающая другую функцию.
2. Внутри декоратора объявляется **вложенная функция**, которая выполняет дополнительный код перед и/или после вызова исходной функции.
3. Декоратор возвращает эту вложенную функцию.

Пример

```python
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
```

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

Python

```python
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

Синтаксис @decorator

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

```python
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()
```

Пример использования декоратора

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

```python
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

Задания для закрепления 2

1. **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

Посмотреть ответ

2. Что делает конструкция @decorator_name?

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Результат выполнения кода	Ответ: d
2. Сопоставление	Ответ: 1-b, 2-a, 3-d, 4-c
Задания на закрепление 2	Вернуться к заданиям
1. Работа декоратора	Ответ: b
2. Назначение конструкции @decorator_name	Ответ: Заменяет функцию на результат работы декоратора

IT CAREER HUB

🔎 Практическая работа

1. 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:

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

Пример вывода:

```
120.0
220.0
```

IT CAREER HUB

Решение:

```python
def vat_calculator(rate):
    def calculate(price):
        return round(price * (1 + rate), 2)
    return calculate

vat_20 = vat_calculator(0.2)
vat_10 = vat_calculator(0.1)

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

IT CAREER HUB

2. 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:

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

Пример вызова:

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

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

Пример вывода:

```python
90.0
200.0
500
```

Решение:

```python
def discount_maker(category_discounts):
    def apply_discount(category, price):
        discount = category_discounts.get(category, 0)
        return round(price * (1 - discount), 2)
    return apply_discount

discounts = {"food": 0.1, "clothes": 0.2}
friday_discount = discount_maker(discounts)

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

IT CAREER HUB

3. Настроенная функция вывода

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

Пример вызова:

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

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

Пример вывода:

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

Решение:

```python
def custom_printer(sep, end):
    def print_func(*args):
        print(*args, sep=sep, end=end)
    return print_func

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

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

IT CAREER HUB

4. 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:

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

Пример вывода:

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

Решение:

```python
def call_counter(func):
    count = 0

    def wrapper():
        nonlocal count
        count += 1
        print(f"Вызов функции '{func.__name__}' №{count}:")
        func()

    return wrapper

@call_counter
def greet():
    print("Привет!")

greet()
greet()
greet()
```