Функции

Функции.

```
def say_hello_to(first_name, second_name):
    """ Function which says hello to the
    person.

    Arguments:
    first_name -- first name of the person
    second_name -- second name
    """
    print(f"Hello {first_name} {second_name}")

    # функция всегда возвращает значение,
    # даже если нет return -- в этом случае
    # она возвращает None
```

Вычисление функций

Где-то в интерпретаторе существует **стек вызовов**, в нем лежат **стек-фреймы**.

Стек-фрейм содержит необходимую информацию о вызове: что за функция, ссылки на переданные аргументы, где вызвали и пр.

При каждом вызове функции интерпретатор сначала вычисляет аргументы функции, затем кладет стек-фрейм на стек вызовов, только после этого начинает выполнение самой функции.

После исполнения функции и выхода из нее стек-фрейм удаляется со стека.

Вычисление функций

С помощью pythontutor убедитесь, что правильно понимаете как будет исполняться код ниже:

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

def evaluate_a():
    return 42

def sum(a, b):
    return a + b + c

c = 30
sum(evaluate_a(), factorial(10))
```

Разрешение имен.

На предыдущем слайде мы не объявляли переменную `c` в функции `sum`, но все работает корректно.

Это произошло, потому что в python разрешение имен ведется в четырех областях видимости:

1. локальной **(local)**,
2. объемлющих блоках **(enclosing block)** ,
3. глобальной **(global)**,
4. встроенной **(builtin)**.

Разрешение имен.

```
def f():
    print(i)

for i in range(3):
    f() #?
```

Разрешение имен.

```
def f():
    print(i)

for i in range(3):
    f()

# 0
# 1
# 2
```

Разрешение имен.

Можно также менять значения переменных, которые находятся вне локальной области видимости.

Для этого нужно объявить переменную с ключевым словом `nonlocal` или `global` (разница очевидна из названия).

```
min = 42
def f():
    global min
    min = 1
    return min

f()
```

Разрешение имен.

```
def cell(value=None):
    def get():
        return value

    def set(update):
        nonlocal value
        value = update

    return get, set

get, set = cell()

set(42)
get()
```

Функция как объект.

Функции, как и все в python, являются объектами.

```
print(say_hello_to)
# <function say_hello_to at 0x1110e3950>

print(type(say_hello_to))
# <class 'function'>

say_hello_to.__doc__
# Function which says hello to the person .
# ...

help(say_hello_to)
# --- // ---

say_hello_to.__name__
# 'say_hello_to'
```

Функция как объект.

В частности, функциям, объявленным пользователем, можно добавить новые атрибуты или изменить старые:

```
def example():
    return example.some_atr

example.some_atr = 'Ooh'

print(example())
# Ooh

example.some_atr = 5
print(example())
# 5
```

Функция как объект.

Вплоть до такого:

```
def example():
    return 42

def oh_no():
    return "Oh no! Thy brok thi snak"

example.__code__ = oh_no.__code__
print(example())
# Oh no! Thy brok thi snak
```

Аргументы функций

```
def just_print(a, b):
    print(f'a={a} b={b}')

# можно передавать аргументы как позициональные
just_print(1, 2)
# a=1 b=2

# аргументы при передаче можно именовать
just_print(b=10, a=5)
# a=5 b=10

# именованные аргументы должны идти после позициональных
sum(b=10, 10)
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Произвольное количество аргументов.

```
def sum(*args): # args -- просто имя, может быть любым
# def sum(*any_possible_name_for_tuple):
    res = 0

    # args -- это обычный tuple
    print(type(args))
    for arg in args:
        res += arg

    return res

print(sum(1, 2, 3, 10))
# <class 'tuple'>
# 16

print(sum())
# <class 'tuple'>
# 0
```

Произвольное количество аргументов.

Иногда требуется, чтобы функция принимала не менее N аргументов.

```
def max(first, *args):
    res = first
    for arg in args:
        res = res if res > arg else arg
    return res

max(1, 2, 3, 10)
# 10

max() # фактически требуем, чтобы был хотя бы один
TypeError: max() missing 1 required positional argument: 'first'
```

Произвольное количество аргументов.

```
def setup(**config):
    print("Initializing with config:")
    print(f"Type of config {type(config)}") # обычный dict
    print("Entries from config:")
    delimiter = ">" + '=' * 5 + "<"
    print(delimiter)
    for (i, (k, v)) in enumerate(config.items()):
        print(f"    Entry {i + 1:0>2}: {k} {v}")
    print(delimiter)
    # some really useful code
    # init(config)
    print("Finished initializing")
```

Произвольное количество аргументов.

```
setup(code='asdab3411', some_number=10, name=5)
# Initializing with config:
# Type of config <class 'dict'>
# Entries from config:
# >======<
#     Entry 01: code asdab3411
#     Entry 02: some_number 10
#     Entry 03: name 5
# >======<
# Finished initializing
```

Произвольное количество аргументов

```
def mix_up(*args, **kwargs):
    print(args, kwargs)

a = (1, 2, 3)
b = {'c': 10, 'e': 30}

# обратите внимание, что распаковывать контейнеры
# можно не только при создании списков и словарей, но
# и при вызове функций:
mix_up(*a, **b)
(1, 2, 3) {'c': 10, 'e': 30}

mix_up(b=10)
() {'b': 10}
```

Обязательные именованные аргументы

С помощью `*` можно указать, что все позициональные аргументы перечислены, остальные обязательно должны быть именованными

```
def example(arg1, *, kw_arg):
    ...

example(10, kw_arg=20)
example(arg1=10, kw_arg=20) # arg1 не обязательно должен быть позициональным

example(10, 20)
TypeError: example() takes 1 positional argument but 2 were given
```

Обязательные позициональные аргументы

С версии 3.8 в python появился дополпонительный синтаксис `/`, чтобы обознать обязательные позиональные аргументы.

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)

# валидный вызов
f(10, 20, 30, d=40, e=50, f=60)

f(10, b=20, c=30, d=40, e=50, f=60)   # b cannot be a keyword
argument
f(10, 20, 30, 40, 50, f=60)           # e must be a keyword
argument
```

Значения аргументов по-умолчанию

```
def load_dataset(filepath, type='csv'):
    with open(filepath) as file:
        return read_as(type, file)

load_dataset('input.csv')
# <DataFrame...>

load_dataset('intput.xls', type='excel')
# <DataFrame...>

# аргумент со значением по-умолчанию
# не обязательно должен быть именованным
load_dataset('intput.xls', 'excel')
# <DataFrame...>
```

Значения аргументов по-умолчанию

```
def load_dataset(filepath, *, type='csv'):
    with open(filepath) as file:
        return read_as(type, file)

load_dataset('input.csv')
# <DataFrame...>

load_dataset('intput.xls', type='excel')
# <DataFrame...>

load_dataset('intput.xls', 'excel')
 ---------
TypeError       Traceback (most recent call last)
----> 1 load_dataset('input.xls', 'excel')

TypeError: load_dataset() takes 1 positional argument but 2 were
given#
```

Значения аргументов по-умолчанию

Значения аргументов по-умолчанию вычисляются один раз при создании функции:

```
def broken_append(x=[]):
    ...

broken_append.__defaults__ ([],)
```

Значения аргументов по-умолчанию

```
def broken_append(x=[]):
    x.append(10)
    return x

broken_append.__defaults__
# ([],)

broken_append()
broken_append()

broken_append.__defaults__ #?
```

Значения аргументов по-умолчанию

```
def broken_append(x=[]):
    x.append(10)
    return x

broken_append.__defaults__
# ([],)

broken_append()
broken_append()

broken_append.__defaults__
# ([10, 10], )
```

Значения аргументов по-умолчанию

Если вынуждены использовать изменяемый объект в качестве значения по-умолчанию, лучше использовать `None`, а объект создавать при необходимости.

```
def normal_append(x=None):
    x = x or []
    x.append(10)
    return x

normal_append.__defaults__
# (None,)
normal_append()
# [10]
normal_append.__defaults__
# (None,)
```

globals

`globals` возвращает словарь, представляющий глобальную таблицу имен.

У каждого модуля своя глобальная таблица имен, поэтому `globals`, вызванный в функции или методе, возвращает словарь для модуля, где они объявлены.

globals

```
def foo():
    return globals()['bar'](1, 2, 3)

def bar(a, b, c):
    return globals()['i'] * a + b * c

i = 100

foo()
#?
```

globals

Словарь, возвращаемый `globals`, можно изменять. Это отразится на глобальной таблице имен:

```
def tommy_says():
    print("Hello!!")

def make_tommy_angry():
    globals()['tommy_says'] = lambda: \
        print("You b***s get out of my house")

tommy_says()
# Hello!!
make_tommy_angry()
tommy_says()
# ...
```

locals

Похожий словарь можно получить для локальной таблицы имен с помощью locals:

```
def print_locals(a, b):
    print(locals())
    c = a * b
    print(locals())

print_locals(2, 3)
# {'a': 2, 'b': 3}
# {'a': 2, 'b': 3, 'c': 6}
```

locals

Но для оптимизации python, на самом деле, на этапе компиляции статически просматривает и связывает переменные (поэтому их можно пронумеровать и расположить последовательно в памяти):

```
x = 10
def unbound():
    print(x) # будет искать в рантайме

def bound():
    print(x) # --\ x был связан с
    #            | переменной, объявленной ниже,
    x = 2    # <-/ при компиляции
```

locals

```
print(unbound())
# 10

print(bound())
# Traceback (most recent call last):
#  ...
#  File "run", line 2, in bound
#    print(x) # --\ x был связан с
# UnboundLocalError: local variable 'x' referenced before
assignment
```

locals

Поэтому `locals` возвращает не настоящую таблицу символов, а словарь, изменения в котором никак не отразятся на работе программы:

```
def heh_locals_unchangable():
    a = 50
    locals()['a'] = 'New Value of a!'
    print(locals())
    print(a)

heh_locals_unchangable()
#?
```

locals

Поэтому `locals` возвращает не настоящую таблицу символов, а словарь, изменения в котором никак не отразятся на работе программы:

```
def heh_locals_unchangable():
    a = 50
    locals()['a'] = 'New Value of a!'
    print(locals())
    print(a)

heh_locals_unchangable()
# {'a': 50}
# 50
```

closure

У функций есть атрибут

```
__closure__
```

Подумайте, что в нем могло бы находиться.

Придумайте примеры, для которых значение атрибута будет не `None` и в нем что-то будет содержаться.

Анонимные функции.

Вспомним анонимные функции.

```
lambda arguments: expression
```

и эквивалентны по поведению

```
def <lambda>(arguments):
    return expression
```

Всё, сказанное про именованные функции, справедливо и для анонимных.