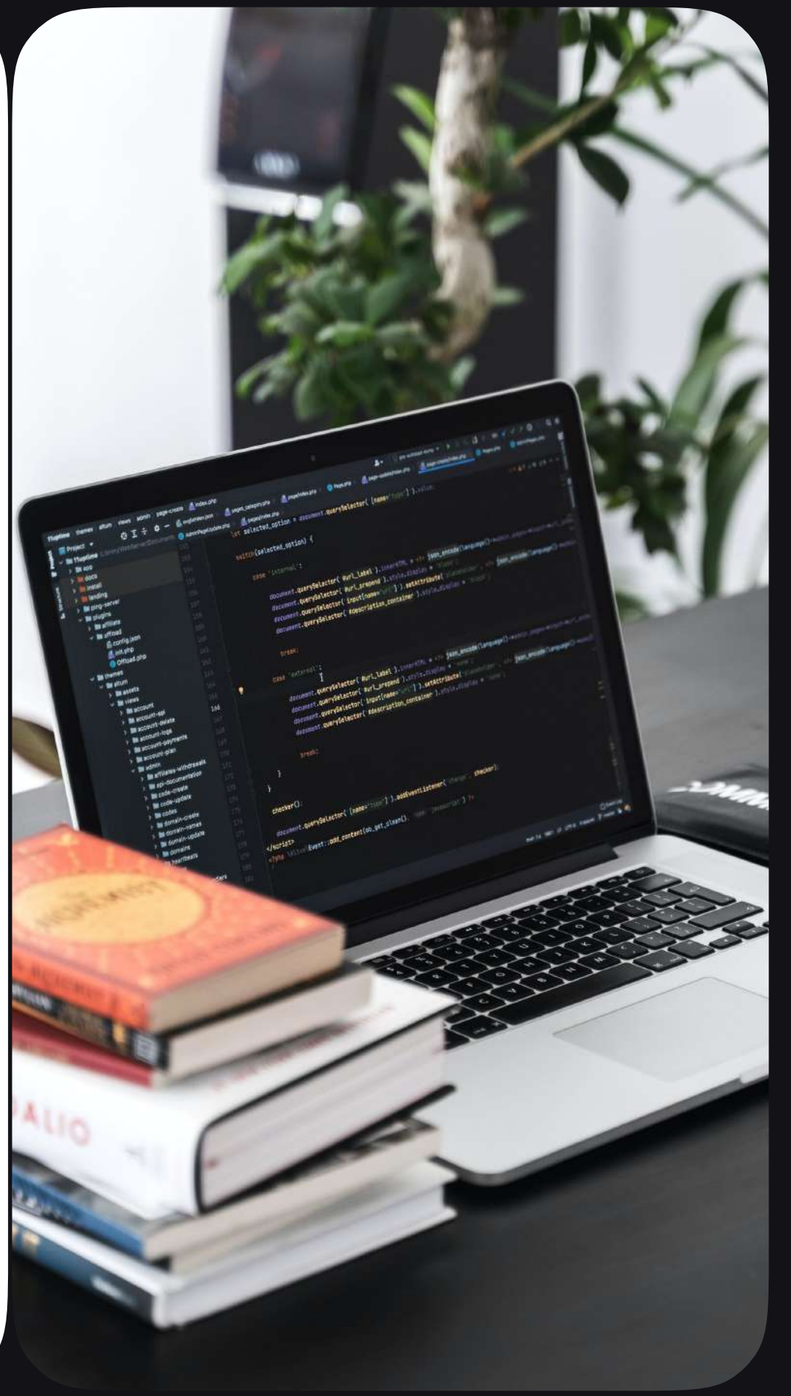
Модуль 3 Занятие 2

Функции. Декораторы



Области видимости

Замыкания

Декораторы

Итераторы

Итерируемые объекты

Генераторы

Локальные переменные

```
def get_sum( a, b, c ):
    total = a + b + c
    print(total)


get_sum(1, 2, 3)
print(total)
```

Параметры (локальные переменные)

Локальная переменная

Ошибка:

```
6
NameError: name 'total' is not defined
```

Локальные переменные — переменные, объявленные внутри функции.
Код за пределами функции не имеет доступа к локальным переменным.

Параметрические переменные — тоже локальные переменные.

Глобальные переменные

```
def get_sum():
    total = a + b + c
    print(total)


total = 0
a, b, c = 1, 2, 3
get_sum()
print(total)
```

Создается локальная переменная

Глобальная переменная

Глобальные переменные

Вывод:

```
6
0
```

Глобальные переменные — переменные, объявленные в основной программе, вне функции, и доступные как в самой программе, так и в ее функциях.

Если в функции создается локальная переменная с тем же именем, что у одной из глобальных, то обращение будет происходить к локальной, а не глобальной переменной.

Глобальные переменные

```
def get_sum():
    global total
    total = a + b + c
    print(total)

total = 0
a, b, c = 1, 2, 3
get_sum()
print(total)
```

Изменение глобальной переменной

Глобальная переменная

Глобальные переменные

Если необходимо внутри функции изменять значение глобальной переменной, то необходимо воспользоваться ключевым словом global.

Использование глобальных переменных — плохая практика. Почему?

Вывод:

```
6
6
```

Пространства имен

Пространство имен (namespaces) — это набор определенных в настоящее время имен и объектов. Можно сказать что пространстве имен это словарь, в котором ключи — это имена объектов, а значения — сами объекты.

В программе Python существует четыре типа пространств имен:

- **built-in** (встроенное пространство имен)
- **global** (глобальное пространство имен)
- **enclosing** (объемлющее пространство имен)
- **local** (локальное пространство имен)

Встроенное пространство имен

Встроенное пространство имен (built-in namespace) содержит имена всех встроенных объектов Python. Интерпретатор создает встроенное пространство имен при запуске, и это пространство имен существует до завершения работы интерпретатора. Получить список имен встроенного пространства имен можно с помощью специальной переменной __builtins__:

Например:

```
print(dir(__builtins__))
```

Во встроенном пространстве имен мы увидим уже знакомые нам объекты, например, исключения, встроенные функции, типы и так далее.

Локальное и глобальное пространство имен

```
def func(a):
    b = 4
    c = 5
    print(locals())


x = 1
y = 2
func(3)
print(globals())
```

Получить список имен локального и глобального пространств имен можно с помощью функций globals() и locals().

Ранее мы сказали, что можно рассматривать пространства имен как словарь и действительно, локальное и глобальное пространство имен реализуются как словари.

Объемлющее пространство имен

```
x = 'global'

def f():
    y = 'enclosing'

    def g():
        z = 'local'
        print(x, y, z)

    g()

f()
```

При каждом выполнении функции интерпретатор создает новое пространство имен. Это пространство имен является локальным для функции и существует до тех пор, пока функция не завершится.

Но что будет если определить одну функцию внутри другой?

Пространство имен, созданное для функции f(), является объемлющим пространством имен для вложенной функции g().

Вывод:

```
global enclosing local
```

Объемлющее пространство имен

```
x = 'global'


def f():
    x = 'enclosing'

    def g():
        nonlocal x
        x = 'change enclosing'
        # print(x)

    g()
    print(x)


f()
```

Если необходимо внутри вложенной функции изменять значение переменной из объемлющего пространства имен, то необходимо воспользоваться ключевым словом nonlocal.

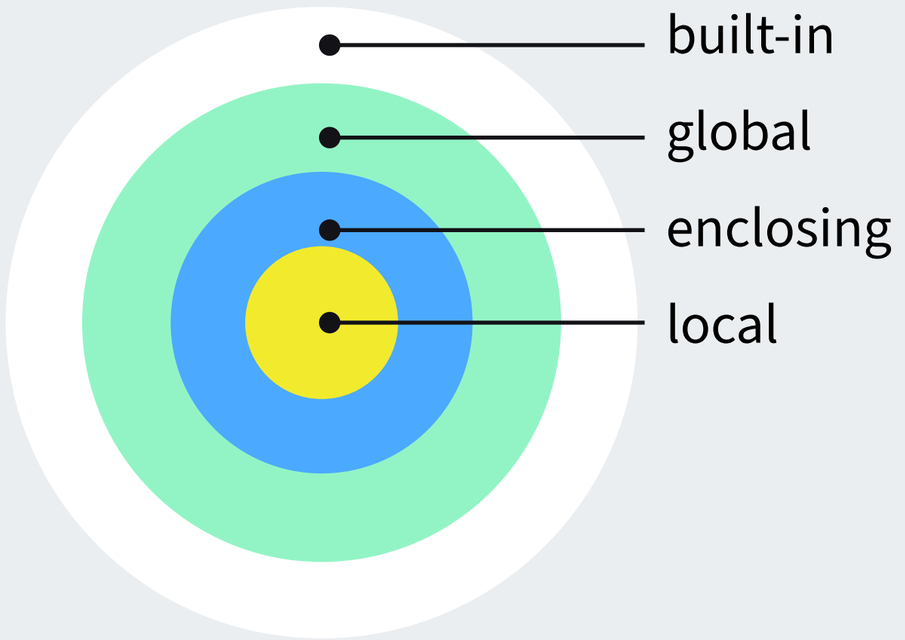
Вывод:

```
global enclosing local
```

Области видимости

Существование нескольких отдельных пространств имен означает, что несколько имен объектов могут существовать одновременно во время выполнения программы Python, и пока каждый объект находится в своем пространстве имен, они не будут мешать друг другу. Таким образом, если код ссылается на имя объекта, то Python ищет это имя в следующих пространствах имен в указанном порядке:

1. local (локальное пространство имен)
2. enclosing (объемлющее пространство имен)
3. global (глобальное пространство имен)
4. built-in (встроенное пространство имен)



Если интерпретатор не находит имя ни в одном пространстве имен, то возникает исключение NameError.

Замыкания

Рассмотрим пример:

```
def hello():
    message = 'Привет'

    def func():
        print(message)

    func()

hello()
```

Вывод:

```
Привет
```

Замыкание (closure) в программировании — это вложенная функция, которая ссылается на одну или более переменных из объемлющей (enclosing) области видимости.

Замыкания позволяют вложенной функции запоминать локальное состояние объемлющей области видимости.

Замыкания

```
def hello():
    message = 'Привет'

    def func():
        print(message)

    return func


f = hello()
f()
```

Вместо вызова функции внутри функции hello будем возвращать функцию func.
Таким образом функция hello возвращает замыкание. Присвоим результат переменной f и так как это функция, то ее можно вызвать.

Вывод:

```
Привет
```

Пример

Замыкание:

```
def mul(x):
    def func(y):
        return x * y

    return func


mul5 = mul(5)
print(mul5(3))
print(mul5(5))
print(mul5(9))
```

Вывод:

```
15
25
45
```

ООП:

```
class Mul:
    def __init__(self, a):
        self.a = a

    def __call__(self, b):
        return self.a * b

mul5 = Mul(5)
print(mul5(3))
print(mul5(5))
print(mul5(9))
```

Вывод:

```
15
25
45
```

Замыкания позволяют избежать использования глобальных переменных и предлагают некоторую форму инкапсуляции данных без использования ООП.

Декораторы

```
def decorator_func(func):
    def wrapper():
        print('Начало')
        func()
        print('Конец')

    return wrapper


def test():
    print('Тест')


test = decorator_func(test)
test()
```

Декоратор — это функция обертка, которая позволяет изменить поведение другой, декорируемой функции, без изменения ее кода.

Вывод:

```
Начало
Тест
Конец
```

Декораторы

```
def decorator_func(func):
    def wrapper(*args, **kwargs):
        result = func(*args, *kwargs)
        return result.upper()

    return wrapper



@decorator_func
def test(message):
    return message

print(test('тест'))
```

Вывод:

```
ТЕСТ
```

Декораторы могут не только менять поведение функции, но и менять значение, возвращаемое этой функцией.

Python предлагает упрощенный синтаксис для работы с декораторами: для этого перед объявлением декорируемой функции ставится @ и имя функции декоратора.

Пример

```python
import time


def time_of_func(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, *kwargs)
        stop_time = time.time()
        print(f'Время работы функции: {stop_time - start_time}')
        return result

    return wrapper


@time_of_func
def test():
    total = 0
    for i in range(10 ** 7):
        total += i
    return total


print(test())
```

Итераторы, итерируемые объекты, генераторы

Рассмотрим пример:

```
result = map(lambda x: x ** 2, range(1, 5))
print(result)
for elem in result:
    print(elem)
```

Вывод:

```
<map object at 0x0000023FFF6B7EB0>
1
4
9
16
25
```

Что же на самом деле возвращают функции map(), filter() и другие?

В Python существуют понятия итерируемый объект (iterable), итератор (iterator) и генератор (generator) — все это разные объекты.

От итерируемого объекта можно получить итератор, а генератор является разновидностью итератора.

Итерируемые объекты

Списки, словари, строки и коллекции — примеры итерируемых объектов в Python.

Итерируемый объект — это объект, который можно проитерировать, например, пройтись по элементам объекта в цикле for. Для этого итерируемый объект должен иметь метод __iter__(), который возвращает объект-итератор. Вызвать метод __iter__() можно с помощью встроенной функции iter().

```
a = [1, 2, 3, 4, 5]
b = iter(a)
print(b)
```

Вывод:

```
<list_iterator object at 0x000002A4C1346290>
```

Итераторы

Итератор, в свою очередь, должен реализовывать метод __next__(), который извлекает из итератора очередной элемент (у итерируемого объекта нет метода __next__(), а у итератора есть). Вызвать метод __next__() можно с помощью встроенной функции next(). Когда будут извлечены все элементы итератора, будет вызвано исключение StopIteration.

```
a = [1, 2, 3]
b = iter(a)
print(next(b))
print(next(b))
print(next(b))
print(next(b))
```

Вывод:

```
1
2
3
StopIteration
```

Как работает цикл for?

```
a = [1, 2, 3]
b = iter(a)
while True:
    try:
        print(next(b))
    except StopIteration:
        break
```

Генераторы

```
def generator():
    i = 1
    yield i
    i += 1
    yield i
    i += 1
    yield i
```

Генератор в Python — это функция, которая возвращает итератор. В функции-генераторе используется yield вместо return. Оператор yield возвращает значение и приостанавливает выполнение функции.

```
g = generator()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

Вывод:

```
<generator object generator at 0x0000025B3CD09A80>
1
2
3
StopIteration
```

Генераторы

Преимущества генераторов:

- простота реализации
- эффективность использования памяти (не нужно хранить всю последовательность, можно вычислять ее «на лету»)
- можно создавать бесконечные итераторы

Также генератор можно получить с помощью выражения генератора:

```
result = (i for i in range(10))
```

```
def fib(limit):
    f1, f2 = 1, 1
    while f1 < limit:
        yield f1
        f1, f2 = f2, f1 + f2


seq = fib(10)
for elem in seq:
    print(elem)
```

Вывод:

```
1
1
2
3
5
8
```

Итоги

- Локальные переменные — переменные, объявленные внутри функции. Глобальные переменные — переменные, объявленные в основной программе.
- Во время работы программы интерпретатор ищет имя объекта, просматривая локальную, объемлющую, глобальную и уже затем встроенную область видимости. Если имя не будет найдено, то вызывается исключение NameError.
- Замыкание — это вложенная функция, которая ссылается на одну или более переменных из объемлющей области видимости.
- Декоратор — это функция обертка, которая позволяет изменить поведение другой функции без изменения ее кода.
- Итерируемый объект — это объект, который можно проитерировать. Из итерируемого объекта можно получить итератор.
- Генератор — это функция, которая возвращает итератор.