Theory: Args

27 minutes 11 / 11 problems solved

Start practicing

3410 users solved this topic. Latest completion was **about 2 hours ago**.

Functions have a flexible syntax in Python. We will find out what allows functions to accept a varying number of arguments and how to unpack iterable objects when calling a function.

§1. Multiple positional arguments

You might be surprised by the fact that everything we've done before with functions limited us in some way. For example, if we don't specify the defaults for arguments, we will always have to pass the exact number of values into such function. However, sometimes it's more convenient when the number of arguments varies. For example, if you are declaring a function that should find the sum of all values passed into it, you never know how many arguments a user might want to use. Let's start with a simple case and define a function with two parameters. It can be done as follows:

```python
def add(a, b):
    return a + b
```

This function makes us pass only two values, we can't just do `add(1, 2, 3)`. Well, what we can do is to set a default value for the third parameter and then call this function with either two or three values. But this hardly solves the problem for more complex cases.

If you are not sure about the number of arguments that your function might take, or if you don't want to limit them, use the following syntax to define a function with `*args`:

```python
def add(a, b, *args):
    total = a + b
    for n in args:
        total += n
    return total
```

This allows you to work with the variable `args`, which is a tuple of remaining *positional* arguments. Its length may vary:

```python
# The length of `args` is 3
print(add(1, 2, 3, 4, 5))

# The length of `args` is 0
print(add(1, 2))
```

The function `add()` now requires two arguments, but if you pass additional values they will be collected in a tuple and get added to the total.

As you might have guessed, `args` is short for "arguments". You don't have to use this conventional name all the time, though:

```python
def will_survive(*names):
    for name in names:
        print("Will", name, "survive?")


will_survive("Daenerys Targaryen", "Arya Stark", "Brienne of Tarth")
```

The output for this function call will be as follows:

```
Will Daenerys Targaryen survive?
Will Arya Stark survive?
Will Brienne of Tarth survive?
```

This works for any variable name as long as there is a single asterisk `*` right before it.

Current topic:

✓ Args

Topic depends on:

✓ Tuple

✓ For loop Stage 1 12★

✓ Default arguments

Topic is required for:

✓ Kwargs

Table of contents:

Normally, `*args` comes after specific parameters:

```
def func(positional_args, defaults, *args):
    pass
```

Once all required arguments have been passed, the remaining values are packed into the tuple.

The parameters that come after `*args` are ***keyword-only***. It means that they can only be used as keywords rather than positional arguments.

```
def recipe(*args, sep=" or "):
    return sep.join(args)


print(recipe("meat", "fish"))               # meat or fish
print(recipe("meat", "fish", sep=" and "))  # meat and fish
```

§2. Unpacking in function calls

The Python syntax enables us to pass all items from a sequence as individual positional arguments using `*`. A single asterisk operator ***unpacks*** an iterable. Let's invoke the `print()` function and see how it works:

```
print(*"fun")        # f u n
print(*[5, 10, 15])  # 5 10 15
```

This code will be equivalent to a call where elements are listed one by one: `print("f", "u", "n")` and `print(5, 10, 15)` respectively. Unpacking just takes less of your time.

Combined with `*args` in our slightly modified function `add()`, unpacking takes away the concern for the number of values both in the function's body and upcoming calls.

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


small_numbers = [1, 2, 3]
large_numbers = [9999999, 1111111]


print(add(*small_numbers))  # 6

print(add(*large_numbers))  # 11111110
```

This is really powerful as it allows you to conveniently handle an arbitrary number of values in your function.

§3. Recap

Let's sum up what we discussed in the topic:

- A function with `*args` can accept a changing number of **positional** arguments.
- The variable name `args` is **arbitrary**, you can always choose another one.
- `*args` provides access to a **tuple** of remaining values.
- The **order of parameters** in the function definition is important, as well as the **order of passed arguments**.
- In function calls, you can use the **unpacking operator** `*` for iterable objects.

Report a typo

Feedback sent!

Start practicing

Comments (15) Hints (0) Useful links (4) Show discussion