

Лекция 5

Функции, аргументы

Области видимости переменных

Функции: определения и вызовы

- *Функция* это конструкция, позволяющая выделить фрагмент программы для многократного использования
- Определение функции

```
def имя_функции(аргумент_1, аргумент_2 = умолчание):  
    блок_инструкций  
    return возвращаемое_значение
```

- Имя функции вводится инструкцией `def`, ее действие аналогично действию инструкции присваивания
- Имя функции это переменная связанная с объектом имеющим свойство "быть вызванным" (callable)
- Количество аргументов функции не ограничено
- Типы аргументов и возвращаемого значения заранее не определены
- Инструкция `return` завершает исполнение функции

Примеры определения и вызова

- Определение функции

```
def fun(a, b, c):  
    return (a + b) / c
```

- a, b и c будем называть *аргументами*
- аргументы эквивалентны локальным переменным функции

- Вызов функции

```
x = fun(1, 2, 3)  
print(x) # => 1.0
```

- 1, 2 и 3 будем называть *параметрами*
- передача параметров эквивалентна присваиванию параметров аргументам в порядке их записи:

```
a = 1  
b = 2  
c = 3
```

Локальные переменные

- Переменная созданная при исполнении блока функции становится *локальной переменной* для этой функции
 - локальная переменная видна только в пределах блока функции
 - локальная переменная видна во всем блоке функции даже если она была создана не в начале блока
 - по завершении блока функции локальная переменная исчезает, а связанный с ней объект будет уничтожен, если он не имеет других связей с программой
 - аргументы функции эквивалентны ее локальным переменным

Существует специальный случай, когда вместо создания локальной переменной используется уже существующая переменная. Этот случай будет рассмотрен ниже.

Значения по умолчанию

- Аргументы функции могут иметь значения по умолчанию
 - в определении функции значение по умолчанию записывается в виде
имя_аргумента=значение_по_умолчанию
 - аргументы имеющие значения по умолчанию должны следовать после всех аргументов значений по умолчанию не имеющих

```
def fun(a, b=2, c=3):  
    return (a + b) / c
```

- Вызов функции
 - При вызове функции передача параметров для аргументов имеющих значения по умолчанию необязательна

```
x = fun(1, 2, 3) # => 1.0  
x = fun(1, 2)   # => 1.0  
x = fun(1)      # => 1.0
```

Два способа передачи параметров

- Использование позиционных параметров это традиционный способ записи параметров: последовательно через запятую
- Именованные параметры передаются в функцию в виде пар *имя_параметра=значение_параметра*, при этом порядок следования именованных параметров значения не имеет

```
x = fun(b=2, c=3, a=1)
print(x) # => 1.0
```

- Можно использовать позиционные и именованные параметры в одном вызове, при этом сначала записываются все позиционные параметры, затем именованные

```
x = fun(1, c=3, b=2)
print(x) # => 1.0
```

- Параметр для данного аргумента можно передать только один раз

```
fun(1, c=3, b=2, a=1) # => Ошибка
```

Символ равно '='

- При *определении* функции символ равно '=' после имени аргумента функции это *значение по умолчанию* для данного аргумента

```
def fun(a=1, b=2, c=3):  
    return (a + b) / c
```

- При *вызове* функции символ равно '=' после имени аргумента это *значение параметра, передаваемое в функцию* через этот аргумент

```
x = fun(a=10, b=20, c=30)
```

- Оба случая в конечном итоге приводят к реальному исполнению инструкции присваивания
- *Аргумент функции это ее локальная переменная.* Передача параметра в функцию это присваивание параметра передаваемого в функцию этой локальной переменной.

Значения по умолчанию и именованные параметры

```
def polynom(x=0, c0=0, c1=0, c2=0, c3=0, c4=0):  
    return c0 + c1 * x**1 + c2 * x**2 + c3 * x**3 + c4 * x**4
```

```
polynom() # => 0  
polynom(c1=3, x=5) # => 15  
polynom(c2=2, c0=1, x=4) # => 33  
polynom(3, 2, 1) # => 5
```

- После именованного параметра нельзя использовать позиционный:

```
polynom(3, c4=2, 1) # Ошибка
```

- Пример функции обработки объекта:
 - первый аргумент без значения по умолчанию - объект
 - остальные аргументы с умолчаниями - настройка процесса обработки

```
def process_object(obj, fit=False, color='white', depth=8):
```


Инструкция return

- Параметром инструкции return является *выражение*, результат вычисления которого станет возвращаемым значением
- Инструкция return без параметра возвращает None
- Возврат из функции не имеющей инструкции return происходит после выполнения последней инструкции ее блока, такая функция также возвращает None
- Функция может содержать несколько инструкций return, первая *исполненная* инструкция return завершит функцию
- Тип возвращаемого значения, как и тип аргументов функции заранее не определен; функции в Питоне полиморфны по своей природе
- Пример:

```
def add3(a, b, c):  
    return a + b + c
```

- здесь *a*, *b* и *c* - любые совместимые последовательности

Функции позволяют

- Многократно использовать программный код и таким образом сократить его избыточность
 - не следует размножать код используя copy-and-paste, вместо этого нужно выделить его в функцию
 - если в функции будет обнаружена ошибка ее придется исправить только один раз
- Лучше структурировать код, делая его подобным тексту на естественном языке

```
файл_данных = открыть_файл_данных(имя_файла_данных)
выделенные_блоки = выделить_значимые_блоки(файл_данных)
if выделенные_блоки:
    обработать_выделенные_блоки(выделенные_блоки)
else:
    сообщить_об_отсутствии_выделенных_блоков()
```

- Уменьшить количество ошибок и упростить сопровождение кода

Выделение фрагмента кода в функцию

- Фрагмент текста программы обладает следующими характеристиками:
 - количество мест в программе, где данный фрагмент должен быть использован
 - количество связей с программой, то есть количество переменных влияющих на исполнение фрагмента и количество переменных, которые фрагмент изменяет сам
 - размер фрагмента в строках кода
 - осмысленность совершаемого фрагментом действия

Баланс этих характеристик служит критерием принятия решения о выделении фрагмента кода в функцию

Определение функции внутри функции

- Функция это объект, имя функции это переменная
- Определение функции это фактически инструкция присваивания объекта-функции переменной-имени
- Такого рода инструкция присваивания может быть исполнена внутри другой (объемлющей) функции. В этом случае имя созданной функции станет локальной переменной объемлющей функции.

```
def a():  
    i = 1  
    s = 'string'  
    print('Call a()')  
    def b():  
        j = 2  
        print('Call b()')  
    b()  
a() # => Call a() Call b()  
b() # => ошибка, имя функции b недоступно за пределами блока функции a
```

- Функция *a* имеет три локальных переменных: *i*, *s*, и *b*

Области видимости переменных

- Виды переменных по области их видимости
 - local: локальные, в текущей функции
 - enclosing (nonlocal): в объемлющей функции
 - global: переменные уровня модуля или глобальные
 - built-in: встроенные в интерпретатор
- Порядок разрешения имен:
local, enclosing, global, built-in (LEGB)
то есть поиск начинается в ближней зоне с уходом вдаль по мере необходимости
- Переменная создается в тот момент, когда она впервые появляется в левой части *исполняющейся* инструкции присваивания
- Область видимости в которой была создана переменная становится ее областью видимости

Переменные уровня модуля (глобальные)

- Глобальные переменные это переменные, созданные вне тела функции то есть на уровне модуля (файла программы)
- "Глобальные" переменные видны только внутри своего модуля, другие модули могут получить к ним доступ только явным использованием инструкции `import`
- Глобальными являются имена функций и классов, определенных в модуле если они определены не внутри других функций или классов
- Глобальные переменные как объекты содержащие данные используются редко, например как глобальный контекст исполнения программы

Категорически не рекомендуется использовать глобальные переменные в библиотеках подпрограмм или классов

Встроенные переменные

- Встроенные переменные находятся в модуле `builtins` и импортируются в программу автоматически так, как будто была выполнена инструкция

```
from builtins import *
```

- Имена встроенных функций `len`, `str`, `int` это встроенные переменные
- Пример: использование встроенной переменной `__name__`, содержащей имя модуля

```
def process_object(obj):  
    pass  
  
if __name__ == '__main__':  
    test_object = create_test_object()  
    process_object(test_object)
```

- Проверка значения `__name__` позволяет определить был ли модуль импортирован или он был запущен как программа

Категории встроенных функций

- **Функции создающие объекты встроенных типов**
 - `bool()`, `bytearray()`, `bytes()`, `complex()`, `dict()`, `float()`, `frozenset()`, `int()`, `list()`, `object()`, `set()`, `tuple()`
- **Функции для работы с числами**
 - `abs()`, `divmod()`, `max()`, `min()`, `ord()`, `pow()`, `round()`, `sum()`
- **Функции для работы с коллекциями**
 - `all()`, `any()`, `enumerate()`, `range()`, `filter()`, `iter()`, `next()`, `len()`, `map()`, `open()`, `reversed()`, `slice()`, `sorted()`, `zip()`
- **Функции для работы с объектами**
 - `callable()`, `delattr()`, `getattr()`, `hasattr()`, `hash()`, `id()`, `isinstance()`, `issubclass()`, `property()`, `setattr()`, `super()`, `type()`
- **Функции для преобразования объекта в строку**
 - `ascii()`, `repr()`, `str()`, `chr()`, `bin()`, `oct()`, `hex()`, `format()`
- **Функции ввода-вывода**
 - `input()`, `print()`

Категории встроенных функций

- Функции-декораторы
 - `classmethod()`, `staticmethod()`
- Функции доступа к внутренним механизмам интерпретатора
 - `breakpoint()`, `compile()`, `dir()`, `eval()`, `exec()`, `globals()`, `help()`, `locals()`, `memoryview()`, `vars()`, `__import__()`
- Полную документацию по встроенным функциям можно найти по ссылке:

<https://docs.python.org/3/library/functions.html>
- Встроенные функции полезно знать потому, что:
 - Многие часто встречающиеся простые задачи уже решены и реализованы в виде встроенных функций
 - Понимание встроенных функций обязательно понадобится при разборе чужих программ

Локальные переменные

- Переменная созданная внутри функции является локальной для данной функции, если она не была предварительно объявлена инструкцией `global` или `nonlocal`
- Время жизни переменной - от ее создания (появления в левой части инструкции присваивания) до уничтожения.
- Локальная переменная уничтожается при выходе из функции или явным образом инструкцией `del`
- Область видимости переменной - от начала функции до момента уничтожения переменной
- Локальная переменная связана с функцией а не с блоком

```
if a > 0:  
    b = 5  
print(b)
```

`b` напечатается при положительном `a`, при отрицательном `a` произойдет ошибка: переменная `b` использована до присваивания

Инструкция `global`

- Инструкция `global` позволяет объявлять глобальную переменную что бы иметь возможность изменять ее внутри блока функции

```
a = 10
```

```
def fun1():  
    print(a)
```

```
def fun2():  
    a = 11  
    print(a)
```

```
def fun3():  
    global a  
    a = 22  
    print(a)
```

```
fun1()    # => 10
```

```
fun2()    # => 11
```

```
print(a)  # => 10
```

```
fun3()    # => 22
```

```
print(a)  # => 22
```

Видимость и время жизни переменных

```
a = 10
```

```
def fun1():  
    print(a) # Error: переменная 'a' использована до присваивания  
    print(b) # Error: переменная 'b' не определена  
    a = 6 / 2  
    print(a) # => 3.0
```

```
def fun2():  
    global a # Переменная a объявлена как глобальная  
    print(a) # => 10  
    a = 6 * 2  
    print(a) # => 12
```

```
def fun3():  
    print(a) # => 12 # a - глобальная переменная  
    b = 8  
    print(b) # => 8  
    del b  
    print(b) # Error: переменная 'b' не определена
```

```
fun1() ; fun2() ; fun3()
```

Инструкция `nonlocal`

- Инструкция `nonlocal` позволяет получить доступ к переменной из областей видимости объемлющих функций

```
a = 10
```

```
def external_fun():  
    a = 22  
    def internal_fun():  
        nonlocal a  
        a = 33  
    print(a) # => 22  
    internal_fun()  
    print(a) # => 33
```

```
external_fun()  
print(a) # => 10
```

- Поиск `nonlocal` переменной происходит от более внутренних блоков функций к внешним и завершается как только переменная будет найдена

Вызов функции, примеры

```
def fun(a, b=2, c=3):  
    return (a + b) / c
```

```
print(fun(4, 5, 3)) # => 3.0. Вызов функции это элементарное выражение,  
fun(3, 5, 2)        # здесь возвращаемое значение 4.0 не использовано
```

- Параметром функции может быть изменяемый объект.
Объект *не копируется*, изменения происходят в самом объекте:

```
m = [1, 2]  
def swap_list(a):  
    t = a[0] ; a[0] = a[1] ; a[1] = t  
    swap_list(m) ; print(m) # => [2, 1]
```

- Возврат результата работы функции через изменяемый объект встречается редко, для этого более удобен кортеж в инструкции return:

```
def divide_with_check(a, b):  
    if b == 0:  
        return None, 'Error: divide by zero'  
    return a / b, 'OK'
```

Произвольное количество аргументов

- Для работы с произвольным количеством аргументов используются конструкции `*переменная` и `**переменная`

```
def fun(p1, p2, *pos_args, nam1, nam2, **nam_args):
    print(p1, p2, pos_args, nam1, nam2, nam_args)

fun(1, 2, 3, 4, 5, nam1='one', nam2='two', nam3='three', last='End')
# => 1 2 (3, 4, 5) one two {'nam3': 'three', 'last': 'End'}
fun(1, 2, 3, 4, 5, nam1='one', nam2='two', p1='three', last='End')
# => Ошибка, попытка один и тот же аргумент передать дважды
```

- Универсальная функция: любое, включая нулевое, количество позиционных и / или именованных аргументов

```
def fun2(*pos_args, **nam_args):
    for p in pos_args: process_positional_parameter(p)
    for n in nam_args: process_named_parameter(n, nam_args[n])
fun2()
fun2(1, 2)
fun2(1, color='white')
fun2(color='white', 1) # Ошибка: позиционный аргумент после именованного
```

Пример: функция dict()

- Синтаксис функции dict

```
dict(key1=value1, key2=value2)
```

стал возможен благодаря обработке произвольного количества именованных аргументов

```
def xdict(**nam_args):  
    return nam_args
```

- Более полная реализация

```
from copy import deepcopy  
def xdict(*pos_args, **nam_args):  
    if len(pos_args) == 1 and isinstance(pos_args[0], dict):  
        return deepcopy(pos_args[0])  
    return nam_args
```

```
xdict(color='white', brand='new', count=8)  
xdict({'a':1, 'b':2, 'last':'End'})
```


Пример: функция print()

```
def xprint(*pos_args, sep=' ', end='\n', file=sys.stdout, flush=False):
    for p in pos_args:
        file.write(str(p))
        file.write(str(sep))
    file.write(str(end))
    if flush:
        file.flush()
    return None
```

- Печать без разделителей

```
xprint('deer', 'copy', sep='') # => deercopy
```

```
# Печать в одну строку
```

```
xprint('a', 1, end=' ')
```

```
xprint('b', 2, end=' ')
```

```
xprint('c', 3) # => a 1 b 2 c 3
```

```
# Печать в файл
```

```
xprint('Sample\nSecond line\nEnd', file=open("test.txt", "w"))
```

Конструкции *переменная и **переменная при вызове функции

- Конструкции *переменная и **переменная при вызове функции позволяют разделить последовательность и / или словарь на отдельные элементы и передать эти элементы в функцию как параметры

```
def polynom(x=0, c0=0, c1=0, c2=0, c3=0, c4=0, debug=False):
    if debug:
        print('x =', x, 'c0 =', c0, 'c1 =', c1,
              'c2 =', c2, 'c3 =', c3, 'c4 =', c4)
    return c0 + c1 * x**1 + c2 * x**2 + c3 * x**3 + c4 * x**4
```

```
pos_args = [ 3, 5 ]
nam_args = { 'c4': 2, 'c3': 1 }
```

```
polynom(11, *pos_args, debug=True, **nam_args)
# => x = 11 c0 = 3 c1 = 5 c2 = 0 c3 = 1 c4 = 2
```

```
polynom(*pos_args, 11, **nam_args, debug=True) # Ошибки нет
```

Аннотации

- Аргументы функций могут быть снабжены аннотациями

```
def power(voltage: 'Напряжение в вольтах',  
         current: 'Ток в амперах',  
         phi: 'Сдвиг по фазе в радианах' = 0) -> 'Мощность в ваттах':  
    return voltage * current * math.cos(phi)
```

- Аргументы вместе с их аннотациями сохраняются в атрибуте `__annotations__` и могут быть использованы в коде программы

```
print(power.__annotations__)  
# => { 'voltage': 'Напряжение в вольтах', 'current': 'Ток в амперах',  
      'phi': 'Сдвиг по фазе', 'return': 'Мощность в ваттах' }
```

- Функция это объект и у нее есть атрибуты:

```
power.new_attribute = 12  
print(power.new_attribute) # => 12
```

Совпадение имен переменных

- Имена передаваемых в функцию переменных и имена аргументов функции находятся в разных областях видимости и не конфликтуют

```
def current(voltage, resistance):  
    i = voltage / resistance  
    return i
```

```
def calculate_current_mA():  
    voltage = 220  
    resistance = 10000  
    i = current(voltage=voltage, resistance=resistance)  
    return int(i * 1000)
```

```
print(calculate_current_mA()) # => 22
```

Заключительные замечания по аргументам функций

- Давайте аргументам осмысленные имена
- Задавайте умолчание всегда, когда оно имеет смысл
- Рекомендуются при вызове функции использовать именованные параметры даже для тех параметров, которые могут быть переданы как позиционные

```
def current(voltage, resistance):  
    return voltage / resistance
```

```
i = current(12, 100)  
i = current(voltage=12, resistance=100)
```

```
def power(voltage, current, phi=0):  
    return voltage * current * math.cos(phi)
```

```
p = power(voltage=220, current=0.01)  
p = power(voltage=220, current=0.01, phi=0.5)
```

Замыкания

```
def create_fun(r):
    resistance = r
    test_no = 1
    def current_by_voltage(voltage):
        nonlocal test_no
        this_test_no = test_no
        test_no = test_no + 1
        return this_test_no, voltage / resistance
    return current_by_voltage

f = create_fun(4)
g = create_fun(2)
print(f(12)) # => (1, 3.0) # (resistance == 4)
print(g(12)) # => (1, 6.0) # (resistance == 2)
print(f(20)) # => (2, 5.0) # (resistance == 4)
print(g(20)) # => (2, 10.0) # (resistance == 2)
```

- Каждая функция созданная вызовом `create_fun()` имеет свои собственные экземпляры переменных `resistance` и `test_no`. Эти переменные видны только внутри своих функций.

Замыкание как factory

- Конструкцию описанную выше иногда называют factory. На основе универсальной функции-шаблона создаются специализированные функции путем уточнения шаблона с помощью параметров.
- Каждая специализированная функция "замыкает" внутри себя свой собственный контекст исполнения.
 - Функция может менять контекст по ходу своей работы
 - Контекст исполнения сохраняется между вызовами функции, этим он подобен набору глобальных переменных
 - Контекст исполнения доступен только внутри своей функции, этим он подобен набору локальных переменных
- Сравнивая концепцию замыкания с концепциями объектно-ориентированного программирования можно считать, что
 - factory подобна классу,
 - специализированная функция подобна экземпляру класса.

Lambda-выражения

- lambda-выражение вводится ключевым словом `lambda`, результатом его вычисления является функция, которую неформально называют lambda-функцией
- После ключевого слова `lambda` следуют разделенные запятой аргументы за которыми следует двоеточие. За двоеточием следует тело функции:

```
f = lambda voltage, current, phi=0: voltage * current * math.cos(phi)
```
- Аргументы функции могут иметь значения по умолчанию, но не аннотации
- Тело lambda-функции это выражение, оно не может содержать инструкции
- lambda-функции могут использовать замыкания, для них эта технология является естественной
- lambda-функции также называют анонимными или безымянными функциями

Lambda-выражение в цикле

- Использование замыкания

```
def create_functions():  
    a = []  
    for i in range(1, 4):  
        a.append(lambda x: x * i)  
    return a  
  
for e in create_functions():  
    print(e(10), end=' ') # => 30 30 30
```

- Использование аргумента по умолчанию

```
def create_functions2():  
    a = []  
    for i in range(1, 4):  
        a.append(lambda x, y=i: x * y)  
    return a  
  
for e in create_functions2():  
    print(e(10), end=' ') # => 10 20 30
```

Lambda-выражение, пример использования

```
d = [ "Иван Петров", "Георгий Иванов", "Валерий Сидоров" ]  
print(d)  
# => ['Иван Петров', 'Георгий Иванов', 'Валерий Сидоров']  
  
print(sorted(d))  
# => ['Валерий Сидоров', 'Георгий Иванов', 'Иван Петров']  
  
print(sorted(d, key=lambda s: s.split()[1]))  
# => ['Георгий Иванов', 'Иван Петров', 'Валерий Сидоров']
```

- Lambda-выражение используется в тех случаях, когда:
 - нужна функция вызов которой записывается только один раз
 - сама функция достаточно проста
 - запись функции в виде lambda-выражения не ухудшает читаемость программы

Рекурсия

- Функция может вызывать сама себя, такой прием называют *рекурсией*

```
def plr(a):  
    if a:  
        e = a.pop()  
        print('before', e)  
        plr(a)  
        print('after ', e)
```

```
plr([ 1, 2, 3, 4 ]) # =>
```

```
before 4  
before 3  
before 2  
before 1  
after 1  
after 2  
after 3  
after 4
```

Рекурсия, продолжение

- Функция $a()$ может вызывать функцию $b()$, в то время как функция $b()$ вызывает функцию $a()$ - это *взаимная рекурсия*
- Рекурсия подобна циклу, как и цикл она может оказаться бесконечной. В отличие от цикла бесконечная рекурсия непрерывно потребляет ресурсы и программа аварийно завершается из-за отсутствия памяти.
- Алгоритм с рекурсией может быть преобразован в алгоритм с циклом и наоборот, критерий выбора - читаемость программы
- В общем случае алгоритмы с циклом более эффективны, чем алгоритмы с рекурсией
- Примеры рекурсивных алгоритмов: обход дерева, разбор выражения

Докажите что рекурсия вашего алгоритма конечна или ограничьте уровень рекурсии явно