

Замыкания и декораторы

Python

Области видимости

Глобальный контекст:

```
value = 'in global scope'

def show_value():
    print('value:', value)

def hide_value():
    value = '***'
    print('value:', value)

show_value() # value: in global scope
hide_value() # value: ***
```

Локальный контекст:

```
def some_function():
    value = 'some'
    print(value)

def other_function():
    value = 'other'
    print(value)

def invalid_function():
    print(value) # Exception!
```

global и nonlocal

```
n = 10
```

```
def set_20():  
    # n in local scope  
    n = 20
```

```
def set_30():  
    # n in global scope  
    global n  
    n = 30
```

```
set_20()  
print(n)    # 10  
set_30()  
print(n)    # 20
```

Ключевое слово **global** используется для того, чтобы переменная искалась в глобальном контексте.

Ключевое слово **nonlocal** прикрепляет идентификатор к переменной из ближайшего окружения (кроме глобального).

Пример использования `nonlocal` посмотрим позже...

Функции могут вернуть функции

```
def some_function():  
    def other_function():  
        print('from other function')  
  
    return other_function  
  
f = some_function()  
f() # from other function
```

Функция `other_function` определена внутри функции `some_function` и возвращается из неё при вызове. Поэтому после присвоения переменной `f` значения его типом будет функция.

Захват локальных переменных

```
c = 30

def sum_maker(a):
    b = 10

    def closur():
        # closur lock a and b from local scope
        return a + b + c

    return closur

f = sum_maker(a=20)
f2 = sum_maker(a=0)
print(f(), f2()) # 60 40
c = 5
print(f(), f2()) # 35 15
```


Счётчик

Будет ли работать код ниже?

```
def counter(start=0):  
    value = start - 1
```

```
    def increase():  
        value += 1  
        return value
```

```
    return increase
```

```
cnt = counter()  
print(cnt())  
print(cnt())  
print(cnt())
```

Этот код вызовет следующее исключение:

UnboundLocalError: local variable 'value' referenced before assignment

Дело в том, что при изменении значения переменной `value` интерпретатор не может найти её значение в локальном контексте, о чём и сообщает в виде исключения.

Исправленный счётчик

Чтобы исправить предыдущий пример нужно явно указать, в каком контексте мы хотим искать `value`. Для этого подойдёт ключевое слово `nonlocal`.

```
def counter(start=0):  
    value = start - 1  
  
    def increase():  
        nonlocal value  
        value += 1  
        return value  
  
    return increase  
  
cnt = counter()  
print(cnt()) # 0  
print(cnt()) # 1  
print(cnt()) # 2
```

Функции высших порядков

Функции высших порядков — это функции, которые могут принимать в качестве аргументов и возвращать другие функции.

```
def apply(a, f):  
    return f(a)  
  
print(apply([1, 3, 2, 8, 4], max)) # 8  
print(apply('test', str.upper)) # TEST
```


Функции-обёртки

```
def wrap(f):  
    def wrapper():  
        print('it is wrapper: ', end='')  
        f()  
  
    return wrapper
```

```
def test_function():  
    print('just test')
```

```
test_function = wrap(test_function)  
test_function()  
# output:  
# it is wrapper: just test
```

Удастся ли обернуть следующую функцию?

```
def other_function(a, b):  
    print(a, b)
```

Почему?

Универсальные обёртки

```
def wrap(f):  
    def wrapper(*args, **kwargs):  
        result = f(*args, **kwargs)  
        print('result in wrapper:', result)  
        return result
```

```
    return wrapper
```

```
def upper(s):  
    return s.upper()
```

```
def power(a, b):  
    return a ** b
```

```
upper = wrap(upper)  
power = wrap(power)
```

```
print(upper('python'))  
print(power(2, 10))
```

Вывод:
result in wrapper: PYTHON
PYTHON
result in wrapper: 1024
1024

Подмена значений

```
def wrap(f):  
    def wrapper(*args, **kwargs):  
        result = f('wrapper')  
        return result  
  
    return wrapper  
  
def test_function(s):  
    print('test:', s)  
    return s + '!'  
  
test_function = wrap(test_function)  
out = test_function('python') # test: wrapper  
print(out) # wrapper!
```

Декоратор

```
def upper_wrap(f):  
    def wrap(*args, **kwargs):  
        result = f(*args, *kwargs)  
        if isinstance(result, str):  
            return result.upper()  
        return result  
  
    return wrap
```

```
def goodbye(name):  
    return f'Goodbye, {name}!'
```

```
goodbye = upper_wrap(goodbye)  
print(goodbye('deponia'))  
# GOODBYE, DEPONIA!
```

same
→

```
def upper_wrap(f):  
    def wrap(*args, **kwargs):  
        result = f(*args, *kwargs)  
        if isinstance(result, str):  
            return result.upper()  
        return result  
  
    return wrap
```

```
@upper_wrap  
def goodbye(name):  
    return f'Goodbye, {name}!'
```

```
print(goodbye('deponia'))  
# GOODBYE, DEPONIA!
```

Декоратор с аргументами

```
def multiply_decorator(repeats=2):  
    def decorator(f):  
        def wrapper(*args, **kwargs):  
            result = f(*args, **kwargs)  
            return result * repeats  
        return wrapper  
    return decorator  
  
@multiply_decorator(repeats=2)  
def hello():  
    return 'Hello!'  
  
@multiply_decorator(repeats=10)  
def multiply(a, b):  
    return a * b  
  
print(hello()) # Hello!Hello!  
print(multiply(2, 2)) # 40
```