**Функции 2**

**Локальные и глобальные переменные**

**Глобальные переменные** – те переменные, которые находятся вне функции.

**Локальные переменные** – те переменные, которые используются внутри функции.

Для создания переменных, обладающих локальной областью видимости, необходимо всего лишь поместить их в отдельный блок кода, изолированный от остальной программы. Чтобы увидеть локальную переменную в действии, достаточно инициализировать целочисленный объект с именем x и значением 100 в функции f, как это сделано в следующем примере:

**def** f():

x = 100

**print**(x)

f()

Будет напечатано 100.

Здесь x имеет локальную область видимости, так как доступна лишь в рамках своей функции f. Вызывая данную функцию из внешней части программы, можно увидеть вывод целочисленного значения на экране. Однако, если попытаться вывести переменную x при помощи метода print вне зоны действия функции f, компилятор тут же выдаст ошибку:

**def** f():

x = 100

**print**(x)

f()

**print**(x)

Так происходит из-за того, что внешняя часть программы ничего не знает о переменной x, поскольку содержит в себе совсем другое пространство имен. Пользоваться локальными объектами можно только в той области, где они были идентифицированы. В обратном же случае компилятор сообщит об ошибке, не сумев обнаружить необходимую переменную.

Чтобы иметь возможность использовать некоторое значение в любой части программы, следует объявить глобальную переменную. Для этого понадобиться создать переменную отдельно от области кода, ограниченной определенным блоком кода, например, функцией. В следующем примере демонстрируется идентификация целочисленного типа данных под названием x, который позже выводится на экран при помощи метода print в функции f:

x = 100

**def** f():

**print**(x)

f()

**print**(x)

Будет выведено два числа 100.

Как можно заметить из результатов выполнения программы, значение 100 воспроизводится не только через f, но и с помощью обычного print. Таким образом, получение доступа к x осуществляется из любой части кода, благодаря глобальной области видимости подобного объекта. Но что будет, если попытаться изменить значение глобальной переменной в некой функции? Результаты такого эксперимента представлены в следующем фрагменте кода:

x = 100

**def** f():

x = 200

f()

**print**(x)

Будет выведено 100.

Функция f присваивает значение 200 переменной с именем x, однако, вопреки ожиданиям, внешний метод print выводит число 100, которое принадлежало x изначально. Происходит так потому, что в данной программе создаются два разных объекта x с локальной, а также глобальной областью видимости. Исправить ситуацию поможет ключевое слово **global**:

x = 100

**def** f():

**global** x

x = 200

f()

**print**(x)

200

Пометив переменную x как **global,** можно обращаться к ее изначальному значению, которое было определено вне зоны действия функции f. Теперь после того как в x поместили число 200, вызов метода print выводит вполне ожидаемый результат, то есть измененное значение.

**Если внутри функции вы нигде не меняете значение переменной, то переменная глобальная.**

**Если внутри функции вы меняете значение переменной, то переменная локальная.**

Локальная переменная исчезнет после завершения функции.

Python не проверяет будет ли когда-либо выполняться присвоение. Если оно написано, то переменная автоматически становится локальной

**def** f():

**if False**:

n=0

**print**(n)

n – локальная переменна

**Области видимости**

Общее правило, касающееся видимости переменных:

1. Внутри функции видны все переменные этой функции (локальные переменные и аргументы функции).
2. Также видны переменные, которые определены снаружи, вне функции.
3. Снаружи не видны никакие переменные, которые определены внутри функции.

**Изменяемые и неизменяемые аргументы функции**

В аргументы, имена которых заданы при объявлении функции, записываются объекты, передаваемые функциям при вызове. При этом, если соответствующим локальным переменным функций, их параметрам, что-то присваивают, эта операция не влияет на передаваемые функциям неизменяемые объекты. Например:

**def** foo(a):

    a = a+5

**print**(a)             # Выводит 15

a = 10

foo(a)

**print**(a)                 # Выводит 10

Как видно, вызов функции никак не повлиял на переменную a. Именно это происходит в том случае, если функции передаётся неизменяемый объект (о чем было сказано выше).

А если же функциям передают изменяемые объекты, то можно столкнуться с поведением системы, которое отличается от вышеописанного.

**def** foo(lst):

    lst = lst + ['new entry']

**print**(lst)          # Выводит ['Book', 'Pen', 'new entry']

lst = ['Book', 'Pen']

**print**(lst)            # Выводит ['Book', 'Pen']

foo(lst)

**print**(lst)            # Выводит ['Book', 'Pen']

Все также, как описано выше. Но если как-то повлиять на элементы изменяемого объекта, переданного функции, мы станем свидетелями кое-чего другого.

**def** foo(lst):

    lst[1] = 'new entry'

**print**(lst)                # Выводит ['Book', 'new entry']

lst = ['Book', 'Pen']

**print**(lst)                     # Выводит ['Book', 'Pen']

foo(lst)

**print**(lst)                     # Выводит ['Book', 'new entry']

Как видите, объект из параметра lst был изменён после вызова функции. Произошло это из-за того, что мы работаем со ссылкой на объект, хранящейся в параметре lst. В результате изменение содержимого этого объекта выходит за пределы функции. Избежать этого можно, просто выполняя глубокие копии подобных объектов и записывая их в локальные переменные функции.

**def** foo(lst):

    lst = lst[:]

    lst[1] = 'new entry'

**print**(lst)               # Выводит ['Book', 'new entry']

lst = ['Book', 'Pen']

**print**(lst)                   # Выводит ['Book', 'Pen']

foo(lst)

**print**(lst)                 # Выводит ['Book', 'Pen']

**Позиционные аргументы**

Рассмотрим следующий код:

**def** diff(x, y):

**return** x - y

res = diff(10, 3)

**print**(res)

Такой код выведет число 7. При вызове функции diff() первому параметру x будет соответствовать первый переданный аргумент, 10, второму параметру y – второй аргумент, 3.

Обычно функции имеют **позиционные аргументы**. Такие аргументы передаются без указания имен. Они называются **позиционными**, потому что именно по позиции, расположению аргумента, функция понимает, какому параметру он соответствует.

В Python можно использовать не только позиционные, но и именованные аргументы.

**Именованные аргументы**

Аргументы, передаваемые с именами, называются **именованными**. При вызове функции можно использовать имена параметров из ее определения.

Рассмотрим следующий код:

**def** diff(x, y):

**return** x - y

res = **diff**(x=10, y=3) # используем именованные аргументы

**print**(res)

Такой код по-прежнему выведет число 7. При вызове функции diff() мы явно указываем, что параметру x соответствует аргумент 10, а параметру y— аргумент 3.

Использование именованных аргументов позволяет нарушать их позиционный порядок при вызове функции. **Порядок упоминания именованных аргументов не имеет значения!**

Мы можем вызвать функцию diff() так:

res = diff(y=3, x=10)

и получить тот же результат 7.

Обычно придерживаются следующего правила, если функция принимает больше трёх аргументов, нужно хотя бы часть из них указать по имени. Особенно важно именовать значения аргументов, когда они относятся к одному типу, ведь без имен очень трудно понять, что делает функция с подобным вызовом.

Мы сталкивались с использованием именованные аргументы sep и end при вызове функции print.

Можно вызывать функции, используя именованные и позиционные аргументы одновременно. Но позиционные значения должны быть указаны **до** любых именованных!

res = diff(10, y=3)

**Необязательные аргументы. Значения по умолчанию**

Бывает, что какой-то параметр функции часто принимает одно и то же значение. Например, для функции print() создатели языка Python установили значения параметров sep и end равными символу пробела и символу перевода строки, поскольку эти значения используют наиболее часто.

Другим примером служит функция int() , преобразующая строку в число. Она принимает два аргумента:

* первый аргумент: строка, которую нужно преобразовать в число;
* второй аргумент: основание [системы счисления](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F).

Это позволяет ей считывать числа в различных системах счисления.

Приведенный ниже код, преобразует двоичное число 101101:

num = **int**('101', 2)

Но чаще всего эта функция используется для считывания из строки чисел, записанных в десятичной системе счисления. Утомительно каждый раз писать 10 вторым аргументом. В таких ситуациях Python позволяет задавать некоторым параметрам значения по умолчанию. У функции int() второй параметр по умолчанию равен 10, и потому можно вызывать эту функцию с одним аргументом. Значение второго подставится автоматически.

Чтобы задать значение параметра по умолчанию, в списке параметров функции достаточно после имени переменной написать знак равенства и нужное значение.

Параметры со значением по умолчанию идут последними, ведь иначе интерпретатор не смог бы понять, какой из аргументов указан, а какой пропущен, и для него нужно использовать значение по умолчанию.

Рассмотрим определение функции make\_circle(), которая рисует круг:

**def** make\_circle(x, y, radius, line\_width, fill):

# тело функции

Поскольку обычно нам нужно рисовать круг с шириной линии, равной 1 с заливкой, то логично установить данные значения в качестве значений по умолчанию:

**def** make\_circle(x, y, radius, line\_width=1, fill=True):

# тело функции

Теперь для того, чтобы нарисовать стандартный круг, то есть круг имеющий ширину линии, равную 1 с заливкой, мы вызываем функцию так:

make\_circle(100, 50, 20)

или так:

make\_circle(x=100, y=50, radius=20)

Если вам хочется поменять ширину линии и заливку, то вы легко можете это сделать:

make\_circle(x=100, y=50, radius=20, fill=**False**)

make\_circle(x=100, y=50, radius=20, line\_width=3)

make\_circle(x=100, y=50, radius=20, line\_width=5, **fill=False**)