

Функции

- Именованный изолированный участок кода, принимающий параметры и возвращающий результат
- Все имена уничтожаются после выхода из функции
- Функция "не видит" имена, определенные в других функциях (кроме случая вложенных функций)
- `def` name(param1, param2, ...)
- `return` something – возвращает результат
- `return` - возвращает None

Функции: области видимости

```
1  def add(param1 , param2 ):
2      "returns sum of two objects "
3      result = param1 + param2
4      return result
5
6  print(add(2 , 3)) # 5
7  print(result) # error
8  print(add(2 , "3")) # error
9  print(add("2" , "3")) #"23"
10
11 help(add)
12
13 # Help on function add in module __main__:
14 #
15 #add(param1 , param2)
16 #     returns sum of two objects
17
18 print(add.__doc__) # returns sum of two objects
```

Функции: области видимости

- Все имена в функции при компиляции делатся на глобальные и локальные.
- Параметры и имена, которым где-либо производится присваивание, - локальные. Остальные - глобальные. На основании этого производится поиск во время исполнения.

```
1      def add_gv( val ):
2          return val + GLOBAL_VAR
3
4      add_gv(1) # error
5      GLOBAL_VAR = 12
6      add_gv(10) == 22
7
8      def add_gv_second( val ):
9          res = val + GLOBAL_VAR
10         GLOBAL_VAR = val
11         return res
12      add_gv_second(10) # error
```

Функции: глобальные переменные

```
1  def never_do_this ( value ):  
2      global SOME_GLOBAL_VAR  
3      SOME_GLOBAL_VAR = value  
4  
5  print (SOME_GLOBAL_VAR) # error  
6  never_do_this (1)  
7  print (SOME_GLOBAL_VAR) # 1
```

Функции: значения по умолчанию

```
1  def f1 (v1 , v2=0):
2      print (v1 , v2)
3
4  f1 (1) # 1 0
5  f1 (2 , 3) # 2 3
6
7  def f2 (a , b=12 , c ):
8      pass #error
9
10 def f3 (v1 , v2=[]):
11     v2.append (v1)
12     return v2
13
14 f3 (0) == [0]
15 f3 (1) == [0 , 1] # O_o
```

Функции: передача параметров

```
1  def f3(a, *args):
2      print(a, args)
3
4  print(f3(1)) # 1 ()
5  print(f3(1, 2)) # 1 (2,)
6  print(f3(1, 2, "abc")) # 1 (2, "abc")
7
8  def x(a, b):
9      return a - b
10
11 x(2, 1) == 1
12 x(b=2, a=1) == -1
13 x(1, a=1) # error
14 x(a=1, 2) # error
```

Функции: передача параметров

```
1  def x(a, **kwargs):
2      print(a, kwargs)
3
4  x(2) => 2, {}
5  x(2, 1) => error
6  x(a=1, b=2, c=3) => 1, {"c":3, "b":2}
7  x(1, fff=True) => 1, {"fff":True}
8
9  def x(a, b, *args, **kwargs):
10     pass
```

Функции: передача параметров

```
1  def sortwords(*wordlist , case_sensitive=False):  
2      print(wordlist , case_sensitive)  
3  
4  sortwords(1,2,3)  # >> (1, 2, 3) False  
5  
6  sortwords(1,2,3 , case_sensitive=True) # >> (1, 2, 3) True
```


Функции: передача параметров

```
1  def compare(a, b, *, key=None):
2      print(a, b, key)
3
4  compare(1, 2, 3)  # TypeError: compare() takes 2 positional
5                    # arguments but 3 were given
6  compare(1, 2) # >> 1 2 None
7  compare(1, 2, key=False)  # >> 1 2 False
```

Функции: передача параметров

```
1  def x(a, b):  
2      print(a, b)  
3  
4  params_tuple = (1, 2)  
5  x(*params_tuple) # 1 2  
6  
7  params_dict = {"a":44, "b":33}  
8  x(**params_dict) # 44 33  
9  
10 x(*[1], **{"b":2}) # 1 2
```

Аннотация

Аннотации - связывание данных с параметрами, никак не используется ядром языка, но см. туру

```
1     def x(a: int , b: 12):  
2         print(a, b)  
3     print(x.__annotation__) # {'a ': int , 'b ': 12}
```

Функции: рекурсия

```
1  def isin(lst , val):
2      for vl in lst:
3          if val == vl:
4              return True
5      return False
6
7  def isin_r(lst , val):
8      if len(lst) == 0:
9          return False
10     if val == lst[0]:
11         return True
12     return isin_r(lst[1:], val)
```

lambda

- Удобный синтаксис для однострочных функций
- `lambda` параметры : выражение
- Запрещены `print`, `for`, `if`,
- но можно `x if expr else y`, так как это выражение

```
1     var = lambda __params__: expr
2     def var(__params__):
3         return expr
4
5     mul2 = lambda x: x * 2
6     def mul2(x):
7         return x * 2
8
9     map((lambda x : x * 2), (1, 2)) == [2, 4]
```

ФП - Введение

- FP and OOP, FP vs Императивное
- Константные переменные
- Локализация побочных эффектов
- Мощная система типов
- Рекурсия
- ...
- Хороший компилятор

ФП - Введение

- Имя функции – переменная, указывающая на объект-функцию в памяти

```
1     def sum(x, y):
2         return x + y
3
4     print(sum) # <function sum at 0x..>
5     sum2 = sum
6     sum2(1, 2) == 3
7     sum2.func_name == 'sum'
8
9     sum = FunctionType(CodeType(...), globals(), 'sum', ...)
```

ФП - Введение

```
1  def map(func , iter ):
2      res = []
3      for i in iter :
4          res.append(func(i))
5      return res
6
7  def mul2(x):
8      return x * 2
9
10 map(mul2, (1, 2, 3)) == [2, 4, 6]
11 map(str, (1, 2, 3)) == ["1", "2", "3"]
```


ФП: Вложенные функции

- Новая функция создается каждый раз, когда python исполняет конструкцию `def`
- Функции могут бы вложенными, можно вернуть вложенную функцию
- Вложенная функция имеет доступ к аргументам функции, в которую она вложена

```
1     def top_func(x):  
2         def embedded_func(val):  
3             return val * 2  
4         return embedded_func
```

ФП:Замыкания

- При возврате вложенной функции, использующей переменные родительской, она сохраняет ссылки на используемые переменные и не дает им уничтожиться

```
1      def add_some(x):
2          def add_closure(val):
3              return val + x
4          return add_closure
5
6      add1 = add_some(1)
7      add5 = add_some(5)
8
9      add1 is add5 == False
10
11     add1(10) == 11
12     add5(10) == 15
13
14     map(add_some(10), (1, 2, 3)) == [11, 12, 13]
```

ФП:Каррирование

- Каррирование - связывание функции с частью параметров, с созданием новой функции

```
1     def bind_1st(func , val):
2         def closure(*vals , **params):
3             return func(val , *vals , **params)
4         return closure
5
6     def max(x, y):
7         if y > x:
8             return y
9         return x
10
11     not_less_than_10 = bind_1st(max, 10)
12     map(not_less_than_10 , (0 , 5 , 10 , 20))
13     # [10 , 10 , 10 , 20]
```

Генераторы/Сопроцедуры

- Генераторы это функции, которые могут сохранить свое состояние, вернуть значение в вызывающую функцию и позже продолжить исполнение с точки остановки
- Для приостановки исполнения используется ключевое слово `yield`
- Функции-генераторы могут использоваться в `for` циклах и везде, где принимается итератор
- `return` со значением запрещен

```
1     def my_generator( value ):
2         while value >= 0:
3             yield value ** 2
4             value -= 1
5
6     for val in my_generator(10):
7         print( val )
```

Генераторы/Сопроцедуры

- Функция содержащая `yield` при вызове возвращает `generator` при этом ни одна строка функции не исполняется, указатель исполнения стоит на первой строке функции
- `next(generator)` продолжает исполнение до следующего `yield` или конца функции
- `yield` приводит к приостановке исполнения и возврату значения
- `return` приводит к созданию исключения `StopIteration` которое автоматически обрабатывается циклом `for`
- Одновременно может существовать сколько угодно генераторов, созданных из одной функции. Все они будут иметь свой набор локальных переменных и указатель исполнения.

Генераторы/Сопроцедуры

```
1  def my_generator( value ):
2      print( "Enter with value =", value )
3      while value >= 0:
4          yield value ** 2
5          value -= 1
6
7  gen1 = my_generator(2)
8  gen2 = my_generator(200)
9  print( next( gen1 ) )
10 #Enter with value = 2
11 #4
12 print( next( gen1 ) )
13 #1
14 print( next( gen1 ) )
15 #0
16 print( next( gen1 ) ) # error
```

Генераторы/Сопроцедуры

- В генератор можно передать значение через `gen.send(val)` или ошибку через `gen.raise(error)` они будут переданны, как результат `yield`

```
1     def my_generator(value):
2         while True:
3             res = (yield value)
4             value = res ** 2
5
6     gen = my_generator(1)
7     print(next(gen) # 1
8     print(gen.send(10)) # 100
9     print(gen.send(10)) # 100
```

Генераторы/Сопроцедуры

- Бесконечные генераторы

```
1     def numbers ( val =0):  
2         while True:  
3             yield val  
4             val += 1
```


itertools

Декораторы

```
1  def func ( param ) :  
2      return param * 2
```

Декораторы

```
1  def func ( param ) :  
2      print ( "param =" , param )  
3      res = param * 2  
4      print ( "result =" , res )  
5      return res
```

Декораторы

```
1  def func ( param ) :  
2      log_params ( func , param )  
3      res = param * 2  
4      log_result ( func , res )  
5      return res
```

Декораторы

```
1  def logged(func):
2      def closure(*dt, **mp):
3          log_params(func, param)
4          res = func(param)
5          log_result(func, res)
6          return res
7      return closure
8
9  def func_(param):
10     return param * 2
11
12  func = logged(func_)
```

Декораторы

```
1     @logged
2     def func_(param):
3         return param * 2
```

Декораторы

- Синтаксический сахар для модификаторов функций
- @name или @name(params)

```
1      @decorator(x, y)
2      def func():
3          pass
4
5      # equal to
6      def func():
7          pass
8
9      func = decorator(x, y)(func)
```

Декораторы

```
1  import functools
2
3  def log_params(func):
4      @functools.wraps(func)
5      def closure(*dt, **mp):
6          res = func(*dt, **mp)
7          print("{}({} {}) = {}".format(
8              func.__name__, dt, mp, res))
9          return res
10     return closure
11
12 @log_params
13 def some_func(x, y):
14     return x + y
15
16 some_func(1, 2) # some_func((1, 2) {}) = 3
```


Применение декораторов

- логирование/трассирование
- проверка типов параметров
- проверка пост и предусловий
- открытие/закрытие транзакций
- освобождение/захват лока,
- взаимосвязь с with

Перегрузка функций

- 3.4 - singledispatch
- <https://github.com/mrocklin/multipledispatch/>
- <https://pypi.python.org/pypi/PEAK-Rules/0.5a1.dev>

АА

- Написать функцию `map` следующими способами: рекурсивным (`map_rq`), на генераторе (`map_yield`) и рекурсивно на генераторе (`map_rq_yield`). Функция `map` принимает два параметра - функцию с одним параметром и итерируемый объект. Возвращает итерируемый объект (список или генератор), который состоит из результатов применения параметра-функции к элементам итерируемого параметра с сохранением последовательности. `map(lambda x: x ** 2, [1,2,3]) == [1, 4, 9]`. Рекурсивные функции должны обрабатывать не более одного элемента на шаг рекурсии. Функции-генераторы должны обрабатывать элементы по мере запроса значений генератора.

АА

- Написать декоратор `time_me`, который делает профилирование функции, используя заданную функцию времени. Вторым параметром передается словарь, который нужно обновлять с каждым вызовом. В его элемент `'num_calls'` нужно заносить количество вызовов, в его элемент `'cum_time'` суммарное время.

```
1  import time
2
3  statistic = {}
4  @time_me(time.time, statistic)
5  def som_func(x, y):
6      time.sleep(1.1)
7
8  time_me(1, 2)
9  time_me(1, 2)
10
11  assert statistic['num_calls'] == 2
12  assert 2.5 > statistic['cum_time'] > 2
```

ДЗ 2.1

- Написать функцию `bind` который позволяет связать функцию-параметр с любыми заданными параметрами. По итогу должна получиться функция, которая принимает недостающие параметры и вызывает функцию-параметр. Корректность введенных параметров проверять не надо, просто вызвать исходную функцию со слитым набором.

```
1     def func(x, y, z, t):
2         return x, y, z, t
3
4     f1 = bind(func, 1, 2, t=13)
5     assert f1([4]) == (1, 2, [4], 13)
```

ДЗ 2.2

- Написать декоратор `me_haskell`, который позволяет функции вести себя как функция в `haskell`. Она принимает параметры и возвращает новые функции до тех пор, пока накопленных за все прошлые вызовы параметров не хватает для вызова. Как только их хватает для вызова - вызывается оригинальная функция и возвращается полученный результат. По мере накопления параметров делает проверка их корректности. Запрещается передавать один параметр два раза и передавать параметры, которые не подходят под спецификацию оригинальной функции. При неверных параметрах делать `raise ValueError(message)`. Для этого посмотреть на функцию `inspect.getargspec`. Для декорируемой функции должны быть запрещены `*` и `**` параметры, это нужно проверить сразу при декорировании.

```
1     @me_haskell
2     def func(x, y, z):
3         return x, y, z
4
5     f1 = func(1)
```

```
6      assert f1(2, 3) == (1, 2, 3)
7
8      f2 = f1(z = True)
9      assert f2("abc") == (1, "abc", True)
10
11     func(1, x=1) # should throw an exception
12     func(1, 2)(y=1) # should throw an exception
13     func(y=12)(1, 2) # should throw an exception (y defined twice)
14     assert func(y=12)(1, z=2) == (1, 12, 2)
```

ДЗ 3

- Написать декоратор `check_me`, который проверяет типа параметров функции. Ограничения на типы переменных задаются в докстринг. При неверном значении делает `raise TypeError(mess)` `mess` - строка, описывающая какой параметр имеет неверный тип и какой тип должен быть. Для этого посмотреть на функцию `inspect.getargspec`. Для декорируемой функции должны быть запрещены `*` и `**` параметры, это нужно проверить сразу при декорировании.

```
1      @check_me
2      def my_func(x, y):
3          """
4              @param x: int
5              @param y: str
6              """
7          return x + str(y)
8
9      my_func(1, 2) # TypeError
10     my_func(2, "3")# ok
```