Python Fundamentals day 7

Today's Agenda

- Types of arguments
- Positional arguments
- Default arguments
- Keyword arguments
- Variable length arguments
- Variable keyword length arguments

Types of arguments

The input passed through a function is technically called as **argument**. Let us know better

Parameters

```
def mul(x,y):
    c=x*y
    print(c)

mul(10,20)
```

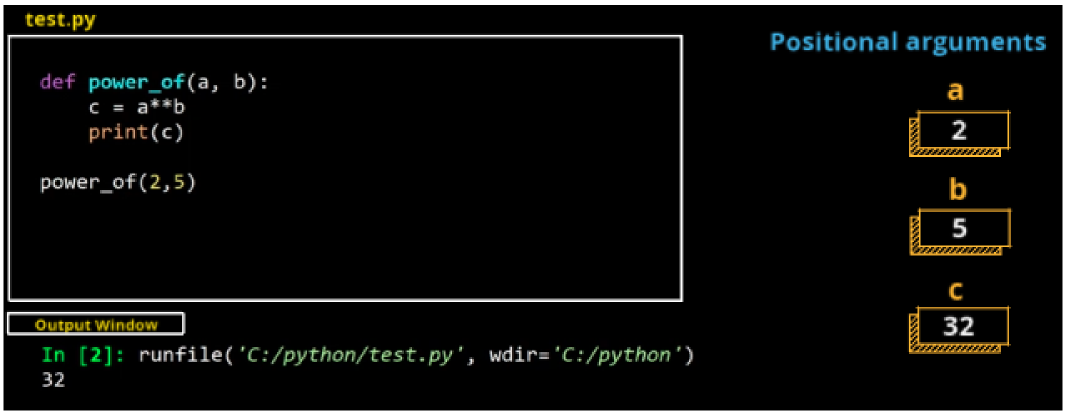
Arguments

Parameters collect the inputs, which is passed to the function when it is called.

We have different types of arguments:

1. Positional arguments
2. Default arguments (Optional arguments)
3. Keyword arguments
4. Variable length arguments (Arbitrary arguments)
5. Variable keyword length arguments (Arbitrary keyword arguments)

Positional Arguments



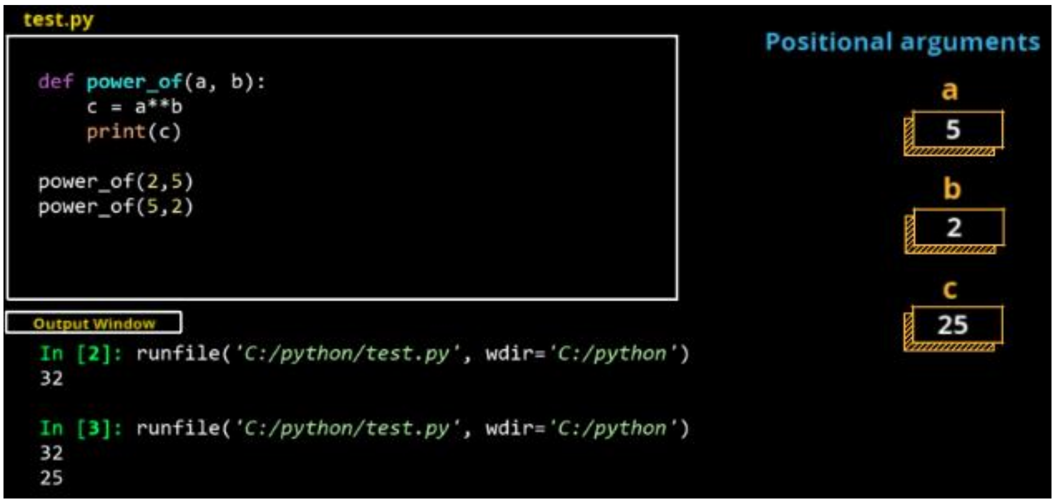
test.py

```
def power_of(a, b):
    c = a**b
    print(c)

power_of(2,5)
```

Output Window

```
In [2]: runfile('C:/python/test.py', wdir='C:/python')
32
```

In the above example we are performing power operation. Here the arguments are 2,5 and parameters are a,b where a gets assigned as 2 and b gets assigned as 5 based on the positions. To verify this let's try to interchange the positions and check whether the value of c changes or remains same.



test.py

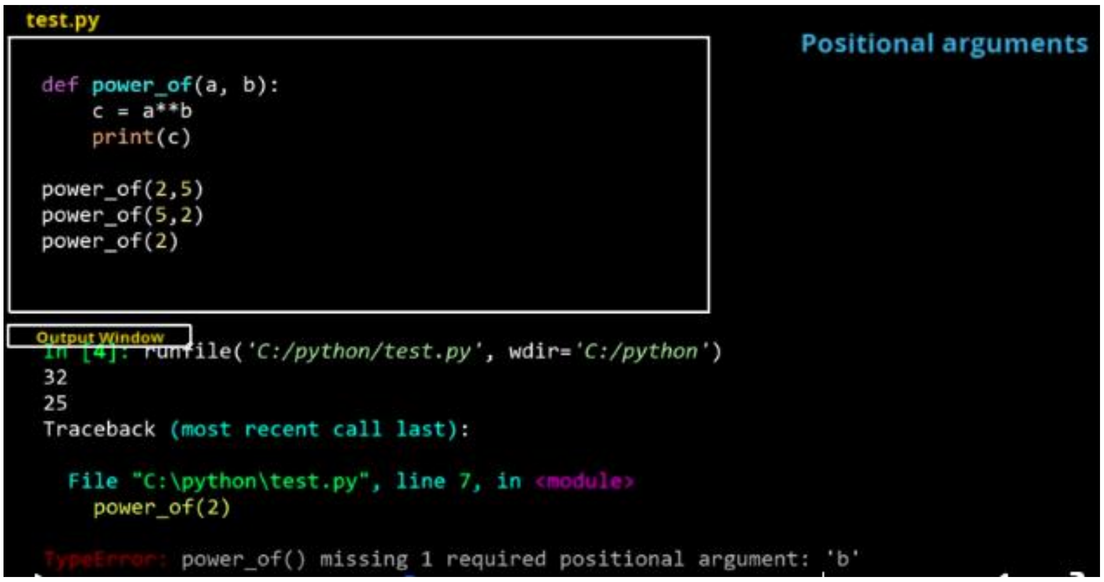
```
def power_of(a, b):
    c = a**b
    print(c)

power_of(2,5)
power_of(5,2)
```

Output Window

```
In [2]: runfile('C:/python/test.py', wdir='C:/python')
32

In [3]: runfile('C:/python/test.py', wdir='C:/python')
32
25
```

As we can see, a gets assigned as 5 and b gets assigned as 2. Which proves the above statement, if passed in the above format, position definitely matters.

If we miss a single argument then error appears, let us see what the error is

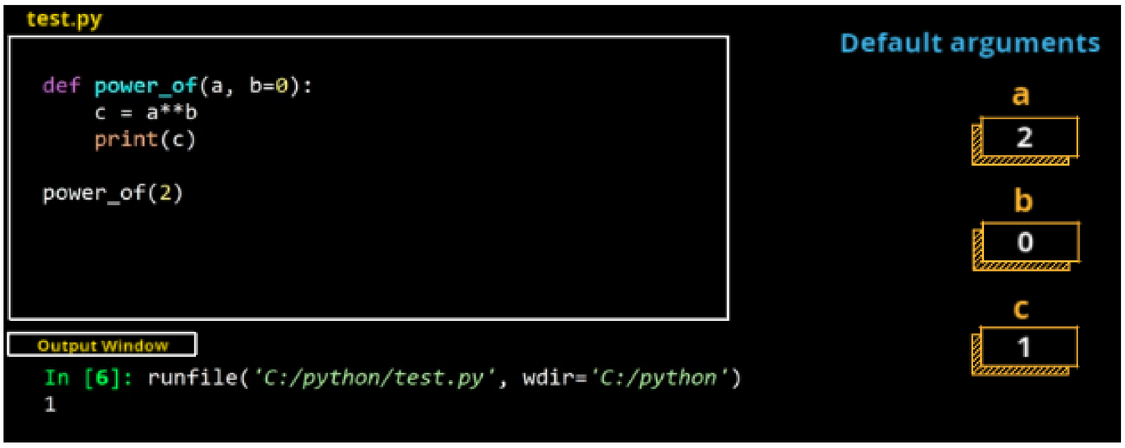


```
test.py
def power_of(a, b):
    c = a**b
    print(c)

power_of(2,5)
power_of(5,2)
power_of(2)
```

```
In [4]: runfile('C:/python/test.py', wdir='C:/python')
32
25
Traceback (most recent call last):

  File "C:\python\test.py", line 7, in <module>
    power_of(2)

TypeError: power_of() missing 1 required positional argument: 'b'
```

Error clearly tells us that we are missing an argument b. So now let us see how to get away with this.

Default Arguments



```
test.py
def power_of(a, b=0):
    c = a**b
    print(c)

power_of(2)
```

```
In [6]: runfile('C:/python/test.py', wdir='C:/python')
1
```

In the above example we have only single argument, assuming b will take the default value assigned to it as 0. Giving a value while declaring the parameter is referred to as default value.

Certainly we can also give two arguments

```
def power_of(a,b=0):
    c=a**b
    print(c)

power_of(2)
power_of(2,5)
```

Output:

```
In [30]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
1
32
```

Let us see another example

```
def fun(a,b=0,x):
    c=a*b*x
    print(c)

fun(5,3,2)
```

Here we have third argument. Let us see if we get the output as expected, that is 5*3*2 = 30

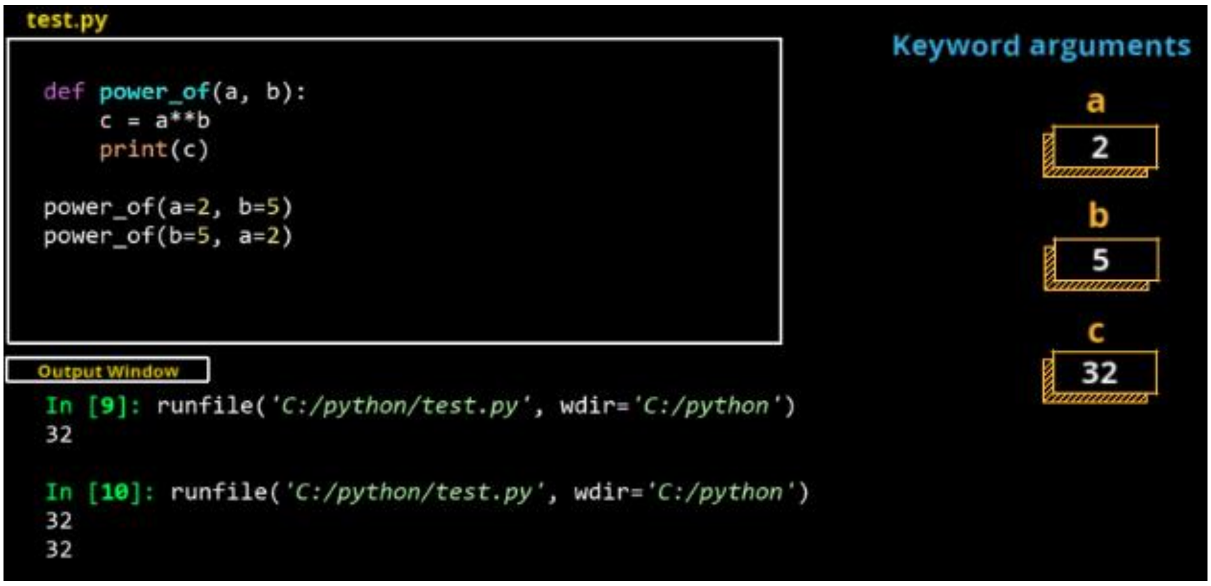
Output:

```
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 1
    def fun(a,b=0,x):
                 ^
SyntaxError: non-default argument follows default argument
```

As the message reflected, non-default argument cannot follow default argument. But certainly after positional arguments we can have as many default arguments as possible.

Keyword arguments

In positional arguments we noticed that if order changes the output also changes. We can overcome this by using keyword arguments, let us see how



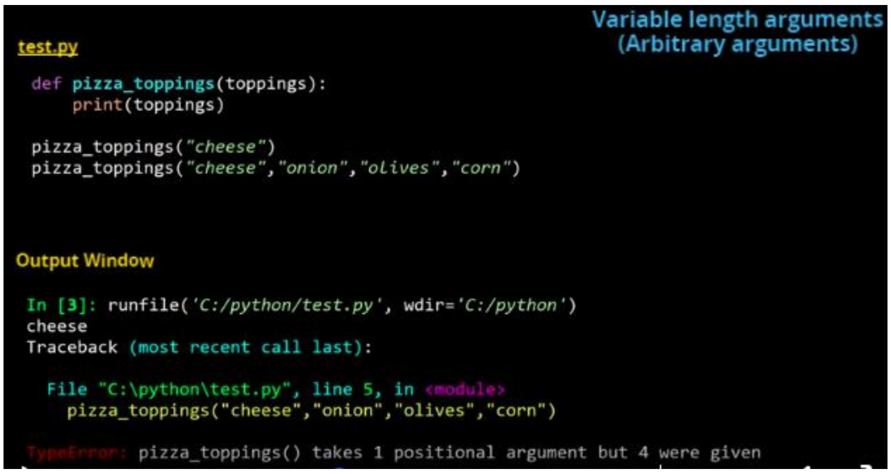
```
def power_of(a, b):
    c = a**b
    print(c)

power_of(a=2, b=5)
power_of(b=5, a=2)
```

```
In [9]: runfile('C:/python/test.py', wdir='C:/python')
32

In [10]: runfile('C:/python/test.py', wdir='C:/python')
32
32
```

We can see that, if we mention the keywords while passing arguments the output does not depend on the order or the position of the arguments. This is the advantage of using keyword arguments.

Variable length argument

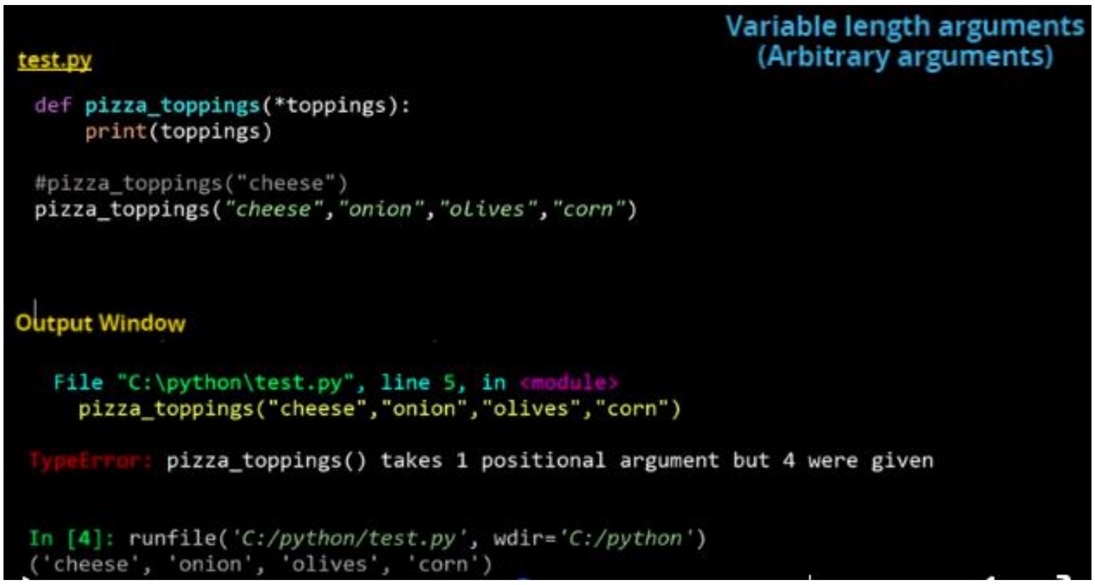
Passing any number of arguments to a function is called as variable length argument. Let us know better by an example



```
def pizza_toppings(toppings):
    print(toppings)

pizza_toppings("cheese")
pizza_toppings("cheese","onion","olives","corn")
```

```
In [3]: runfile('C:/python/test.py', wdir='C:/python')
cheese
Traceback (most recent call last):

  File "C:\python\test.py", line 5, in <module>
    pizza_toppings("cheese","onion","olives","corn")

TypeError: pizza_toppings() takes 1 positional argument but 4 were given
```

In the above example we are trying to pass different number of arguments but only first argument is what the function is taking, so certainly some changes in function declaration is needed, let us see what is the change and how that change will accept multiple arguments



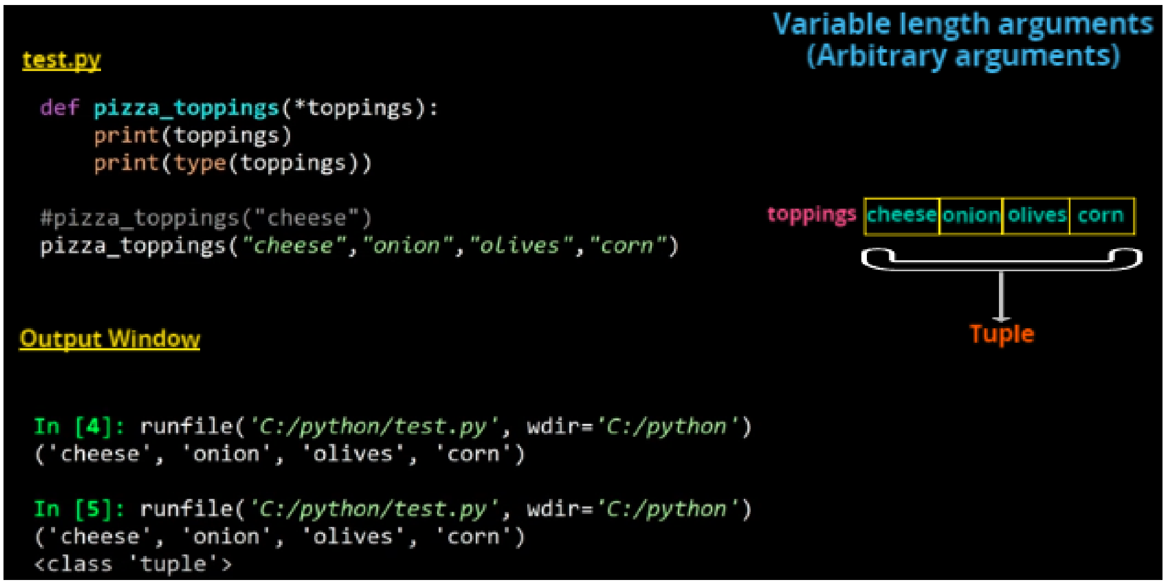
test.py

```
def pizza_toppings(*toppings):
    print(toppings)

#pizza_toppings("cheese")
pizza_toppings("cheese","onion","olives","corn")
```

Output Window

```
  File "C:\python\test.py", line 5, in <module>
    pizza_toppings("cheese","onion","olives","corn")

TypeError: pizza_toppings() takes 1 positional argument but 4 were given


In [4]: runfile('C:/python/test.py', wdir='C:/python')
('cheese', 'onion', 'olives', 'corn')
```

We can now see that by attaching a star or asterisk in front of the parameter, the function is now ready to take different number of arguments. This is because toppings is now considered as **tuple**. Let us verify that by printing the type of toppings



test.py

```
def pizza_toppings(*toppings):
    print(toppings)
    print(type(toppings))

#pizza_toppings("cheese")
pizza_toppings("cheese","onion","olives","corn")
```

Output Window

```
In [4]: runfile('C:/python/test.py', wdir='C:/python')
('cheese', 'onion', 'olives', 'corn')

In [5]: runfile('C:/python/test.py', wdir='C:/python')
('cheese', 'onion', 'olives', 'corn')
<class 'tuple'>
```

Note: * in front of the parameter makes it a tuple. But while calling for it we should just enter name which does not contain *.

Let us see another case

```
def pizza_toppings(*toppings,crust):
    print(toppings)
    print(crust)

pizza_toppings("cheese",crust="thin")
```

Output:

```
In [32]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
('cheese',)
thin
```

In the above example we are trying to pass another argument. But crust is a positional argument/non-default argument. So if we use keyword and assign the value, the output is as expected. But what if we don't use the keyword?? Let's see

```
def pizza_toppings(*toppings,crust):
    print(toppings)
    print(crust)

pizza_toppings("cheese","thin")
```

Output:

```
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 5,
in <module>
    pizza_toppings("cheese","thin")

TypeError: pizza_toppings() missing 1 required keyword-only
argument: 'crust'
```

Here we see that cheese and thin both are considered as toppings. So if we are passing any arguments before or after the variable length argument it is mandatory to use keyword arguments.

Variable keyword length arguments

In the previous example we saw how to pass different number of arguments, but all of them were toppings. What if each argument we pass represents different data. Let us see how to resolve this

```
def collect_student_data(*data):
    print(data)


collect_student_data("Rohit",28,60.5,'M')
```

Output:

```
In [34]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
('Rohit', 28, 60.5, 'M')
```

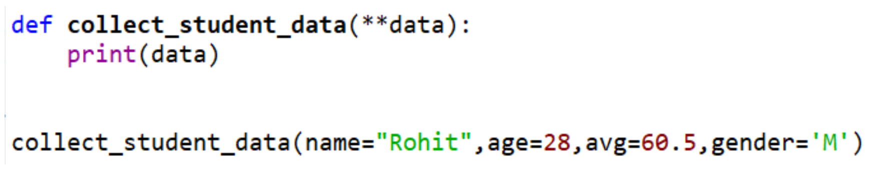
We certainly got the output, but we still can't recognise what is 28, 60.5, M. To resolve this we must give a key to each value entered as shown below

```
def collect_student_data(*data):
    print(data)


collect_student_data(name="Rohit",age=28,avg=60.5,gender='M')
```

Output:

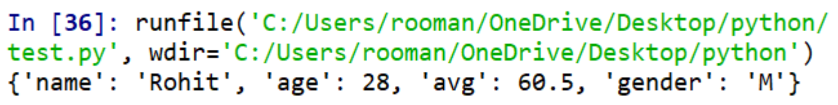
```
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 5,
in <module>
    collect_student_data(name="Rohit",age=28,avg=60.5,gender='M')

TypeError: collect_student_data() got an unexpected keyword
argument 'name'
```

The error states that it doesn't recognise name and similarly age, avg, gender. To resolve this we must make another change, let us see what that is, and how it resolves the issue



```
def collect_student_data(**data):
    print(data)


collect_student_data(name="Rohit",age=28,avg=60.5,gender='M')
```

Output:



```
In [36]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'age': 28, 'avg': 60.5, 'gender': 'M'}
```

All we did was add another * to the parameter.

And we can see in output that the keys and values are enclosed with { } which states ** in front of a parameter makes it a dictionary. Which can store different arguments along with keywords associated with it.

Here also we have the same rule as seen earlier that if we are passing any arguments before or after the variable keyword length argument it is mandatory to use keyword arguments.