Функции и замыкания

Функции обозначаются ключевым словом def. Естественно, это тоже отдельный объект с методом () (метод __call__())

```
>>> def fun(a, b):
...     c = a*2 + b
...     return c
...
>>> fun
<function fun at 0x7f5994b7a200>
>>> fun()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() missing 2 required positional arguments: 'a' and 'b'
>>> fun(123, 456)
702
```

В Питоне любая функция что-то возвращает. Если не явно, то возвращается объект None

```
>>> def nfun(a):
...     c = a * 2
...
>>> nfun(234)
>>> res = nfun(234)
>>> print(res)
None

>>> type(None)
<class 'NoneType'>
>>> not None
True
```

Утиная типизация и тут работает: функция - лишь алгоритм действий с операциями для объектов. Если у объектов есть соответствующие методы, всё будет работать

```
>>> fun(123, 456)
702
>>> fun("qwe", "qwerty")
'qweqweqwerty'
```

Глобальные и локальные переменные не перемешиваются. Даже с одинаковыми именами

```
>>> c = "QQ"
>>> c
'QQ'
>>> def fun(a, b):
...     c = a*2 + b
...     print(locals(), globals())
...     return c
...
>>> fun(100, 200)
{'a': 100, 'b': 200, 'c': 400} {'__name__': '__main__', '__doc__': None, '__package__':
None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 's': [3456, 'QQ', 1, 2,
3], 'fun': <function fun at 0x7f5994b7a5c0>, 'nfun': <function nfun at 0x7f5994b7a480>,
'res': None, 'c': 'QQ'}
400
```

В функции можно сделать и глобальную переменную

```
>>> def fun(a, b):
...     global c
...     c = a*2 + b
...     print(locals(), globals())
...     return c
...
>>> c
'QQ'
>>> fun(100, 200)
{'a': 100, 'b': 200} {'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 's': [3456, 'QQ', 1, 2,
3], 'fun': <function fun at 0x7f5994b7a2a0>, 'nfun': <function nfun at 0x7f5994b7a480>,
'res': None, 'c': 400}
400
>>> c
400
```

Новый прикол: Очевидно, в функции подтягиваются глобальные переменные

```
>>> def fun(a, b):
...     d = a + b + c
...     return d
...
>>> fun(1, 2)
403
>>> del c
>>> fun(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fun
NameError: name 'c' is not defined
```

А в следующем примере с непонятно, какая: глобальная или локальная. Поэтому это ошибочная ситуация

```
>>> c = 3
>>> def fun(a, b):
...     d = a + b + c
...     c = d
...     return d
...
>>> fun(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fun
UnboundLocalError: cannot access local variable 'c' where it is not associated with a value
>>>
```

Немножко приколов с функциями: можно возвращать что угодно, любые объекты, передавать также. Более того, можно внутри функции создавать другой объект-функцию и возвращать её

```
>>> def fufu(fun, arg):
...     return [fun(arg), fun(2*arg)]
...
>>> bin(1254)
'0b10011100110'
>>> fufu(bin, 32546)
['0b111111100100010', '0b1111111001000100']
```

```
>>> def doub(fun):
...     def newfun(arg):
...             return 2 * fun(arg)
...     return newfun
...
>>> doub(bin)(1254)
'0b100111001100b10011100110'
>>> doub(bin)
<function doub.<locals>.newfun at 0x7f5994b7a520>
```

Полезная функция для работы с последовательностями - sorted() - получают на вход последовательность, возвращают отсортированный по правилам список значений. Тоже имеет много параметров по умолчанию. В частности, полезная штука ***key***, в которой можно задать функцию-ключ, которая будет применяться к элементам последовательности, и сравниваться будут именно результаты

```
>>> sorted([1, 2, 3345, 65432, 6543, 12345, 76543, 2345])
[1, 2, 2345, 3345, 6543, 12345, 65432, 76543]
>>> sorted(range(100, 40, -3))
[43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79, 82, 85, 88, 91, 94, 97, 100]
```

```
>>> def keyfun(a):
...     return a % 5
...
>>> sorted(range(100, 40, -3), key = keyfun)
[100, 85, 70, 55, 91, 76, 61, 46, 97, 82, 67, 52, 88, 73, 58, 43, 94, 79, 64, 49]
```

Параметры в функциях делятся на три типа - Позиционные (задаются в строгом порядке их следования в параметрах функции, локальные имена не указываются), Именованные (задаются в любом порядке, обязательно указание локального имени) и смешанные
Позиции их в аргументах функции и разделители классов параметров приведены ниже

```
>>> # def (pos, /, pos+def+name, *, name)
```

```
>>> def fun(*args):
...     print(args)
...
>>> fun()
()
>>> fun(12345)
(12345,)
>>> fun(12345, 12345, 123456, "qwert", )
(12345, 12345, 123456, 'qwert')
```

Замыкание
Рассмотрим примерчик

```
>>> def f1(x):
...     def f2():
...             return x
...     return f2
...
>>> res = f1(100500)
>>> res()
100500
```

Получается Нелогичная дичь: локальное имя умерло, тогда откуда ссылка на 100500?
Парсер во время разбора функции f1 выясняет, что имя x не локальное и не глобальное - nonlocal
Так где же ссылка на значение?

```
>>> res = f1(1)
>>> res.__closure__[0]
<cell at 0x7f5994b4ebc0: int object at 0xa69e48>
>>> res.__closure__[0].cell_contents
1
```

Такая штука называется Позднее Связывание и\или Замыкание. Внутри объекта сохранён безыменованный адрес на объект, который далее при выполнении вызова подтягивается в функцию для исполнения

Ещё один прикол функций питона - самодокументация
Висящая текстовая строка в функции - это самодокументация этой самой функции. В модулях - самое то. По сути при работе функции эта строка, как объект, создаётся и сразу уничтожается. Но интерпретатор это отлавливает и сохраняет в память. Доступ к документации осуществляется специальным методом объекта функции

```
>>> def f(a: int, b: int) -> int:
...     "Documentation"
...     return a*2 + b
...
>>> f(3, 4)
10
>>> f.__doc__
'Documentation'
>>> help(f)
```

Последняя фича - Лямбда-функции. Это безыменованные функции, состоящие из выражения, результат которого сразу отправляется на выход функции. Очень удобно использовать лямбда-функции при задании, н-р, ключа сортировки

```
>>> res = lambda x: x % 5
>>> res
<function <lambda> at 0x7f20041231a0>
>>> res(10)
0
>>> res(17)
2


>>> arr = [i for i in range(25, 47, 3)]
>>> arr
[25, 28, 31, 34, 37, 40, 43, 46]
>>> arr = sorted(arr, key = lambda x: (x * 4) % 7)
>>> arr
[28, 37, 25, 46, 34, 43, 31, 40]
>>>
```