Глава 5 Функции

Функции являются **абстракциями**, в которых детали реализации некоторого действия скрываются за отдельным именем. Хорошо написанный набор функций позволяет использовать их много раз. Стандартная библиотека Python содержит множество готовых и отлаженных функций, многие из которых достаточно универсальны, чтобы работать с широким спектром входных данных. Например, встроенные функции, математические функции модуля `math`, функции обработки строк, списков, словарей и т.п.

Даже если некоторый участок кода не используется несколько раз, но по входным и выходным данным он достаточно автономен, его смело можно выделить в отдельную функцию. Такой подход реализует концепцию **структурного программирвания** и делает программу более понятной и удобочитаемой. С другой стороны подобный подход обеспечивает реализацию технологий **функционального программирования**.

> **Функциональное программирование** – это стиль программирования, использующий только композиции функций. Другими словами, это программирование в выражениях, а не в императивных командах.

> **Функция** – это обособленный участок кода, который можно переиспользовать несколько раз в разных местах программы, обратившись к нему по имени, которым он был назван. Мы можем вызывать функцию с какими-то аргументами (или без них) и получать значения обратно. Функции позволяют упаковывать часть кода для его последующего повторного вызова. При вызове происходит выполнение команд тела функции (команд, находящихся внутри функции).

Зачем нужны функции?

Разберёмся, для чего используются функции:

1. **Структурирование кода** - функции позволяют сделать код более читабельным: увидеть одну строчку кода и по названию понять общую идею куда проще, чем разбираться с исходным кодом самого алгоритма.
2. **Выполнение одной задачи несколько раз**.
3. **Выполнение одной задачи, но с различными входными данными**.

Функции ценны тем, что они могут менять своё поведение при разных входных данных без вмешательства в исходный код самой функции. Функция становится для нас как «чёрный ящик». Мы можем только передавать нашей функции какие-то данные и получать результат её выполнения вне зависимости от того, что творится в основной программе. То, что будет происходить внутри, нас не интересует, нам важен только результат.

Определить функцию в Python можно двумя способами:

- с помощью оператора `def`;

- с помощью `lambda` -функции.

Первый способ позволяет использовать операторы. При втором – определение функции может быть только выражением.

§ 5.1 Определение функции с помощью оператора def

Для определения функции в языке Python, нужно использовать ключевое слово `def` , после которого указать **имя функции** и в круглых скобках – **формальные параметры** (аргменты функции). *С помощью отступов* задается блок кода функции, где располагаются выполняемые функцией команды - **тело функции**. Результат выполнения функции (одно или несколько значений) указывается после ключевого слова `return` .

В общем случае **описание функции** можно представить следующим образом:

```
def <имя_функции>(<список аргументов>):
    '''Строки документации'''
    <блок кода функции>
    return <возвращаемые параметры>
```

Имя функции по PEP8 определяется с использованием нижнего подчеркивания. *Список аргументов* определяет множество формальных параметров. Формальные параметры являются локальными именами внутри тела определения функции, а при вызове функции они заменяются на фактические параметры, которые определены в вызывающей программе и которые определяют выполнение вызываемой функции. *Строки документации* описывают процесс, который реализуется внутри функции. *Ключевое слово* `return` используется для возврата в основную программу значений из функции. Если не писать `return` , то умолчанию вернется `None` .

Функция может быть любой сложности, т.е. внутри конструкции `def -> return` , мы можем написать любой код.

Для вызова функции нужно использовать круглые скобки и, если нужно, передать туда параметры.

```
<имя_функции>(<список аргументов>)
```

Чтобы получить документационную строку, можно обратиться к атрибуту `__doc__` , а имя функции – с помощью атрибута `__name__` .

Все в Python является объектом, включая функцию. Функции в Python'е – это такие же объекты, как и, например, строки, списки или классы. Функцию можно передавать в другую функцию, их можно возвращать из функций, создавать внутри функций, их можно создавать на лету. Функции – это **объекты первого класса**.

> **Объект первого класса** – это объект, который может быть динамически создан, уничтожен, передан функции, возвращен как значение и имеет все права, как и другие переменные на языке программирования. Применительно к функции это означает, что нет ограничений на ее использование. Это то же самое, что и любой другой объект.

Пример. Определим функцию, которая находит сумму двух переданных ей аргументов.

In [1]:

```
def add(x, y):
    '''Функция принимает два аргумента и возвращает их сумму'''
    return x + y

print("Результат вызова функции 'add':", add(45, 31))
print("Результат вызова функции 'add':", add('abc', 'DEF'))
print('Описание:', add.__doc__)
print('Имя:', add.__name__)

f = add
print(f(1, 6))
print(f)
```

```
Результат вызова функции 'add': 76
Результат вызова функции 'add': abcDEF
Описание: Функция принимает два аргумента и возвращает их сумму
Имя: add
7
<function add at 0x0000014946F7CD30>
```

Самую простейшую в мире функциию можно определить следующим образом:

```
def f():
    '''TODO: '''
    pass
```

Эта функция не имеет параметров, ничего не делает и ничего не возвращает. Оператор `pass` означает «ничего не делай»; он используется там, где синтаксически необходим оператор, а делать ничего не нужно (после `if` или `elif` , после `def` и т.д.). Его удобно использовать при написании программ как "заглушку" в том месте, где потом будем вставлен некоторый код, который пока не пишм (в силу ряда причин).

In [14]:

```
def dist(x1, y1, x2, y2):
    '''Вычилиние длины отрезка, заданного координатами концов'''
    pass

def area(a, b, c):
    '''TODO'''
    pass

x1, y1 = map(float, input("Введи x1, y1: ").split())
x2, y2 = map(float, input("Введи x2, y2: ").split())
x3, y3 = map(float, input("Введи x3, y3: ").split())
a = dist(x1, y1, x2, y2)
b = dist(x3, y3, x2, y2)
c = dist(x1, y1, x3, y3)
s = area(a, b, c)
```

```
Введи x1, y1: 1 2
Введи x2, y2: 8 6
Введи x3, y3: 4 -5
```

§ 5.2 Определение функции с помощью lambda-функции

Часто возникает потребность возвратить результат вычисления некоторого выражения единожды, без повторного обращения к вычислениям. В этом случае нет необходимости создавать функцию. Достаточно будет определить функцию без имени, т.е. **анонимную функцию** с помощью `lambda` -выражения.

> **Лямбда-функции** – особые, анонимные функции, имеющие ряд ограничений, по сравнению с обычными функциями. Они локально решают единственную задачу, связанную с вычислением некоторого выражения) и возвращают только одно значение.

`Lambda` позволяет определить функцию прямо `in place` , то есть без ключевого слова `def` , использовать её на месте и забыть про нее потом. Пример **определения функции** с помощью `lambda` -выражения (лямбда-функции):

```
func = lambda x, y: x + y
```

Вышеприведенная функция определяет лямбда-выражение, которое принимает два аргумента и возвращает их сумму.

При использовании `lambda` -функции ее аргументы заключают в круглые скобки:

B [2]:

```
c = (lambda x, y: x + y) (3, 6)
print(c)
```

```
9
```

Применение лямбда-функции выглядит, как выражение, давайте посмотрим на примере:

```
# Обычная функция
def search_len(arg_1):
    return len(arg_1)

# Лямбда-функция
result = lambda x: len(x)
```

Обычно, лямбда-функции применяют при вызове функций, которые в качестве аргументов содержат функции. Проблема использования лямбда-функций состоит в том, что иногда усложняется читаемость кода.

```
func = lambda x, y: x + y
func(1, 2)                    # 3
func('a', 'b')                # 'ab'
(lambda x, y: x + y)(1, 2)    # 3
(lambda x, y: x + y)('a', 'b')  # 'ab'
```

Лямбда-функции не имеют имени, поэтому могут возникать проблемы с отловом ошибки.

Лямбда функция **работает по следующему принципу**:

`(lambda` *перечисляются аргументы через запятую* `:` **что то с ними делается** `)` `(` *передаем аргументы* `)`

В результате `lambda` -выражения получается **безымянный объект-функция**. Количество аргументов `lambda` -функции неограниченно, но возвращаемое ею значние только одно.

Обычно, лямбда-функции применяют при вызове функций, которые в качестве аргументов содержат функции.

`Lambda` -функция имеет синтаксические отличия от нормальной функции. В частности, **лямбда имеет следующие характеристики**:

- она может содержать только выражения и не может включать операторы в свое тело;
- она пишется как одна строка исполнения;
- она не поддерживает аннотации типов;
- она может быть немедленно вызвана (IIFE, *Immediately Invoked Function Expression*).

Пример. Используя `lambda` -функцию определить объем конуса.

B [3]:

```
from math import pi

h = float(input("Высота конуса = "))
r = float(input("Радиус основания = "))
v = (lambda h, r : (pi * r**2 * h)/3)(h, r)
print('Объем =', round(v, 2))
```

```
Высота конуса = 6
Радиус основания = 4
Объем = 100.53
```

Пример. Напишем функцию, которая превращает список чисел в список строк.

Чтобы превратить список чисел в список строк нам нужно использовать функцию `map` и передать в функцию `map` , конструктор класса `str` . При применении конструктора класса `str` к числу, у нас получается строка (именно это нам и нужно). Нам нужно пробежаться по списку и всё передать в конструктор `str` . Мы можем вызвать нашу функцию в `range` и получить список строк от нуля до девяти:

```
# Вариант 1 (с lambda-функцией).
ls = [1, 2, 5, -8]
print(list(map(lambda s: str(s), ls)))

# Вариант 2 (без lambda-функциии).
def string_list(num_list):
    return list(map(str, num_list))

ls = [1, 2, 5, -8]
print(string_list(ls))
```

В [5]:

```
def string_list(num_list):
    return list(map(str, num_list))

ls = [1, 2, 5, -8]
print(string_list(ls))
```

```
['1', '2', '5', '-8']
```

В [6]:

```
ls = [1, 2, 5, -8]
print(list(map(lambda s: str(s), ls)))
```

```
['1', '2', '5', '-8']
```

§ 5.3 Передача параметров в функцию и обратно

5.3.1 Аннотирование типов

В примере с определением функции `add` мы не указывали явно, какого типа параметры функция ожидает, потому что Python – это динамический язык. Поэтому функция не выдала ошибку при передаче ей слагаемых типа `str`, поскольку для них определена операция сложения (конкатенация). В Паскале и C, например, типы **аннотируются**, т.е. явно указывается, какого типа должен быть параметр функции и какого типа возвращаемые значения.

В Python'е последних версий появилась возможность аннотировать типы, и делается это с помощью **двоеточия для входных параметров и стрелочки для указания типа возвращаемого из функции значения**.

Однако, если мы передадим даже параметры других типов, то у нас код все равно исполняется, потому что Python – это динамический язык, и аннотация типов призвана помочь программисту или его IDE отловить какие-то ошибки. Если вы считаете это необходимым, можете использовать аннотацию типов.

```
def add(x: int, y: int) -> int:
    return x + y

print(add(10, 11))
print(add('Знание', add(' - ', 'сила')))
```

В [7]:

```
def add(x: int, y: int) -> int:
    return x + y

print(add(10, 11))
print(add('Знание', add(' - ', 'сила')))
```

```
21
Знание - сила
```

5.3.2 Механизм передачи параметров в функции

В статических языках программирования проводится очень четкое различие между передачей параметров по ссылке и по значению (например в Паскале это параметры-переменные и параметры-значения). В языке Python **каждая переменная** имеет определенную **связь идентификатора с объектом в памяти** компьютера. И именно это имя, именно эта ссылка на объект передается в вызываемую функцию.

Пример 1. Использование в качестве передаваемого входного параметра изменяемого объекта.

In [8]:

```
def extender(source_list, extend_list):
    source_list.extend(extend_list)

values = [1, 2, 3]
extender(values, [4, 5, 6])
print(values)
```

```
[1, 2, 3, 4, 5, 6]
```

В примере определили функцию `extender`, которая принимает `source_list` (типа `list`) и пытается расширить (с помощью метода списка `extend`) `source_list` с помощью `extend_list` а, то есть просто добавить в конец все его элементы. Определяем `values` и пытаемся расширить `values` с помощью $[4, 5, 6]$. Окажется, что `values` у нас изменится. Что же произошло? Наша ссылка на объект в памяти попала в `extender`. И так как `list` является изменяемым объектом, мы можем изменить этот `list`, и объект в памяти изменился. Т.о. `values` глобально поменял свое значение.

Пример 2. Использование в качестве передаваемого входного параметра неизменяемого объекта.

In [9]:

```
def replacer(source_tuple, replace_with):
    source_tuple = replace_with

user_info = ('Иванов', '8-910-563-25-32')
replacer(user_info, ('Петров', '8-920-365-56-36'))
print(user_info)
```

```
('Иванов', '8-910-563-25-32')
```

В примере мы попытаемся как-то изменить неизменяемое значение, в данном случае `tuple`, то у нас ничего не получится, потому что мы передаем ссылку на объект в памяти, но объект является неизменяемым. В данном случае у нас `user_info` осталось неизменным.

Важно! Стоит быть внимательным с изменением каких-то глобальных переменных внутри функции. Это является плохим тоном, и не стоит так программировать, потому что часто бывает не очевидно, если вы вызываете какую-то функцию, а объект глобально изменяется. Используйте возвращаемое значение и не путайте других программистов.

5.3.3 Именованные аргументы

В Python существуют **именованные аргументы**, которые иногда бывают полезны.

```
def say(greeting, name):
    print('{} {}!'.format(greeting, name))

say('Привет', 'Мир')
say(name='Мир', greeting='Привет')
```

В примере мы можем определить функцию `say`, которая принимает два аргумента – `greeting` и `name`, просто какого-то приветствует. Однако мы можем также передать эти параметры в другом порядке, проименовав их с помощью `name` и `greeting`. Таким образом, мы передаем вначале имя, а потом `greeting`, тем не менее, всё работает точно так же.

Пример. Используя именнованне аргументы написать функцию вычисления объема конуса.

В [ ]:

```
from math import pi

def cone_volume(height, radius):
    return (pi * radius**2 * height)/3

h = float(input("Высота конуса = "))
r = float(input("Радиус основания = "))
v = cone_volume(radius=r, height=h)
print('Объем =', round(v, 2))
```

Здесь мы в явном виде указали какой аргумент должен принимать какое значение. В этом заключается работа именованных аргументов.

В функции можно выделить ***позиционные*** (*positional*) и ***именованные*** (*keyword*) аргументы. Из названий можно предположить, что одни аргументы зависят от позиции, а вторые от имени (всё логично).

Важная особенность: все именованные аргументы должны идти строго после позиционных, как при объявлении функций, так и при их вызове.

Рассмотрим практический пример:

В [24]:

```
def func(a, b, c):
    print('a =', a)
    print('b =', b)
    print('c =', c)

# Правильно
func(1, 2, c=3)

# Неправильно
# func(a=1, 2, 3)
```

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

5.3.4 Аргументы функции, заданные по умолчанию

Если попытаться вызвать функцию без какого-либо параметра, который она ожидает, то выпадет ошибка `TypeError`, потому что функция ожидает параметр и не знает, что делать, если его нет. Так, в результате выполнения следующего кода возникнет ошибка:

```
from math import pi

def cone_volume(height, radius):
    return (pi * radius**2 * height)/3

print('Объем =', round(cone_volume(6), 2))
```

В [19]:

```
from math import pi

def cone_volume(height, radius):
    return (pi * radius**2 * height)/3

print('Объем =', round(cone_volume(6), 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-33d8976f19c7> in <module>
      4     return (pi * radius**2 * height)/3
      5
----> 6 print('Объем =', round(cone_volume(6), 2))

TypeError: cone_volume() missing 1 required positional argument: 'radius'
```

Часто возникают ситуации, когда у функции имеются аргументы, которые **можно передавать, а можно не передавать**. И у этих аргументов, могут быть какие-то заданные по умолчанию значения.

Если при описании функции аргументам (которые указываются в круглых скобках после имени функции) задать начальные значения, то они будут считаться необязательными. Обязательные агрументы задаются указанием имени. При описании функции **вначале указываются обязательные параметры** функции, а затем не обязательные. При вызове функции допускается не указывать необязательные параметры. В этом случае зачения аргументов не изменяются и определяются начальными значениями.

В [3]:

```
from random import randint

def newlist(a, b, n=10):
    return [randint(a, b) for _ in range(n)]

ls = newlist(-10, 10, 15)
ls1 = newlist(-5, 20)
print(ls)
print(ls1)
```

```
[7, 1, -7, 3, 9, 0, -9, 6, -2, 0, 4, -3, 8, 0, 5]
[1, -3, 16, 3, 19, 4, -3, 9, 12, 9]
```

Важно! Стоит быть внимательным с аргументами по умолчанию, если мы используем в качестве

аргументов по умолчанию изменяемые значения.

Пример. Использование в качестве аргументов по умолчанию изменяемых значений. Пусть имеется функция `append_one` , и в качестве значения по умолчанию `iterable` используется список. Собственно `append_one` просто прибавляет единичку к переданному списку или дефолтному списку, т.е. возвращается либо исходный список плюс один, либо список из единички. Если мы передадим в `append_one` список из единицы, нам вернется две единички, что мы и ожидаем.

В [12]:

```
def append_one(iterable=[]):
    iterable.append(1)
    return iterable

print(append_one([1]))
```

```
[1, 1]
```

Однако что же произойдет, если мы вызовем `append_one` два раза?

В [13]:

```
print(append_one())
print(append_one())
```

```
[1]
[1, 1]
```

Вначале, как мы и ожидаем, к нам вернется единичка, потому что мы взяли наше дефолтное значение, добавили единичку в список и вернули. Однако если мы вызовем во второй раз, окажется, что у нас уже две единички, хотя мы явно ожидаем одну.

Чтобы понять, почему это так, можно посмотреть на дефолтное значение нашей функции, и окажется, что там уже содержатся эти самые две единички.

В [14]:

```
print(append_one.__defaults__)
```

```
([1, 1],)
```

Почему так происходит? При определении функции, когда интерпретатор Python'a бежит по файлу с кодом, определяется связь между именем функции и дефолтными значениями. У каждой функции появляется словарь (`tuple`) с дефолтными значениями. И именно в эти переменные каждый раз и происходит запись. В рассматриваемом примере список создается один раз и каждый раз, когда мы к нему обращаемся, мы обращаемся к одной и той же переменной - не важно, сколько бы мы не вызывали функцию раз, всегда будем работать не с новым значением списка (как это было бы с неизменяемыми типами данных), а работаем с одним и тем же списком.

Таким образом, если дефолтные значения являются изменяемыми, в них можно записывать, потому что это обычные переменные.

Что же нужно делать в таком случае? Нужно определять дефолтные значения как `None`. И если нам не передан какой-то параметр, мы просто создаем новый список на лету. Можно это делать двумя механизмами, как в примере:

В [ ]:

```
# Способ 1. Задание значений изменяемых объектов.
def function(iterable=None):
    if iterable is None:
        iterable = []
```

В [ ]:

```
# Способ 2. Задание значений изменяемых объектов.
def function(iterable=None):
    iterable = iterable or []
```

Правильный вариант последнего примера будет, например, таким:

В [30]:

```
def append_one(iterable=None):
    iterable = iterable or []
    iterable.append(1)
    return iterable

print(append_one([1]))
print(append_one([1, 2, 3]))
print(append_one.__defaults__)
```

```
[1, 1]
[1, 1]
[1, 2, 3, 1]
(None,)
```

Важно! Аргументы по умолчанию создаются один раз в момент инициализации функции (первого прочтения интерпретатором инструкции `def`), а не при каждом вызове функции. Если в качестве аргументов по умолчанию использовать изменяемые типы данных (списки, словари), то они создаются один раз и будут использоваться на протяжении времени жизни функции, так как **при передаче словаря передаются не все его значения, а указатель на первый его элемент**.

В качестве значений по умолчанию НЕ рекомендуется использовать изменяемые типы: множество, список, словарь. В качестве значений по умолчанию используются неизменяемые типы: целое число, строка, кортеж. Использование в качестве значения по умолчанию аргумента функции изменчивого объекта (списка, множества, словаря) может привести к казусам, поскольку этот объект – один и тот же для всех вызовов функций.

5.3.5 Функция, возвращающая несколько значений

В языке Python функция может возвратить **только одно значение**, которое может быть кортежем. А **количество элементов в кортеже может быть любым**.

Пример. Составим функцию, одновременно выполняющую основные операции с целочисленными значениями:

- определение суммы цифр в числе;
- определение произведения цифр в числе;
- определение максимальной/ минимальной цифр в числе.

В [16]:

```
def digit_analiz(n):
    summa = 0
    product = 1
    mn = 9
    mx = 0
    while n > 0:
        digit = n % 10
        summa +=  digit
        product *= digit
        if digit >  mx:
            mx = digit
        elif digit < mn:
            mn = digit
        n //= 10
    return summa, product, mn, mx

s, p, min_n, max_n = digit_analiz(8123)
print('Сумма цифр =', s)
print('Произведение цифр =', p)
print('Минимальная цифра =', min_n)
print('Максимальная цифра =', max_n)
```

```
Сумма цифр = 14
Произведение цифр = 48
Минимальная цифра = 1
Максимальная цифра = 8
```

5.3.6 Определение функции, способной принимать любое число аргументов

Очень красивой особенностью Python'а является возможность **определения функции, которая принимает разное количество аргументов**.

Пример. Составим функцию сложения для неизвестного количества входных аргументов.

В [21]:

```
def add(*args):
    return sum(args)

print(add(2, 8, 5, 7))    # выведет 22
print(add(10, 8))         # выведет 18

l = [1, 9, 6]
print(add(*l))            # выведет 16
```

```
22
18
16
```

Для обработки переменного количества аргументов используется **конструкция со звездочкой**. Ключевой момент – использование астерИска, т.е. звездочки «*», которая означает упаковку всех входящих аргументов в кортеж, а `args` – просто название переменной, которое может быть любым. В теле функции мы можем с этой переменной `args` что-то делать. Название идентификатора `args` – это соглашение между разработчиками.

Чтобы передать в функцию `add` структуру список, этот список тоже надо пометить звездочкой. Если не указать звездочку, то при входе в функцию список будет упакован в кортеж и получится структура `([1, 9, 6],)` – кортеж из одного элемента. Кроме того возникнет ошибка.

Важно! В конструкцию со звездочкой нельзя передавать генераторы, поскольку это может привести к неконтролируемому расходу памяти.

Пример. Упаковка – распаковка аргументов в список. Определим функцию `printer`, которая принимает разное количество аргументов.

В [29]:

```
def printer(*args):
    for item in args:
        print(item)

ls = ['Колчин', 'Пылькин', 'Перепелкин', 'Соколова' ]
printer(*ls)
```

```
('Колчин', 'Пылькин', 'Перепелкин', 'Соколова')
Колчин
Пылькин
Перепелкин
Соколова
```

Пример. Написать функцию, которая будет перемножать любое количество переданных ей аргументов.

In [29]:

```
def mul(*nums):
    p = 1
    for n in nums:
        p *= n
    return p

print(mul(3, 6, 7, 5))
print(mul(*[3, 6, 7, 5]))
```

```
630
630
```

Пример. Упаковка – распаковка аргументов в словарь.

Определим функцию `printer`, которая принимает разное количество именованных аргументов. Собственно, все записывается в `kwargs`, и `kwargs` останется словарем `dict`. Если мы передадим два именованных аргумента, `a=10` и `b=11`, то у нас получится словарь, и мы можем потом этот словарь как-то использовать, используя параметры и, например, просто их выводя на экран.

In [30]:

```
def printer(**kwargs):
    print(type(kwargs))
    for key, value in kwargs.items():
        print('{}: {}'.format(key, value))

printer(a=10, b=11)
```

```
<class 'dict'>
a: 10
b: 11
```

Точно так же мы можем разыменовывать, разворачивать эти словари в обратную сторону, используя `**`.

In [31]:

```
def printer(**kwargs):
    print(type(kwargs))
    for key, value in kwargs.items():
        print('{}: {}'.format(key, value))

dict_ab = {'a': 10, 'b': 11}
printer(**dict_ab)
```

```
<class 'dict'>
a: 10
b: 11
```

Таким образом, если у нас есть словарь, мы можем передавать значения из этого словаря как именованные аргументы в нашу функцию. Таким образом, когда мы используем **две звездочки** при вызове функции, у нас первым параметром становится `a`, а вторым параметром становится `b`.

Важно! функцию можно вызвать без параметров, если:

1. используются `*args` , `**kwargs` ;
2. у всех аргументов есть значения по умолчанию;
3. она не ожидает аргументов.

Пример. Функция с обычными (позиционными) и именованными аргументами:

В [54]:

```
def knife(arg, *args, **kwargs):
    print(arg)
    print(args)
    print(kwargs)
    return None

# knife(1)  # 1  ()  {}
# knife(1, 2, 3, 78, 4, 5)
# knife(1, 2, a='abc', b='sdf')
# knife(1, a='abc', 3, 4) # !!! ошибка

lst = [2, 3, 4, 5]
dct = {'a': 'abc', 'b': 'sdf'}
knife(10, *lst, **dct)
```

```
10
(2, 3, 4, 5)
{'a': 'abc', 'b': 'sdf'}
```

Резюмируем: чтобы функция могла принимать неограниченное количество позиционных аргументов, есть специальная конструкция `*args` , а для именованных аргументов – `**kwargs` . `Args` и `kwarg s` не являются зарезервированными словами, это просто общее обозначение, `args` – это сокращение от *arguments* (аргументы), а `kwargs` – сокращение от *keyword arguments* (именованные аргументы). Важно, что они должны начинаться с одной и двух звёздочек соответственно. Каждая из этих конструкций используется для распаковки аргументов соответствующего типа, позволяя вызывать функции со списком аргументов переменной длины, как в случае функции `print` .

Чтобы правильно обрабатывать `*args` и `**kwargs` , нужно представлять, чем они являются. Собственно, `args` – это кортеж, а `kwargs` – это словарь.

5.3.7 Области видимости переменных

> **Области видимости** определяют, в какой части программы мы можем работать с той или иной переменной, функцией и т.п., а от каких они «скрыты». Крайне важно понимать, как использовать только те значения и переменные, которые нам нужны, и как интерпретатор языка себя при этом ведёт.

В Python существует 3 области видимости:

- локальная – то, что определяется в теле функции;
- глобальная – то, что определяется в самой программе;
- нелокальная (добавлена в Python 3).

5.3.7.1 Локальная область видимости

Переменные, определяемые внутри инструкции `def`, видны только программному коду внутри инструкции `def`. К этим именам нельзя обратиться за пределами функции. То есть переменные внутри функций находятся внутри *локальной области видимости*.

Имена, определяемые внутри инструкции `def`, не вступают в конфликт с именами, находящимися за пределами инструкции `def`, даже если и там, и там присутствуют одинаковые имена переменных.

В [1]:

```
def local():
    x = 10  # локальная переменная
    print(x)   # 10

x = 50
local()
print(x)       # 50
```

```
10
50
```

Переменная `x`, которой присвоено значение 50 за пределами функции `local`, полностью отлична от переменной `x`, которой присвоено значение 10 внутри функции `def`. Здесь можно провести такую аналогию: если локальную область видимости рассматривать как бункер, то, что происходит внутри этого бункера, не видно внешнему миру. В этом и есть принцип локальной области видимости.

В любом случае **область видимости переменной** (где она может использоваться) всегда **определяется местом, где ей было присвоено значение**, и не имеет никакого отношения к месту, откуда была вызвана функция.

5.3.7.2 Глобальная область видимости

Рассмотрим следующий пример:

В [1]:

```
def local():
    print(x)  # 10 - так как x нет в локальной области видимости,
              # мы берем её из глобальной области

x = 10
local()
print(x)      # 10
```

```
10
10
```

Если запустить данный код, то мы увидим, что распечаталось два раза значение 10. Это происходит потому, что, не найдя переменную в локальной области, функция обращается к глобальной области видимости, которая находится на уровне модуля, и берёт значение оттуда. И здесь не нарушается так называемое «правило бункера». Из бункера мы можем увидеть, что происходит во внешнем мире, но вот наоборот это не работает.

Рассмотрим еще один пример (который приведет к ошибке):

```
x = 3

def func():
    print(x)   # --> UnboundLocalError: local variable 'x' referenced before assig
nment
    x = 5
    x += 5
    return x

print(func())
```

Ошибка возникает в строке `print(x)` (**UnboundLocalError**: *local variable `x` referenced before assignment*), потому что Python замечает, что вы пытаетесь распечатать значение **локальной** (!) переменной `x` в функции `func` до её объявления, что и приводит к ошибке, так как `x` ещё не определён.

Ещё раз повторим правило, которое было озвучено ранее: **область видимости переменной** (где она может использоваться) **всегда определяется местом, где ей было присвоено значение**. То есть переменная `x` в четвертой строке не имеет никакого отношения к переменной `x` в первой строке. Они просто имеют одинаковые названия, не более того. Соответственно, переменная `x` здесь выступает как локальная и никакого отношения к глобальной области не имеет. Интерпретатор начинает проверять локальную область видимости функции `func`, видит, что тут объявлена такая переменная, пытается обратиться к ней, но не может получить её значение, потому что эта переменная получает его в пятой строке.

Важно понимать, что **переменные, объявленные вне области видимости функции, нельзя изменять**. Пусть имеется глобальная переменная `x`, и мы пытаемся прибавить к ней внутри функции какое-то значение.

```
x = 3

def func():
    x += 5
    return x

print(func())
```

Мы пытаемся к `x` прибавить 5 при вызове функции `func`. У нас ничего не получается, возникает ошибка (локальная переменная `x`, на которую ссылаются до присваивания), потому что мы **не можем внутри функции изменять объекты из глобальной области видимости**. Если мы внутри функции изменяем какие-то глобальные объекты, очень часто это не очевидно. У нас есть глобальная переменная, и вызов каких-то функций меняет ее значение. Совершенно непонятно, что это происходит и часто приводит к запутанному коду.

В [7]:

```
x = 3

def func():
    x += 5
    return x

print(func())
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-7-77d3da13e41f> in <module>
      5     return x
      6
----> 7 print(func())

<ipython-input-7-77d3da13e41f> in func()
      2
      3 def func():
----> 4     x += 5
      5     return x
      6

UnboundLocalError: local variable 'x' referenced before assignment
```

В Питоне **аргументы функции передаются по ссылке**, а не копируются при вызове функции. Аргументы и определенные внутри функции переменные попадают в локальную область видимости функции (`locals()`).

Иногда возникает необходимость изменить глобальные данные из функции. Это можно сделать с помощью ключевого слова `global` , который объявляет переменную доступной для блока кода, следующим за оператором. Давайте попробуем использовать глобальную область видимости для исправления нашей ошибки из предыдущего примера:

В [2]:

```
x = 3

def func():
    global x     # объявляем, что переменная является глобальной
    print(x)
    x = 5
    x += 5
    return x

func()
print(x)
```

```
3
10
```

В данном случае с помощью оператора `global` мы в локальную область видимости помещаем нашу глобальную переменную `x` .

Хотим обратить ваше внимание на то, что **не нужно менять глобальные переменные внутри функции**. Комьюнити Python объявило такую практику очень нежелательной, так как из-за этого исправление кода становится намного сложнее. Использование `global`, как и в других языках, **не является признаком хорошего тона**.

Так, если в следующей программе скрыть содержимое функции, то не понятно, почему сначала $PI = 3.14$, а потом $PI = 3.1415$.

В [16]:

```
PI = 3.14

def area_circle(r):↔

print('Число PI до вызова функции =', PI)
print(area_circle(10))
print('Число PI после вызова функции =', PI)
```

```
Число PI до вызова функции = 3.14
314.15000000000003
Число PI после вызова функции = 3.1415
```

С помощью функций, классов и модулей в большинстве случаев удается обойтись без модификации глобальных объектов.

Резюме про локальные и глобальные области видимости.

- Функции образуют локальную область видимости, а скрипты (программы) – глобальную.
- Эти две области взаимосвязаны между собой следующим образом: каждый скрипт – это глобальная область видимости, то есть пространство имен, в котором создаются переменные на уровне файла.
- Каждый вызов функции создаёт новую локальную область видимости. По умолчанию все имена, которым присваиваются значения внутри функции, помещаются в локальную область видимости. Если необходимо присвоить значение имени верхнего уровня в модуле (глобальной переменной), это имя необходимо объявить внутри функции глобальным с помощью инструкции `global`.

5.3.7.3 Нелокальная область видимости

Понятие *нелокальная область видимости* появилось в Python 3 вместе с ключевым словом `nonlocal`. Логика его написания примерно такая же, как и у `global`. Но у `nonlocal` есть особенность. `Nonlocal` используется чаще всего во вложенных функциях, когда мы хотим дать интерпретатору понять, что для вложенной функции определённая переменная не является локальной, но она и не является глобальной в общем смысле. Предположим, мы хотим сделать функцию, которая будет возвращать нам функции.

Резюме про `global` и `nonlocal`.

В Python'e при изменении переменных можно указывать ключевые слова `global` или `nonlocal` (не рекомендуется использовать):

- утверждение `global` может использоваться для указания того, что конкретные переменные находятся в глобальной области видимости и должны быть восстановлены там;
- утверждение `nonlocal` указывает на то, что конкретные переменные живут в рамках ограждающей области видимости и должны восстанавливаться там.

Пример использования `global` и `nonlocal` .

В [8]:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("После локального присвоения:", spam)
    do_nonlocal()
    print("После нелокального присваивания:", spam)
    do_global()
    print("После глобального присвоения:", spam)

scope_test()
print("В глобальной области видимости:", spam)
```

```
После локального присвоения: test spam
После нелокального присваивания: nonlocal spam
После глобального присвоения: nonlocal spam
В глобальной области видимости: global spam
```

5.3.7.4 Еще раз о пространствах имен и функциях

В Питоне используется несколько пространств имен, вложенных друг в друга:

1. Пространство имен модуля (`globals`), которое включает в себя пространство имен встроенных функций (`builtins`).
2. Пространство имен объемлющих функций (`nonlocal`).
3. Пространство имен текущей функции (`locals`).

Переменная в функции может относиться только к одному из этих пространств имен. Рассмотрим следующий пример:

В [13]:

```
x = 3

def func():
    print(x)
    x = 1

# Раскомментируйте, чтобы получить ошибку
# func()
```

Компилятор Питона при трансляции функции руководствуется следующими правилами:

1. Если `x` не используется в левой части присваивания, а также отсутствует в объемлющих функциях, то `x` трактуется, как элемент из `globals`.
2. Если `x` не используется в левой части присваивания, но присутствует в объемлющей функции, то формируется `nonlocal`-ссылка на `x` из объемлющей функции.
3. Если `x` присваивается некоторое значение, то `x` является элементом `locals`.

В примере выше `x` трактуется, как локальная переменная, поэтому вызов `print` до присваивания приводит к ошибке. Разумеется, с помощью `global` или `nonlocal` можно явно указать, к какому пространству имен принадлежит `x`:

In [15]:

```
x = 3

def func():
    global x
    print(x)
    x = 1

func()
print(x)
```

```
3
1
```

5.3.8 Примеры решения задач

Пример. Определить наименьшее общее кратное двух чисел (НОК).

Для вычисления НОК двух чисел, надо их произведение разделить на НОД (наибольший общий делитель).

In [5]:

```
def nod(m, n):
    while m * n != 0:
        if m > n:
            m -= n
        else:
            n -= m
    return m + n

def nok(a,b):
    return a*b/nod(a,b)

print(nok(45, 75))
```

```
225.0
```

Пример. Рассматривается множество целых чисел, принадлежащих числовому отрезку $[2095; 19402]$, которые являются простыми числами и у которых первая цифра больше последней. Найдите количество таких чисел и наибольшее такое число, которое оканчивается на 21. Отдельно определите количество палиндромов в этом диапазоне. В ответе запишите три целых числа: сначала количество, затем наибольшее такое число, которое оканчивается на 21, количество палиндромов.

Ответ: 455 9721 173

B [1]:

```
def ispalindrome(n):
    return str(n) == str(n)[::-1]

def isprime(n):
    if n == 1:
        k = 1
    else:
        k = 0
    d = 2
    while d * d <= n and k == 0:
        if n % d == 0:
            k += 1
        d += 1
    return k==0

def firstlast(n):
    return str(n)[0] > str(n)[-1]

a, b = 2095, 19402
cnt, mx, k = 0, a - 1, 0
for n in range(a, b + 1):
    if isprime(n) and firstlast(n):
        cnt += 1
        if n % 100 == 21:
            mx = n
    if ispalindrome(n):
        k += 1
print(cnt, mx, k)
```

```
455 9721 173
```

Пример. Напишите программу, которая принимает арифметическое выражение в качестве аргумента и выводит результат этого выражения. Необходимо использовать функции. Программа должна поддерживать следующие арифметические операции: `+`, `-`, `/`, `*`, `%` (получение процента от числа), `**` (возведение в квадрат), `**x` (возведение в степень числа `x`).

В [10]:

```
def summa(st):
    ls = st.split('+')
    return int(ls[0]) + int(ls[1])

def difference(st):
    ls = st.split('-')
    return int(ls[0]) - int(ls[1])

def product(st):
    ls = st.split('*')
    return int(ls[0]) * int(ls[1])

def division(st):
    ls = st.split('/')
    if int(ls[1]) != 0:
        return int(ls[0]) / int(ls[1])
    else:
        return 'Деление на ноль'

def percent(st):
    ls = st.split('%')
    return int(ls[0]) / 100

def power(st):
    ls = st.split('**')
    if ls[1] == '':
        st = 2
    else:
        st = int(ls[1])
    return int(ls[0]) ** st

v = input('Введите выражение: ')
if '+' in v:
    print(summa(v))
elif '-' in v:
    print(difference(v))
elif '**' in v:
    print(power(v))
elif '*' in v:
    print(product(v))
elif '/' in v:
    print(division(v))
elif '%' in v:
    print(percent(v))
else:
    print('Действие не выполнимо в калькуляторе')
```

```
Введите выражение: 25**2
625
```

Можно убрать всю логику вычислений в функцию, оставив в основной программе только вызов функции `calc` , реализующей функционал калькулятора. Кроме того в функцию добавим удаление пробельных символов для случая, когда пользователь разделял в выражении действия и операнды пробелами.

В [15]:

```
def calc(string):
    for i in string:
        if i == ' ':
            string=string.replace(' ', '')
    if '+' in string:
        var = string.split('+')
        print(int(var[0])+int(var[-1]))
    elif '-' in string:
        var = string.split('-')
        print(int(var[0])-int(var[-1]))
    elif '/' in string:
        var = string.split('/')
        print(int(var[0])/int(var[-1]))
    elif '*' in string and '**' not in string:
        var = string.split('*')
        print(int(var[0])*int(var[-1]))
    elif '%' in string:
        var = string.split('%')
        print(int(var[0])/100)
    elif '**' in string:
        var = string.split('**')
        if var[-1] == '':
            print(int(var[0])**2)
        else:
            print(int(var[0])**int(var[-1]))

calc('1+ 9')
calc('100%')
calc('4**')
calc('25 ** 2')
```

```
10
1.0
16
625
```

5.3.9 Задания для самостоятельного выполнения

При выполнении работы необходимо, воспользовавшись прицпами нисходящего программирования (проектированием приложений "сверху-вниз"), выделить необходимые для решения задачи функции.

Задача 1. Найти все простые натуральные числа, не превосходящие n, двоичная запись которых представляет собой палиндром, т. е. читается одинаково слева направо и справа налево.

Задача 2. Составить программу для нахождения чисел из интервала $[a, b]$, имеющих наибольшее количество делителей.

Задача 3. Найти все натуральные числа, не превосходящие заданного n, которые делятся на каждую из своих цифр.

Задача 4. На отрезке $[100, N]$ $(210 < N < 231)$ найти количество чисел, составленных из цифр a, b, c.

Задача 5. Из заданного числа вычли сумму его цифр. Из результата вновь вычли сумму его цифр и т. д. Через сколько таких действий получится нуль?

Задача 6. Два простых числа называются «близнецами», если они отличаются друг от друга на 2 (например, 41 и 43). Напечатать все пары «близнецов» из отрезка $[n, 2n]$, где n – заданное натуральное число, большее 2.

Задача 7. Два натуральных числа называются «дружественными», если каждое из них равно сумме всех делителей (кроме его самого) другого (например, числа 220 и 284). Найти все пары «дружественных» чисел, которые не больше данного числа N.

Задача 8. Натуральное число, в записи которого n цифр, называется числом Армстронга, если оно равно сумме своих цифр, возведённых в степень n (равную количеству его цифр). Например, число 153 содержит 3 цифры, и $153 = 1^3 + 5^3 + 3^3$, таким образом 153 – это число Армстронга.

Задача 9. Написать программу-калькулятор, которая производит арифметические операции (сложение, вычитание, умножение, деление) над целыми числами в q-ичной системе счисления. В случае получения дробного результата при делении количество выводимых символов после запятой в результате должно задаваться в качестве входного параметра, причем необходимо предусмотреть задание этого значения по умолчанию в функции.

Задача 10. Написать программу-конвертер между p-ичной и q-ичной системами счисления, которая позволяет переводить положительные целые и дробные числа из одной системы счисления в другую.

§ 5.4 Рекурсия

Как видно, функции могут вызывать другие функции – это вполне обыденная ситуация. При этом функция может вызывать саму себя. Такой тип вызова называется рекурсивным.

> **Рекурсивная функция** – это функция, вызывающая сама себя и обрабатывающая полученный результат до тех пор, пока не достигнет терминального условия (условия остановки). Количество раз, сколько функция вызвала сама себя, называется **глубиной рекурсии**.

Самый простой пример рекурсивного вызова функции – вычисление факториала числа:

In [40]:

```
# Рекурсивное определение факториала
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n - 1)

print(fac(5))
```

```
120
```