Python

Лекция 2. Функции

Сайфулин Дмитрий, Слободкин Евгений
ИТМО, 20 ноября 2023

MASTER OF
SOFTWARE
ENGINEERING

Знакомство с функциями

Знакомимся

```python
def head(xs: list[int]) -> int | None:          # типовые аннотации опциональны
    """Возвращает первый элемент списка."""    # docstring
    if not xs:
        return xs
    return xs[0]
```

Знакомимся

```
def head(xs: list[int]) -> int | None:          # типовые аннотации опциональны
    """Возвращает первый элемент списка."""    # docstring
    if not xs:
        return xs
    return xs[0]
```

```
>>> head([])
None

>>> head([10, 20, 30])
10
```

Знакомимся

```
# Функция всегда что-либо возвращает
# Если возвращаемого значения явно не написано, то функция возвращает None
def head1(xs):
    for x in xs:
        return x
```

Знакомимся

```
# Функция всегда что-либо возвращает
# Если возвращаемого значения явно не написано, то функция возвращает None
def head1(xs):
    for x in xs:
        return x
```

```
>>> head1([])
None

>>> head1([10, 20, 30])
10
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1

>>> succ.__name__
'succ'
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1

>>> succ.__name__
'succ'

>>> succ.__doc__
'Возвращает следующее число.'
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1

>>> succ.__name__
'succ'

>>> succ.__doc__
'Возвращает следующее число.'

>>> succ.__annotations__
{'x': <class 'int'>, 'return': <class 'int'>}
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1

>>> succ.__name__
'succ'

>>> succ.__doc__
'Возвращает следующее число.'

>>> succ.__annotations__
{'x': <class 'int'>, 'return': <class 'int'>}

>>> dir(succ)
['__annotations__', … '__str__', '__subclasshook__']
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1

>>> succ.__name__
'succ'

>>> succ.__doc__
'Возвращает следующее число.'

>>> succ.__annotations__
{'x': <class 'int'>, 'return': <class 'int'>}

>>> dir(succ)
['__annotations__', … '__str__', '__subclasshook__']

>>> succ.attr = 42 # Можем добавлять любой атрибут к функции
>>> succ.attr
42
```

Функция является объектом

```
def succ(x: int) -> int:
    """Возвращает следующее число."""
    return x + 1

def pred(x: int) -> int:
    return x - 1

>>> succ.__code__ = pred.__code__  # 🤭
>>> succ(1)
0
```

Способы передачи аргументов

Именованные аргументы

```
def sub(a, b):
    return a - b


>>> sub(10, 3)
7
```

Именованные аргументы

```
def sub(a, b):
    return a - b


>>> sub(10, 3)
7

>>> sub(10, b=3)
7

>>> sub(b=3, a=10)  # sub(a=10, b=3) тоже ОК
7
```

Именованные аргументы

```
def sub(a, b):
    return a - b


>>> sub(10, 3)
7

>>> sub(10, b=3)
7

>>> sub(b=3, a=10)   # sub(a=10, b=3) тоже ОК
7

>>> sub(a=10, 3)  # Сначала должны идти позиционные аргументы
SyntaxError: positional argument follows keyword argument

>>> sub(3, a=10)
TypeError: sub() got multiple values for argument 'a'
```

Строго именованные/позиционные аргументы

```
# Аргументы до / можно вызывать только без имени, а после * — только по имени
def banner(text, /, border, *, width):
    return f" {text} ".center(width, border)
```

Строго именованные/позиционные аргументы

```
# Аргументы до / можно вызывать только без имени, а после * — только по имени
def banner(text, /, border, *, width):
    return f" {text} ".center(width, border)

>>> banner("Python", "~", 50)
'~~~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~~~'
```

Строго именованные/позиционные аргументы

```
# Аргументы до / можно вызывать только без имени, а после * — только по имени
def banner(text, /, border, *, width):
    return f" {text} ".center(width, border)

>>> banner("Python", "~", 50)
'~~~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~~~'

>>> banner("Python", "=", width=48)
'==================== Python ===================='
```

Строго именованные/позиционные аргументы

```
# Аргументы до / можно вызывать только без имени, а после * — только по имени
def banner(text, /, border, *, width):
    return f" {text} ".center(width, border)

>>> banner("Python", "~", 50)
'~~~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~~~'

>>> banner("Python", "=", width=48)
'================== Python =================='

>>> banner("Python", border="=", width=50)
'==================== Python ====================='
```

Строго именованные/позиционные аргументы

```
# Аргументы до / можно вызывать только без имени, а после * — только по имени
def banner(text, /, border, *, width):
    return f" {text} ".center(width, border)

>>> banner("Python", "~", 50)
'~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~'

>>> banner("Python", "=", width=48)
'================== Python =================='

>>> banner("Python", border="=", width=50)
'==================== Python ====================='

>>> banner(text="Python", border="=", width=50)
TypeError: banner() got some positional-only arguments …
```

Строго именованные/позиционные аргументы

```
# Аргументы до / можно вызывать только без имени, а после * — только по имени
def banner(text, /, border, *, width):
    return f" {text} ".center(width, border)

>>> banner("Python", "~", 50)
'~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~'

>>> banner("Python", "=", width=48)
'================== Python =================='

>>> banner("Python", border="=", width=50)
'==================== Python ====================='

>>> banner(text="Python", border="=", width=50)
TypeError: banner() got some positional-only arguments …

>>> banner("Python", "=", 48)
TypeError: banner() takes from 1 to 2 positional arguments but 3 were given
```

Аргументы по умолчанию

```
def banner(text, /, border="=", *, width=50):
    return f" {text} ".center(width, border)
```

Аргументы по умолчанию

```
def banner(text, /, border="=", *, width=50):
    return f" {text} ".center(width, border)
```

```
>>> banner("Python")
'====================== Python ======================'
```

Аргументы по умолчанию

```
def banner(text, /, border="=", *, width=50):
    return f" {text} ".center(width, border)


>>> banner("Python")
'====================== Python ======================'

>>> banner("Python", "~")
'~~~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~~~'

>>> banner("Python", width=48)
'====================== Python ======================'
```

Аргументы по умолчанию

```
def banner(text, /, border="=", *, width=50):
    return f" {text} ".center(width, border)


>>> banner("Python")
'====================== Python ======================'

>>> banner("Python", "~")
'~~~~~~~~~~~~~~~~~~~~~~ Python ~~~~~~~~~~~~~~~~~~~~~~'

>>> banner("Python", width=48)
'==================== Python ===================='

>>> banner.__defaults__ # Аргументы по умолчанию хранятся в поле объекта
('=',)

>>> banner.__kwdefaults__ # К чему такой подход может привести?
{'width': 50}
```

Изменяемые аргументы по умолчанию

```
def append_42(l=[]):
    l.append(42)
    return l
```

Изменяемые аргументы по умолчанию

```
def append_42(l=[]):
    l.append(42)
    return l
```

```
>>> append_42()
[42]
```

Изменяемые аргументы по умолчанию

```
def append_42(l=[]):
    l.append(42)
    return l


>>> append_42()
[42]

>>> append_42()  # 😬
[42, 42]
```

Изменяемые аргументы по умолчанию

```
# Будем всегда создавать новый список
def append_42(l=None):
    l = l or []
    l.append(42)
    return l
```

Изменяемые аргументы по умолчанию

```
# Будем всегда создавать новый список
def append_42(l=None):
    l = l or []
    l.append(42)
    return l
```

```
>>> append_42()
[42]

>>> append_42()  # 👍
[42]
```

*args и **kwargs

Функции с произвольным числом аргументов

```
def mult(*args):
    res = 1
    for arg in args:
        res *= arg
    return res
```

Функции с произвольным числом аргументов

```
def mult(*args):
    res = 1
    for arg in args:
        res *= arg
    return res
```

```
>>> mult()
1

>>> mult(2, 3, 4)
24
```

Функции с произвольным числом аргументов

```
# А если хотим как минимум один элемент принимать?
def mult(x, *args):
    res = x
    for arg in args:
        res *= arg
    return res
```

Функции с произвольным числом аргументов

```
# А если хотим как минимум один элемент принимать?
def mult(x, *args):
    res = x
    for arg in args:
        res *= arg
    return res
```

```
>>> mult()
TypeError: mult() missing 1 required positional argument: 'x'

>>> mult(2, 3, 4)
24
```

Функции с произвольным числом аргументов

```
# Функция принимает название команды, значения её параметров и запускает её
```

Функции с произвольным числом аргументов

```
# Функция принимает название команды, значения её параметров и запускает её
import command

def run(cmd_name, **kwargs):
    f = getattr(command, cmd_name)
    return f(**kwargs)
```

Функции с произвольным числом аргументов

```
# Функция принимает название команды, значения её параметров и запускает её
import command

def run(cmd_name, **kwargs):
    f = getattr(command, cmd_name)
    return f(**kwargs)

>>> run("reboot", worker_id=1, ignore_errors=True)
'[OK] Worker 1 successfully rebooted!'
```

Функции с произвольным числом аргументов

```
# Функция принимает название команды, значения её параметров и запускает её
import command

def run(cmd_name, **kwargs):
    f = getattr(command, cmd_name)
    return f(**kwargs)

>>> run("reboot", worker_id=1, ignore_errors=True)
'[OK] Worker 1 successfully rebooted!'

>>> run("create", label="python")
'[OK] Worker with label "python" has been created. Its ID is 1'
```

Что же такое *args и **kwargs?

```
def inner(*args, **kwargs):
    print("args:", args, "args type:", type(args))
    print("kwargs:", kwargs,"kwargs type:", type(kwargs))
```

Что же такое *args и **kwargs?

```
def inner(*args, **kwargs):
    print("args:", args, "args type:", type(args))
    print("kwargs:", kwargs,"kwargs type:", type(kwargs))
```

```
>>> inner()
args: () args type: <class 'tuple'>
kwargs: {} kwargs type: <class 'dict'>
```

Что же такое *args и **kwargs?

```
def inner(*args, **kwargs):
    print("args:", args, "args type:", type(args))
    print("kwargs:", kwargs,"kwargs type:", type(kwargs))
```

```
>>> inner()
args: () args type: <class 'tuple'>
kwargs: {} kwargs type: <class 'dict'>

>>> inner(42)
args: (42,) args type: <class 'tuple'>
kwargs: {} kwargs type: <class 'dict'>
```

Что же такое *args и **kwargs?

```
def inner(*args, **kwargs):
    print("args:", args, "args type:", type(args))
    print("kwargs:", kwargs,"kwargs type:", type(kwargs))
```

```
>>> inner()
args: () args type: <class 'tuple'>
kwargs: {} kwargs type: <class 'dict'>

>>> inner(42)
args: (42,) args type: <class 'tuple'>
kwargs: {} kwargs type: <class 'dict'>

>>> inner(42, b=43)
args: (42,) args type: <class 'tuple'>
kwargs: {'b': 43} kwargs type: <class 'dict'>
```

Немного про распаковку (*args)

```
def min(x, *args):
    m = x
    for arg in args:
        m = m if m < arg else arg
    return m


>>> l = [20, 42, 10, 30]
>>> min(l)
[20, 42, 10, 30]  # Кажется, это не то, что мы хотим
```

Немного про распаковку (*args)

```
def min(x, *args):
    m = x
    for arg in args:
        m = m if m < arg else arg
    return m
```

```
>>> l = [20, 42, 10, 30]
>>> min(l)
[20, 42, 10, 30]   # Кажется, это не то, что мы хотим

>>> min(*l)  # То же самое, что min(20, 42, 10, 30)
10
```

Немного про распаковку (*args)

```
def min(*args, default=None):
    if not args:
        return default
    m = args[0]
    for arg in args:
        m = m if m < arg else arg
    return m
```

```
>>> min()
None

>>> min(*[], 42)
42
```

Немного про распаковку (*args)

```
# * также можно пользоваться при разборе приходящего объекта

>>> head, *tail = [10, 20, 30]
>>> (head, tail)
(10, [20, 30])
```

Немного про распаковку (*args)

```
# * также можно пользоваться при разборе приходящего объекта

>>> head, *tail = [10, 20, 30]
>>> (head, tail)
(10, [20, 30])

>>> *_, last = [10, 20, 30]
>>> last
30
```

Немного про распаковку (*args)

```
# * также можно пользоваться при разборе приходящего объекта

>>> head, *tail = [10, 20, 30]
>>> (head, tail)
(10, [20, 30])

>>> *_, last = [10, 20, 30]
>>> last
30

>>> first, *_, last = [10, 20, 30]
>>> (first, last)
(10, 30)
```

Немного про распаковку (*args)

```
# * также можно пользоваться при разборе приходящего объекта

>>> head, *tail = [10, 20, 30]
>>> (head, tail)
(10, [20, 30])

>>> *_, last = [10, 20, 30]
>>> last
30

>>> first, *_, last = [10, 20, 30]
>>> (first, last)
(10, 30)

>>> e1, e2, e3 = [10, 20, 30]
>>> (e1, e2, e3)
(10, 20, 30)
```

Немного про распаковку (**kwargs)

```
# Хотим сделать фабрику, чтобы гибко могли настраивать значения атрибутов:
```

Немного про распаковку (**kwargs)

```
# Хотим сделать фабрику, чтобы гибко могли настраивать значения атрибутов:

>>> generate()
User(name='Вася', email='pupkin47@mail.ru', age=20)

>>> generate(name="Петя", email="petr@petrov.ru")
User(name='Петя', email='petr@petrov.ru', age=23)
```

Немного про распаковку (**kwargs)

```
# Хотим сделать фабрику, чтобы гибко могли настраивать значения атрибутов:

>>> generate()
User(name='Вася', email='pupkin47@mail.ru', age=20)

>>> generate(name="Петя", email="petr@petrov.ru")
User(name='Петя', email='petr@petrov.ru', age=23)

>>> invalid_user = generate(email="123456789")
>>> invalid_user
User(name='Вася', email="123456789", age=21)

>>> is_valid(invalid_user)
False
```

Немного про распаковку (**kwargs)

```
def generate(**fields):
    # Сгенерируем «случайного» пользователя
    # Конечно, имена и email-ы тоже можно генерировать
    d = {"name": "Вася", "email": "pupkin47@mail.ru", "age": randint(18, 100)}
    # Перекроем сгенерированные значения для полей, которые переданы в функцию
    return User(**{**d, **fields})
```

Немного про распаковку (**kwargs)

```
def generate(**fields):
    # Сгенерируем «случайного» пользователя
    # Конечно, имена и email-ы тоже можно генерировать
    d = {"name": "Вася", "email": "pupkin47@mail.ru", "age": randint(18, 100)}
    # Перекроем сгенерированные значения для полей, которые переданы в функцию
    return User(**{**d, **fields})
```

```
>>> generate()
User(name='Вася', email='pupkin47@mail.ru', age=20)

>>> generate(name="Петя", email="petr@petrov.ru")
User(name='Петя', email='petr@petrov.ru', age=23)

>>> invalid_user = generate(email="123456789")
>>> invalid_user
User(name='Вася', email="123456789", age=21)
```

Области видимости

Обращение к необъявленной переменной

```
# Переменная c пока нигде не объявлена
>>> def print_c():
...     print(c)

>>> print_c()
NameError: name 'c' is not defined
```

Обращение к необъявленной переменной

```
# Переменная c пока нигде не объявлена
>>> def print_c():
...     print(c)

>>> print_c()
NameError: name 'c' is not defined

>>> c = 42
>>> print_c()
42
```

Области видимости (до Python 3.12)

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing, замыкания. Чуть позже обсудим отдельно.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

# Часто это правило называют «LEGB-rule»
```

Области видимости (до Python 3.12)

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing , замыкания. Чуть позже обсудим отдельно.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

c = 1

def print_c():
    print(c)  # c берется из глобальной области видимости

print_c()  # 1
```

Области видимости (до Python 3.12)

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing , замыкания. Чуть позже обсудим отдельно.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

def min(x, y):
    return x

def f(x, y):
    def min(x, y):
        return y
    return min(x, y) # Используем min, объявленный внутри f

print(f(10, 20))  # 20
```

Области видимости (наши дни)[1]

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, локальная. Посмотреть можно с помощью `locals`.
# 2. Type Params, типовые параметры. Посмотреть можно с помощью __type_params__.
# 3. Closure/Enclosing , замыкания. Чуть позже обсудим отдельно.
# 4. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 5. Builtin, встроенные функции. Все функции из модуля `builtins`.

class A[X]:
    X = 1
    def method(self, x: X) -> X:
        print(X, type(X))

>>> A().method(42)
X <class 'typing.TypeVar'>

>>> A.method.__annotations__
{'x': 1, 'return': 1}
```


1. Что нового в новом Python?: https://www.youtube.com/watch?v=nTGVKIFvI3Q&ab_channel=Globus

nonlocal и global

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing , замыкания. Чуть позже обсудим отдельно.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

# Хотим изменить переменную вне локальной области видимости:

>>> value = 0

>>> def set_value(x):
...     value = x

>>> set_value(42)
>>> value  # 😔
0
```

nonlocal и global

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing , замыкания. Чуть позже обсудим отдельно.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

# Хотим изменить переменную вне локальной области видимости:

>>> value = 0

>>> def set_value(x):
...     global value # value из глобальной области видимости
...     value = x

>>> set_value(42)
>>> value  # 😃
42
```

nonlocal и global

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing , замыкания. Чуть позже обсудим отдельно.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

# А что если переменная объявлена в объемлющей функции?
# Тогда обратиться к ней можем с помощью nonlocal:
def f(x):
    value = 0
    def inner(x):
        nonlocal value
        value = x
    inner(x)
    return value

>>> f(42)
42
```

Замыкания

Замыкания

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing, замыкания. ~~Чуть позже~~ Сейчас обсудим.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

# Немного изменим прошлый пример:

def f(x):
    value = 42
    def inner(x):
        return value + x  # Используем value из функции f
    return inner(x)

>>> f(30)
72
```

Замыкания

```
# Переменная последовательно ищется в следующих областях:

# 1. Locals, Локальная. Посмотреть можно с помощью `locals`.
# 2. Closure/Enclosing, замыкания. ~~Чуть позже~~ Сейчас обсудим.
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# 4. Builtin, встроенные функции. Все функции из модуля `builtins`.

# Немного изменим прошлый пример:

def f(x):
    value = 42
    def inner(x):
        return value + x  # Используем value из функции f
    print("closure", inner.__closure__)
    return inner(x)

>>> f(30)
closure (<cell at 0x1027eef80: int object at 0x102f7a9c8>,)
72
```

Анонимные функции

Анонимные функции

```
>>> succ = lambda x: x + 1
>>> succ(41)
42
```

Анонимные функции

```
>>> succ = lambda x: x + 1
>>> succ(41)
42
```

```
>>> fst = lambda f, *_: f
>>> fst(1, 2, 3, 4)
1
```

Анонимные функции

```
>>> succ = lambda x: x + 1
>>> succ(41)
42


>>> fst = lambda f, *_: f
>>> fst(1, 2, 3, 4)
1


>>> users = [{"name": "Вася", "age": 18}, {"name": "Петя", "age": 15}]

>>> list(sorted(users, key=lambda user: user["age"]))
[{'name': 'Петя', 'age': 15}, {'name': 'Вася', 'age': 18}]
```

Классическая ошибка

```
printers = []
for i in range(10):
    printers.append(lambda: print(i))

for printer in printers:
    printer()
```

Классическая ошибка

```
printers = []
for i in range(10):
    printers.append(lambda: print(i))

for printer in printers:
    printer()

# 9
# 9
...
# 9  🤨
```

Классическая ошибка

```
printers = []
for i in range(10):
    printers.append(lambda: print(i))

for printer in printers:
    printer()

# 9
# 9
…
# 9  🤨


# Вам в домашней работе предстоит объяснить причины подобного поведения
# и предложить способ «починки» данного кода
```

globals и locals

globals

```
# Переменная последовательно ищется в следующих областях:
# ...
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# ...
```

globals

```
# Переменная последовательно ищется в следующих областях:
# ...
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# ...

# globals() возвращает словарь, представляющий собой глобальную таблицу имён:

x = 1
def foo():
    pass

print(globals())

# {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x1021e9350>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins'
(built-in)>, '__file__': 'main.py', '__cached__': None, 'x': 1, 'foo':
<function foo at 0x10218c5e0>}
```

globals

```
# Переменная последовательно ищется в следующих областях:
# ...
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# ...

# Попробуем что-либо поменять в globals():

x = 1
def foo():
    pass

globals()["foo"] = lambda: print(42)
foo()  # 42
```

globals

```
# Переменная последовательно ищется в следующих областях:
# ...
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# ...

# Попробуем что-либо поменять в globals():

x = 1
def foo():
    pass

globals()["foo"] = lambda: print(42)
foo()  # 42

globals()["bar"] = lambda: print("hello from bar")
bar()  # hello from bar
```

globals

```
# Переменная последовательно ищется в следующих областях:
# ...
# 3. Globals, глобальные переменные. Посмотреть можно с помощью `globals`.
# ...

# globals() возвращает список глобальных переменных для того модуля, где
# функция объявлена:
```

```
# mod1.py

x = 1
def foo():
    print(globals())
```

```
# mod2.py

from mod1 import foo

x = 2
foo()
# {..., "x": 1, "foo": <function foo at ...>}
```

locals

```
# Возвращает словарь локальных переменных:

def foo(x):
    v = 2
    w = x + v
    print(locals())
    return w


>>> foo(42)
{'x': 42, 'v': 2, 'w': 44}
44
```

locals

```
# Однако изменение словаря locals() не приведёт к ожидаемому эффекту:

def foo(x):
    locals()["x"] = 41
    return x


>>> foo(42)
42
```

locals

```
# Однако изменение словаря locals() не приведёт к ожидаемому эффекту:

def foo(x):
    locals()["x"] = 41
    return x


>>> foo(42)
42

# Скажем просто, что так происходит из-за оптимизацией CPython.
# Так как нам известны какие переменные будут в функции, то мы на этапе
# компиляции можем выделить слоты в памяти для локальных переменных.
# locals в этом случае будет возвращать некоторое представление
# над структурой, которая отвечает за работу с локальными переменными.

# Почитать подробнее можно по ссылке:
https://stackoverflow.com/questions/63388571/locals-defined-inside-python-function-do-not-work
```

locals

```
# Пройдёт ли такой assert прямо в коде модуля:

assert locals() == globals()
```

locals

```
# Пройдёт ли такой assert прямо в коде модуля:

assert locals() == globals()   #  Да! А почему?
```