# Lecture-32

**Packing and unpacking of arguments in Python:**

**Using the Python args Variable in Function Definitions**

*args can be really useful, because it allows you to pass a varying number of positional arguments. Take the following example:

```
# sum_integers_args.py
def my_sum(*args):
    result = 0
    # Iterating over the Python args tuple
    for x in args:
        result += x
    return result
 print(my_sum(1, 2, 3))
```

In this example, you’re no longer passing a list to my_sum(). Instead, you’re passing three different positional arguments. my_sum() takes all the parameters that are provided in the input and packs them all into a single iterable object named args.

Note that **args is just a name.** You’re not required to use the name args. You can choose any name that you prefer, such as integers:

```
# sum_integers_args_2.py
def my_sum(*integers):
    result = 0
    for x in integers:
        result += x
    return result
print(my_sum(1, 2, 3))
```

The function still works, even if you pass the iterable object as integers instead of args. All that matters here is that you use the **unpacking operator** (*).

Bear in mind that the iterable object you'll get using the unpacking operator * is not a list but a tuple. A tuple is similar to a list in that they both support slicing and iteration. However, tuples are very different in at least one aspect: lists are mutable, while tuples are not. To test this, run the following code. This script tries to change a value of a list:

```
# change_list.py

my_list = [1, 2, 3]

my_list[0] = 9

print(my_list)
```

The value located at the very first index of the list should be updated to 9. If you execute this script, you will see that the list indeed gets modified:

```
$ python change_list.py
[9, 2, 3]
```

The first value is no longer 0, but the updated value 9. Now, try to do the same with a tuple:

```
# change_tuple.py
my_tuple = (1, 2, 3)
my_tuple[0] = 9
print(my_tuple)
```

Here, you see the same values, except they're held together as a tuple. If you try to execute this script, you will see that the Python interpreter returns an error:

```
Traceback (most recent call last):
  File "change_tuple.py", line 3, in <module>
    my_tuple[0] = 9
TypeError: 'tuple' object does not support item assignment
```

This is because a tuple is an immutable object, and its values cannot be changed after assignment. Keep this in mind when you’re working with tuples and *args.

## Using the Python kwargs Variable in Function Definitions

Okay, now you’ve understood what *args is for, but what about **kwargs? **kwargs works just like *args, but instead of accepting positional arguments it accepts keyword (or **named**) arguments. Take the following example:

```
# concatenate.py
def concatenate(**kwargs):
    result = “ "
    # Iterating over the Python kwargs dictionary
    for arg in kwargs.values():
        result += arg
    return result
 print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

When you execute the script above, concatenate() will iterate through the Python kwargs dictionary and concatenate all the values it finds:

```
RealPythonIsGreat!
```

Like args, kwargs is just a name that can be changed to whatever you want. Again, what is important here is the use of the **unpacking operator** (**).

So, the previous example could be written like this:

```
# concatenate_2.py

def concatenate(**words):

    result = ""

    for arg in words.values():

        result += arg

    return result

 print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

Note that in the example above the iterable object is a standard dict. If you iterate over the dictionary and want to return its values, like in the example shown, then you must use .values().

In fact, if you forget to use this method, you will find yourself iterating through the **keys** of your Python kwargs dictionary instead, like in the following example:

```
def concatenate(**kwargs):
    result = ""
    # Iterating over the keys of the Python kwargs dictionary
    for arg in kwargs:
        result += arg
    return result
 print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

Now, if you try to execute this example, you'll notice the following output:

```
abcde
```

As you can see, if you don't specify .values(), your function will iterate over the keys of your Python kwargs dictionary, returning the wrong result.

## Unpacking With the Asterisk Operators: * & **

You are now able to use *args and **kwargs to define Python functions that take a varying number of input arguments. Let’s go a little deeper to understand something more about the **unpacking operators**.

The single and double asterisk unpacking operators were introduced in Python 2. As of the 3.5 release, they have become even more powerful, thanks to PEP 448. In short, the unpacking operators are operators that unpack the values from iterable objects in Python. The single asterisk operator * can be used on any iterable that Python provides, while the double asterisk operator ** can only be used on dictionaries.

Let’s start with an example:

```
my_list = [1, 2, 3]

print(my_list)
```

This code defines a list and then prints it to the standard output:

```
[1, 2, 3]
```

Note how the list is printed, along with the corresponding brackets and commas.

Now, try to prepend the unpacking operator * to the name of your list:

```
# print_unpacked_list.py

my_list = [1, 2, 3]

print(*my_list)
```

Here, the * operator tells print() to unpack the list first.

In this case, the output is no longer the list itself, but rather *the content* of the list:

```
1 2 3
```

Can you see the difference between this execution and the one from print_list.py? Instead of a list, print() has taken three separate arguments as the input.

Another thing you'll notice is that in print_unpacked_list.py, you used the unpacking operator * to call a function, instead of in a function definition. In this case, print() takes all the items of a list as though they were single arguments.

You can also use this method to call your own functions, but if your function requires a specific number of arguments, then the iterable you unpack must have the same number of arguments.

To test this behavior, consider this script:

```
def my_sum(a, b, c):
    print(a + b + c)
 my_list = [1, 2, 3]
my_sum(*my_list)
```

Here, my_sum() explicitly states that a, b, and c are required arguments.

If you run this script, you’ll get the sum of the three numbers in my_list:

OUTPUT:

```
6
```

The 3 elements in my_list match up perfectly with the required arguments in my_sum().

Now look at the following script, where my_list has 4 arguments instead of 3:

```
# wrong_unpacking_call.py
def my_sum(a, b, c):
    print(a + b + c)
 my_list = [1, 2, 3, 4]
my_sum(*my_list)
```

In this example, my_sum() still expects just three arguments, but the * operator gets 4 items from the list. If you try to execute this script, you’ll see that the Python interpreter is unable to run it:

```
$ python wrong_unpacking_call.py
Traceback (most recent call last):
  File "wrong_unpacking_call.py", line 6, in <module>
    my_sum(*my_list)
TypeError: my_sum() takes 3 positional arguments but 4 were given
```

When you use the * operator to unpack a list and pass arguments to a function, it’s exactly as though you’re passing every single argument alone. This means that you can use multiple unpacking operators to get values from several lists and pass them all to a single function.

To test this behavior, consider the following example:

```
# sum_integers_args_3.py
def my_sum(*args):
    result = 0
    for x in args:
        result += x
    return result
 list1 = [1, 2, 3]
list2 = [4, 5]
list3 = [6, 7, 8, 9]
 print(my_sum(*list1, *list2, *list3))
```

If you run this example, all three lists are unpacked. Each individual item is passed to my_sum(), resulting in the following output:

```
45
```

## Merging Lists:

Another interesting thing you can do with the unpacking operator * is to split the items of any iterable object. This could be very useful if you need to merge two lists, for instance:

```
my_first_list = [1, 2, 3]
my_second_list = [4, 5, 6]
my_merged_list = [*my_first_list, *my_second_list]
 print(my_merged_list)
```

The unpacking operator * is prepended to both my_first_list and my_second_list.

If you run this script, you'll see that the result is a merged list:

**OUTPUT:**

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

## Unpacking string:

Remember that the * operator works on *any* iterable object. It can also be used to unpack a string:

```
# string_to_list.py

a = [*"RealPython"]

print(a)
```

In Python, strings are iterable objects, so * will unpack it and place all individual values in a list a:

```
$ python string_to_list.py

['R', 'e', 'a', 'l', 'P', 'y', 't', 'h', 'o', 'n']
```

To see why, consider the following example:

```
*a, = "RealPython"

print(a)
```

## ** is used for dictionaries unpacking

```
# A sample program to demonstrate unpacking of
# dictionary items using **
def fun(a, b, c):
    print(a, b, c)
  # A call with unpacking of dictionary
d = {'a':2, 'b':4, 'c':10}
fun(**d)
```

**Output-**

```
2   4    10
```

Here ** unpacked the dictionary used with it, and passed the items in the dictionary as keyword arguments to the function. So writing “fun(1, **d)” was equivalent to writing “fun(1, b=4, c=10)”.

## Packing:

When we don’t know how many arguments need to be passed to a python function, we can use Packing to pack all arguments in a tuple.

```
# A Python program to demonstrate use of packing
 # This function uses packing to sum unknown number of arguments
def mySum(*args):
   sum = 0
   for i in range(0, len(args)):
      sum = sum + args[i]
   return sum
print(mySum(1, 2, 3, 4, 5))
print(mySum(10, 20))
```

**Output:**

```
15

30
```

The above function mySum() does ‘packing’ to pack all the arguments that this method call receives into one single variable. Once we have this ‘packed’ variable, we can do things with it that we would with a normal tuple. args[0] and args[1] would give you the first and second argument, respectively. Since our tuples are immutable, you can convert the args tuple to a list so you can also modify, delete and re-arrange items in i.

**# A Python program to demonstrate packing of dictionary items using ****

```
def fun(**kwargs):
      # kwargs is a dict
    print(type(kwargs))
      # Printing dictionary items
    for key in kwargs:
        print("%s = %s" % (key, kwargs[key]))
  # Driver code
fun(name="geeks", ID="101", language="Python")
```

**Output :**

```
<class 'dict'>
language = Python
name = geeks
ID = 101
```

## Merging of dictionaries:

You can even merge two different dictionaries by using the unpacking operator **:

```
# merging_dicts.py
my_first_dict = {"A": 1, "B": 2}
my_second_dict = {"C": 3, "D": 4}
my_merged_dict = {**my_first_dict, **my_second_dict}
 print(my_merged_dict)
```

Here, the iterables to merge are my_first_dict and my_second_dict.

Executing this code outputs a merged dictionary:

**OUTPUT:**

```
{'A': 1, 'B': 2, 'C': 3, 'D': 4}
```

## References:

1.Introduction to Computation and Programming using Python, by John Guttag, PHI Publisher

2. Fundamentals of Python first Programmes by Kenneth A Lambert, Copyrighted material Course Technology Inc. 1 st edition (6th February 2009)

3. https://www.tutorialspoint.com/python/index.htm

4. https://www.geeksforgeeks.org/python-programming-language

5. https://www.w3schools.com/python/

****END OF THE LECTURE***

***THANK YOU***