# Урок 17.02. Передача изменяемых и неизменяемых данных в функцию

Мы с вами уже знаем по сути все необходимые для работы типы данных: целые числа, вещественные числа, булевы, строки и базовые коллекции (списки, словари, множества, кортежи). Также мы знаем, что ни делятся на изменяемые и неизменяемые типы. И сегодня мы чуть больше узнаем, что это значит.

Рассмотрим такой пример.

Есть список из трех чисел и переменная, в которой хранится какое-то значение, например, 100. Далее вызываем функцию и передаем её список и переменную. В функции мы увеличиваем все значения на 10. При этом из функции мы ничего не возвращаем.

def change\_values(some\_list, number):

for i\_index, i\_value in enumerate(some\_list):

some\_list[i\_index] = i\_value + 10

number += 10

my\_list = [1, 2, 3]

num = 100

change\_values(my\_list, num)

print(my\_list, num)

Тогда, как мы уже знаем, значения чисел в списке изменится, а значение переменной не изменится.

И теперь нам пришло время узнать страшную тайну, что в python нет переменных. И когда мы создаем переменную и присваиваем ей какое-то значение, то используются имена и ссылки на определенные объекты, а мы для простоты использовали устоявшийся термин переменная. Тогда в нашей программе num – это имя, которое ссылается на объект целочисленного типа, т.е. на число 100, также объекту присваивается уникальный идентификатор.

Вот так это выглядит в памяти python.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Object | |
| Python Name |  | Id | **<…>** |
| **num** |  | Type | **Integer** |
|  |  | Value | **100** |
|  |  | Reference count | **1** |

Давайте теперь присвоим num новое значение, например, 200. В таком случае в памяти поменяется ссылка num на другой объект, и вот так будет выглядеть в памяти.

num = 200

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Object | |
| Python Name |  | Id | **<…>** |
| **num** |  | Type | **Integer** |
|  |  | Value | **100** |
|  |  | Reference count | **0** |
|  |  |  |  |
|  |  | Object | |
|  |  | Id | **<…>** |
|  |  | Type | **Integer** |
|  |  | Value | **200** |
|  |  | Reference count | **1** |

И теперь у нас останется объект со счетчиком ссылок равным 0 (объект «100»), и он будет удален за не надобностью.

С именами и ссылками мы разобрались. Теперь поговорим про другие типы данных.

Переменная num ссылалось на не изменяемый тип данных, по-английски immutable. К ним относят: int, float, str, tuple, bool. И неизменяемые типы данных передаются в функцию по значению, т.е. передавая в функцию переменную num мы передавали не ссылку на весь объект «100», а только его значение, а внутри функции внутренней переменной number присваивалось значение 100 или, как мы уже знаем, создавался новый объект с этим значением. Поэтому num никак не изменялся.

Теперь мы переходим к изменяемым типам данных (по-английски mutable). К ним относятся: list, dict, set. И тут начинается самое интересное, как мы уже знаем, в функцию передается не сам список (его значения), а передается ссылка на этот список, и тогда в функции мы уже имеем доступ к непосредственно значениям этого списка.

Давайте попробуем это изучить на практике.

x = 10

y = x

print(id(x), id(y), id(10))

x += 1

print(id(x), id(y))

Этот пример демонстрирует, какие идентификаторы (id) у нас получаются у переменных при их присваивании и пере присваивании.

Теперь попробуем тоже самое с list.

list\_1 = [1, 2, 3]

list\_2 = list\_1

print(id(list\_1), id(list\_2))

list\_2.append(100)

list\_2 += [100]

print(id(list\_1), id(list\_2))

print(list\_1, list\_2)

Этот код уже демонстрирует, что присвоив list\_2 ссылку на list\_1, мы изменяем один и тот же список, но только используем разные имена, т.к. имя привязано к одному и тому же адресу.

Поэтому нужно очень внимательно подходить к выбору типа данных и их работе в функциях. Ведь очень много багов возникают при неправильном использовании изменяемых и неизменяемых типов данных.

Но что же нам делать, если нужно изменить список внутри функции, но не менять в месте откуда был вызов. Для этого есть функция копирования – copy. Давайте изменим нашу первоначальную программу.

def change\_values(some\_list, number):

some\_list = some\_list.copy()

for i\_index, i\_value in enumerate(some\_list):

some\_list[i\_index] = i\_value + 10

number += 10

print(some\_list, number)

my\_list = [1, 2, 3]

num = 100

change\_values(my\_list, num)

print(my\_list, num)

Таким способом нам удалось это реализовать. Такую же копию можно создать и для других изменяемых типов данных.

Однако бывают вложенные списки и тут наше копирование уже не поможет. Давайте это продемонстрируем.

def change\_values(some\_list):

some\_list = some\_list.copy()

for i\_index, i\_list in enumerate(some\_list):

for j\_index, j\_number in enumerate(i\_list):

some\_list[i\_index][j\_index] = j\_number + 10

print(f'список в функции вызова: {list\_1}')

list\_1 = [[1, 2, 3], [4, 5, 6]]

print(f'начальный список: {list\_1}')

change\_values(list\_1)

print(f'список после вызова функции: {list\_1}')

Запустим и видим, что копирование нам не помогло, думаю вы догадались почему (скопировались ссылки на внутренние списки, но не на значения). В таких случаях в python есть функция «глубокого копирования» (deep copy) данных. Эта функция расположена в стандартной библиотеке copy.

import copy

def change\_values(some\_list):

some\_list = copy.deepcopy(some\_list)

for i\_index, i\_list in enumerate(some\_list):

for j\_index, j\_number in enumerate(i\_list):

some\_list[i\_index][j\_index] = j\_number + 10

print(f'список в функции вызова: {list\_1}')

list\_1 = [[1, 2, 3], [4, 5, 6]]

print(f'начальный список: {list\_1}')

change\_values(list\_1)

print(f'список после вызова функции: {list\_1}')