Python *args

Summary: in this tutorial, you'll learn about the Python `*args` parameters and how to use them for defining variadic functions.

Tuple unpacking

The following unpacks a tuple (https://www.pythontutorial.net/python-basics/python-tuples/) into two variables (https://www.pythontutorial.net/python-basics/python-variables/) :

```
x, y = 10, 20
```

Python assigns 10 to x and 20 to y. It's similar to passing two arguments to a function:

```
def add(x, y):
    return x + y


add(10, 20)
```

In this example, Python passed `10` to `x` and `20` to `y` .

Similarly, the following assigns `10` to `x`, `20` to `y`, and the list (https://www.pythontutorial.net/python-basics/python-list/) `[30, 40]` to `z`:

```
x, y, *z = 10, 20, 30, 40

print(x)
print(y)
print(z)
```

Python uses the same concept for the function arguments. For example:

```
def add(x, y, *args):
    total = x + y
    for arg in args:
        total += arg

    return total


result = add(10, 20, 30, 40)
print(result)
```

The `add` function accepts three parameters `x`, `y`, and `*args`. The `*args` is a special argument preceded by a star (`*`).

When passing the positional arguments `10`, `20`, `30`, and `40` to the function, Python assigns `10` to `x`, `20` to `y`, and a tuple `(30, 40)` to `args`.

It's like tuple unpacking (https://www.pythontutorial.net/python-basics/python-unpacking-tuple/) except that the `args` is a tuple, not a list.

Introduction to the Python *args parameter

When a function has a parameter preceded by an asterisk (`*`), it can accept a variable number of arguments. And you can pass zero, one, or more arguments to the `*args` parameter.

In Python, the parameters like `*args` are called variadic parameters. Functions that have variadic parameters are called variadic functions.

Note that you don't need to name `args` for a variadic parameter. For example, you can use any meaningful names like `*numbers` , `*strings` , `*lists` , etc.

However, by convention, Python uses the `*args` for a variadic parameter.

Let's take a look at the following example:

```
def add(*args):
    print(args)


add()
```

Output:

```
()
```

The `add` function shows an empty tuple (https://www.pythontutorial.net/python-basics/python-tuples/) .

The following shows the type of the `args` argument and its contents:

```
def add(*args):
    print(type(args))
    print(args)

add()
```

Output

```
<class 'tuple'>
()
```

Since we didn't pass any argument to the `add()` function, the output shows an empty tuple.

The following passes three arguments to the `add()` function:

```
def add(*args):
    print(args)


add(1,2,3)
```

Output:

```
(1, 2, 3)
```

Now, the `args` has three numbers 1, 2, and 3. To access each element of the `args` argument, you use the square bracket notation `[]` with an index:

```
def add(*args):
    print(args[0])
    print(args[1])
    print(args[2])


add(1, 2, 3)
```

Also, you an use a `for` loop (https://www.pythontutorial.net/python-basics/python-for-loop-list/) to iterate over the elements of the tuple.

The following shows how to add all numbers of the `args` tuple in the `add()` function:

```
def add(*args):
    total = 0
    for arg in args:
        total += arg
    return total


total = add(1, 2, 3)
print(total)
```

Output:

```
6
```

Python *args argument exhausts positional arguments

If you use the `*args` argument, you cannot add more positional arguments. However, you can use keyword arguments (https://www.pythontutorial.net/python-basics/python-keyword-arguments/) .

The following example results in an error because it uses a positional argument after the `*arg` argument:

```
def add(x, y, *args, z):
    return x + y + sum(args) + z


add(10, 20, 30, 40, 50)
```

Error:

```
TypeError: add() missing 1 required keyword-only argument: 'z'
```

To fix it, you need to use a keyword argument after the `*args` argument as follows:

```
def add(x, y, *args, z):
    return x + y + sum(args) + z


add(10, 20, 30, 40, z=50)
```

In this example, Python assigns `10` to `x`, `20` to `y`, `(30,40)` to `args`, and `50` to `z`.

Unpacking arguments

The following `point` function accepts two arguments and returns a string representation of a point with x-coordinate and y-coordinate:

```
def point(x, y):
    return f'({x},{y})'
```

If you pass a tuple to the `point` function, you'll get an error:

```
a = (0, 0)
origin = point(a)
```

Error:

```
TypeError: point() missing 1 required positional argument: 'y'
```

To fix this, you need to prefix the tuple `a` with the operator `*` like this:

```
def point(x, y):
    return f'({x},{y})'
```

```
a = (0, 0)
origin = point(*a)
print(origin)
```

Output:

```
(0,0)
```

When you precede the argument `a` with the operator `*`, Python unpacks the tuple and assigns its elements to `x` and `y` parameters.

Summary

- Use Python `*arg` arguments for a function that accepts a variable number of arguments.
- The `*args` argument exhausts positional arguments so you can only use keyword arguments after it.