



Функции

Технологии и языки программирования

Юдинцев В. В.

Кафедра теоретической механики
Самарский университет

20 октября 2017 г.

Содержание

- 1 Объявление функций
- 2 Переменное количество аргументов
- 3 Пространство имён
- 4 Задание

Объявление функций

Объявление функции

```
1 def mean(a):  
2     s = sum(a)  
3     return s/len(a)
```

- Ключевое слово `def`, имя функции, список аргументов
- Результат возвращается при помощи оператора `return`
- Если оператор `return` отсутствует, то функция возвращает `None`

```
print( 'Среднее значение ', mean([1,2,3]) )
```

Несколько аргументов

```
1 def kinetic_energy(m, v):  
2     T = m*v*v*0.5  
3     return T
```

```
1 print( """Кинетическая  
2 энергия тела с массой {} кг, движущегося  
3 со скоростью {} м/с, равна  
4 {} Дж """.format(10, 5, kinetic_energy(10,5) ) )
```

Кинетическая энергия тела с массой 10 кг, движущегося со скоростью 5 м/с, равна 125.0 Дж

Позиционные аргументы

Объявление функции:

```
1 def kinetic_energy(m, v):  
2     return 0.5*m*v*v
```

Вызов функции:

```
kinetic_energy(5, 10)
```

- Аргументы функции `kinetic_energy` должны передаваться в том порядке, в котором это задумано автором функции
- При вызове функции необходимо помнить смысловой порядок аргументов

Именованные аргументы

Объявление функции:

```
1 def kinetic_energy(m, v):  
2     return 0.5*m*v*v
```

Возможна передача функции именованных аргументов:

```
1 T = kinetic_energy(m = 5, v = 10)  
2  
3 T = kinetic_energy(v = 10, m = 5)
```

- Порядок аргументов может быть произвольным
- Такой способ вызова позволяет исключить ошибки

Параметры по умолчанию

```
1 def height_max(v0, g = 9.81):  
2     """  
3     Максимальная высота подъёма груза, брошенного вверх  
4     с начальной скоростью v0 (м/с),  
5     при ускорении свободного падения g (м/с**2)  
6     """  
7     return 0.5 * v0**2 / g
```

Высота подъёма груза на Земле, $g \approx 9.81 \text{ м/с}^2$

```
>> height_max(10)  
5.098399102681758
```

Высота подъёма груза на Луне, $g \approx 1.62 \text{ м/с}^2$

```
>> round(height_max(10, 1.62), 1)  
30.9
```


Параметры по умолчанию

При использовании **ССЫЛОЧНЫХ ТИПОВ** в параметрах по умолчанию их значение при изменении внутри функции сохраняется и до следующего вызова этой функции:

```
1 def add_element_to_b(a, b=[]):  
2     b.append(a)  
3     return b
```

При первом вызове функции: b – пустой список

```
add_element_to_b(4)  
[4]
```

При втором вызове функции: b “вспомнит” последнее значение

```
add_element_to_b(5)  
[4, 5]
```

Параметры по умолчанию

Исправленный вариант

```
1 def add_element_to_b(a, b=None):  
2     if b == None:  
3         b = []  
4     b.append(a)  
5     return b
```

```
>> add_element_to_b(4)  
[4]  
>> add_element_to_b(5)  
[5]
```

Пример

Функция вычисления корней квадратного уравнения

$$ax^2 + bx + c = 0$$

```
1 from math import sqrt
2
3 def solve(a, b, c):
4     d = b**2 - 4*a*c
5     if a == 0 & b == 0:
6         return None
7     if a == 0:
8         return -c/b
9     if d < 0:
10        return None
11    else:
12        return ( 0.5*(-b+sqrt(d))/a, 0.5*(-b-sqrt(d))/a )

print( solve(1, 10, 3) )
```

Задача

Матрица размерности 2×2 задана вложенным списком:

```
mat = [ [1, 2], [4, 5] ]
```

Напишите функцию **det**, вычисляющую (возвращающую) определитель любой матрицы размерности 2×2 .

Функция – объект первого класса

Объект первого класса – элемент, которые может быть передан как параметр, возвращён из функции, присвоен переменной.

```
1 get_length = len  
2 get_length( [1, 2, 3] )
```

Методы объектов

```
1 l = [1,2]  
2 x = [2,4]  
3  
4 add_to_l = l.append  
5 add_to_x = x.append  
6  
7 add_to_l (1)  
8 print ( l )  
9  
10 add_to_x (1)  
11 print ( x )
```

Замыкания

```
1 def step_function(x0, y0, y1):
2     def step(x):
3         if x < x0:
4             return y0
5         else:
6             return y1
7     return step
```

Вызов функции `step_function` создает объект функцию `step` с заданными значениями `x0`, `y0`, `y1`:

```
unit_step = step_function(0.0, 0.0, 1.0)
```

```
>> unit_step(-2)
0
```

```
>> unit_step(1)
1
```

- Функция (внутренняя), определяемая в теле другой (внешней) функции и создаваемая каждый раз во время выполнения внешней функции
- Внутренняя функция содержит ссылки на локальные переменные внешней функции

Переменное количество аргументов

Переменное число аргументов

Позиционные аргументы

Упаковка позиционных аргументов в кортеж `args`

```
1 def zeros (* args ) :  
2     print ( 'Аргументы: ', args )  
3     if len( args ) == 1 :  
4         return [0]* args [0]  
5     if len( args ) == 2 :  
6         return [[0]* args [0]]* args [1]
```

Переданные аргументы будут собраны в кортеж `args`

```
>> zeros (3,3)  
Аргументы: (3, 3)  
[[0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0]]
```

Переменное число аргументов

Позиционные аргументы

```
1 def zeros(n1, *args):  
2     if len(args) == 0:  
3         return [0]*n1  
4     if len(args) == 1:  
5         return [[0]*n1]*args[0]
```

```
>> zeros()  
TypeError: zeros() missing 1 required positional  
argument: 'n1'
```

Переменное число аргументов

Позиционные аргументы

```
1 def zeros(n1, *args):
2     if len(args) == 0:
3         return [0]*n1
4     if len(args) == 1:
5         return [[0]*n1]*args[0]
```

```
>> zeros()
TypeError: zeros() missing 1 required positional
argument: 'n1'
```

```
>> zeros(3)
[0, 0, 0]
```

Переменное число аргументов

Именованные аргументы

```
1 def zeros (**kwargs) :  
2     if kwargs.get('dim1') :  
3         res = [0]*kwargs['dim1']  
4     if kwargs.get('dim2') :  
5         res = [res]*kwargs['dim2']  
6     return res
```

Переданные аргументы будут собраны в словарь `kwargs`

```
>> zeros(dim1=3)  
[0, 0, 0]
```

```
>> zeros(dim1=2, dim2=2)  
[[0, 0], [0, 0]]
```

Распаковка позиционных аргументов

```
1 def zeros(n1, *args):  
2     row = [0]*n1  
3     if len(args) == 0:  
4         return row  
5     if len(args) == 1:  
6         return [row]*args[0]
```

Вызвать функцию `zeros` можно, передав два параметра

```
>> zeros(2,2)  
[[0, 0], [0, 0]]
```

или один `список` или `кортеж` из двух элементов с модификатором `*`

```
>> dim = (2,2)  
>> zeros(*dim)  
[[0, 0], [0, 0]]
```

Распаковка именованных аргументов

```
1 def zeros (** kwargs) :  
2     if kwargs.get('dim1') :  
3         res = [0]*kwargs['dim1']  
4     if kwargs.get('dim2') :  
5         res = [res]*kwargs['dim2']  
6     return res
```

Вызвать функцию `zeros` можно, передав два параметра

```
>> zeros(dim1=2,dim1=2)  
[[0, 0], [0, 0]]
```

или один **словарь** с модификатором `**`

```
>> dims = ('dim1': 2, 'dim2': 2)  
>> zeros(**dims)  
[[0, 0], [0, 0]]
```

Документирование функции

```
1 import numpy as np
2
3 def orbital_velocity ( height ) :
4     """ Скорость движения по круговой орбите вокруг Земли км/с
5     height - высота орбиты в км """
6     Rz = 6371
7     mu = 398600.4418
8     return np.sqrt (mu/(Rz+height))
```

```
>> help (orbital_velocity)
```

Help on function orbital_velocity in module __main__:

orbital_velocity(height)

Скорость движения по круговой орбите вокруг Земли км/с
height - высота орбиты в км

Пространство имён

Локальные переменные

При каждом вызове функции создаётся новое локальное пространство имён

```
1 a = 1 # Глобальная переменная модуля
2
3 def fun(x):
4     a = 2 # Локальная переменная
5     return x + a
```

```
>> fun(3)
```

```
5
```

```
>> a
```

```
1
```

Области видимости. Правило LEGB

При использовании имени переменной внутри функции интерпретатор последовательно ищет соответствующий этому имени объект в **локальной (L)** области, в **объемлющей (E)** области, в **глобальной (G)** области, а затем во **встроенной (B)**

```
1 a = 1 # Глобальная переменная модуля
2
3 def parent_fun(x):
4     b = 10
5     def fun(x):
6         return x + a + b
7     return fun(x)
```

```
>> parent_fun(7)
18
```

Инструкция `global`

```
1 a = 1 # Глобальная переменная модуля
2 def fun(x):
3     b = 10 + a # a из глобальной области
4     a = 1      # ОШИБКА!
5     return b
```

Local variable 'a' referenced before assignment

Изменить глобальную переменную можно, объявив её глобальной:

```
1 a = 1 # Глобальная переменная модуля
2 def fun(x):
3     global a
4     a = 3
5     b = 10 + a
6     return b
```

Инструкция `nonlocal`

При необходимости изменить переменную **объемлющей** области её необходимо объявить `nonlocal`

```
1 a = 1 # Глобальная переменная модуля
2 def fun(x):
3     a = 10 # Локальная переменная модуля функции fun
4     def inner_fun(x):
5         nonlocal a # Переменная из объемлющей области
6         a = a + 5 # Было 10, стало 15
7         b = a + x
8         return b
9     res = inner_fun(x)
10    return (a, res)
```

```
>> fun(1)
(15, 16)
```

Задание

Игра “Жизнь”

- Колония клеток заданна множеством пар координат (множество кортежей), например:

```
1 | colony = { (1,1), (2,1), (3,1), (3,2), (2,3) }
```

- Напишите функцию `count_neighbors(cell)`:
функция возвращает количество соседей у клетки с координатами `cell = (x, y)`
- Напишите функцию `get_colony_area(colony)`:
функция возвращает множество клеток, граничащих с колонией, включая клетки, занятые колонией
- Напишите функцию `next_generation(colony)`:
функция возвращает множество клеток следующего поколения, количество умерших и родившихся клеток

Игра “Жизнь”

- Колония клеток заданна множеством пар координат (множество кортежей), например:

```
1 | colony = { (1,1), (2,1), (3,1), (3,2), (2,3) }
```

- Напишите функцию

```
def game_life( born=(3, ), survives=(2,3) ) :  
    ...  
    return next_generation
```

возвращающую функцию типа `next_generation(colony)`, которую можно использовать для исследования эволюции колонии по различным правилам, задаваемым параметрами `born` и `survives`.