teclado

ADVANCED PYTHON

Day 17: Flexible Functions with *args and **kwargs

Welcome to day 17 of the 30 Days of Python series! Today we're going to be talking about how use the `*` and `**` operators to make truly flexible functions that can accept any number of arguments.

The problem

Some of you may have noticed that several of the built in functions accept a variable number of arguments. For example, let's look at good old `print`.

`print` can actually take an arbitrary number of positional arguments,

and it will print them all side by side by default:

```
print(1, 2, 3, 4, 5)  # 1 2 3 4 5
```

The character that gets put between these values is controlled by another parameter called `sep`, which has a default value of `" "`. That's why we get a single space between each item.

We can write this explicitly like this:

```
print(1, 2, 3, 4, 5, sep=" ")  # 1 2 3 4 5
```

We have another post which goes into more detail on `print` if you're interested.

While this behaviour seems relatively simple, this is not something we can replicate right now. We have to provide an argument for every parameter, and conversely every argument must have a corresponding parameter as well. This means we need to know how many values we're going to accept in advance.

So, how we do something like `print`, where we can just accept as many arguments as the user provides?

Accepting an arbitrary number of positional arguments

Let's start by looking at just positional arguments, since this is the behaviour we observe in `print`.

The way we accept any number of positional arguments is by defining a special kind of parameter which has a `*` right before its name. To be very clear, this `*` is actually an operator, and not part of the parameter name.

Generally speaking we call this special parameter `args`, which is short for arguments.

When we include a `*` parameter, Python is going to gather up any unassigned positional arguments when we the call the function, and it's going to put them all in a tuple. This tuple is what gets assigned to our `*` parameter.

Let's look at an example so we can see this in action.

We're going to define a function called `mul`, which is going to take in any two numbers as arguments (with two parameters), and multiply them. Then we'll re-define it using `*args`.

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

Here we can call the function like so, passing two positional arguments. Remember these are positional arguments because the position of the argument is what matters when assigning values to parameters.

```
mul(5, 10)
```

Since `5` is in first position, it gets assigned to the `x` parameter.

Now we will use `*args` to define the function. The parameter will collect any positional arguments that aren't assigned to anything else, and make them into a tuple of values. Now we'll have to do this:

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

mul(5, 10)
```

It's rare to create functions like the one above, because the purpose of `*args` is to accept any number of arguments. Since we're accessing `args[0]` and `args[1]`, that means we really want *two* arguments. Therefore, using `*args` is not the right choice there.

But have a look at this other function, that accepts any number of friend names and greets them all:

```
def multigreet(*args):
        for name in args:
                print(f"Hello, {name}!")

multigreet("Rolf", "Bob", "Anne")
```

Try calling the function yourself, passing in as many names as you like as arguments.

One thing to be very aware of, is that when we refer to our parameter in the loop, we use the name `args`, not `*args`. Always remember that this `*` is an operator.

Style note

As I mentioned before, it's a convention to use the parameter names, `args`, when we want to gather up an excess positional arguments; however, this isn't always the right choice.

Writing `*args` is very useful when the arguments can be anything, but in the function above, we have a very good idea of what the values represent: the values should all be names.

In the case of `multigreet`, therefore more appropriate to use `*names` instead, like this:

```
def multigreet(*names):
        for name in names:
                print(f"Hello, {name}!")
```

Don't use `*args` if you have a better name. Remember that variable names are an excellent tool for aiding readability, so always use the most descriptive name you can.

Parameter order with `*args`

When we use a parameter like `*args` we have to be very aware of the order of our parameters. This is because any parameters we define after the `*args` cannot accept positional arguments.

This makes some sense. If we're gathering up all the excess positional arguments with our `*args` parameter, then there's nothing left by time we get to the parameters that come after it. All we have remaining are keyword arguments.

For example, if we change our `multigreet` function to the code below and try to call it,

```
def multigreet(*names, other):
        for name in names:
                print(f"Hello, {name}!")

multigreet("Jose", "Phil")
```

we get a `TypeError` .

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    multigreet("Jose", "Phil")
TypeError: multigreet() missing 1 required keyword-only argum
ent: 'other'
```

In this case we get a very clear message. We have a missing *keyword-only* argument.

If we instead put the `other` parameter first, this exception goes away. Python is more than happy to assign to the positional arguments in order, and then gather up what's left for our `*names` parameter.

Placing arguments after the `*` parameter can also be very useful, because we can force users to use keyword arguments for certain parameters. This is useful for things we don't want user to accidentally change, or things for which we want to have established defaults.

`print` actually makes use of this with `sep` and `end`, which only accept keyword arguments.

Accepting an arbitrary number of keyword arguments

We've talked about positional arguments, now it's time to talk about dealing with any number of keyword arguments. If you want an example from the built in functions, we can look at `dict`.

We've seen some different ways to call `dict`, but one that we haven't looked it is creating a dictionary using `dict` and keyword arguments.

For example, we can write this:

```
dict(name="Phil", age=29, city="Budapest", nationality="Briti
sh")
```

Here the keywords become keys, and the values we match to each keyword become the values associated with those keys. The code above is therefore the same as writing this:

```
{
        "name": "Phil",
        "age": 29,
        "city": "Budapest",
        "nationality": "British"
}
```

`dict` doesn't know in advance how many keys and values we want to create our dictionary with, so it has to be flexible and accept any number of keyword arguments.

We can achieve the same thing with another special parameter, this time with `**` in front of its name. The conventional name for this parameter is `**kwargs`, which is short for keyword arguments.

Take a look at this example, that prints out keyword arguments in a nice format:

```
def pretty_print(**kwargs):
        for key, value in kwargs.items():
                print(f"{key}: {value}")

pretty_print(title="The Matrix", director="Wachowski", year=1
999)

# title: The Matrix
# director: Wachowski
# year: 1999
```

Note that `**kwargs` collects all unassigned keyword arguments and creates a dictionary with them, that you can then use in your function. That's why we have access to `.items()` in there, because `kwargs` is a dictionary.

If we define both `*args` and `**kwargs` for a given function, `**kwargs` has to come second. If the order is reversed, Python considers it invalid syntax.

Really, `**kwargs` has to be the final parameter that you define, because it has to come after `*args` as well as after any other keyword arguments. After all, it's going to gather up any remaining keyword arguments, so any parameters that come after it will never get provided a value.

Style note

Just like with `*args`, we shouldn't be bound by convention when using `**kwargs`. If we know what data to expect, and we can more usefully describe it with another name, you definitely should do so.

The convention exists for situations where the arguments are completely unknown to us when defining the function.

Other uses for `*` and `**`

The `*` and `**` operators are very versatile, because we can not only use them for gathering up values into a single collection, we can also use them for the opposite: unpacking an iterable into individual values.

We've already looked at unpacking (also called destructuring) back in day 9, but we were always doing this in the context of assigning to a fixed number of variables. In those cases we simply matched the values provided by our iterable to the same number of variables, and Python took care of the rest.

However, sometimes this approach isn't always possible. For example, what if we want to destructure an iterable so that we can pass many values to `*args`?

The answer is we put a `*` before the iterable we're passing in as an argument.

For example, let's say I want to take this list of numbers and print the numbers on a single line with pipe (|) characters between each number. We could do this:

```
numbers = [1, 2, 3, 4, 5]

print(*numbers, sep=" | ")
```

If we run this code, we get the following output:

```
1 | 2 | 3 | 4 | 5
```

If we want to destructure a dictionary, we can do the same thing, but remember that by default, Python gives us the keys when we iterate over a dictionary. You're therefore often going to need to use the `values` or `items` methods when using `*`.

For example, this function prints details about a movie:

```
def print_movie(*args):
        for value in args:
                print(value)

movie = {
        "title": "The Matrix",
        "director": "Wachowski",
        "year": 1999
}

print_movie(*movie.values())

# The Matrix
# Wachowski
# 1999
```

Really the function is printing the arguments it receives, one on each line. If we pass in `movie.keys()` for example, it'll print the keys instead:

```
print_movie(*movie.keys())

# title
# director
# year
```

Not all that useful!

We can also use `**` to turn a dictionary into a series of keyword arguments.

Using the function above with `**kwargs` and dictionary destructuring, we get this:

```
def print_movie(**kwargs):
        for key, value in kwargs.items():
                print(f"{key}: {value}")

movie = {
        "title": "The Matrix",
        "director": "Wachowski",
        "year": 1999
}

print_movie(**movie)

# title: The Matrix
# director: Wachowski
# year: 1999
```

Here when we do `**movie` when calling the function, it turns the dictionary into keyword arguments. These are passed to `print_movie`,

and the `**kwargs` parameter collects them back into a dictionary.

You could just as well do this:

```
def print_movie(movie_details):
        for key, value in movie_details.items():
                print(f"{key}: {value}")

movie = {
        "title": "The Matrix",
        "director": "Wachowski",
        "year": 1999
}

print_movie(movie)

# title: The Matrix
# director: Wachowski
# year: 1999
```

And it would be valid, but `**kwargs` can give us more flexibility when it comes to collecting unassigned keyword arguments, and not only those coming from a dictionary.

For example, we could do this:

```
def print_movie(**kwargs):
        for key, value in kwargs.items():
                print(f"{key}: {value}")

movie = {
        "title": "The Matrix",
        "director": "Wachowski",
        "year": 1999
}

print_movie(studio="Warner Bros", **movie)
```

```
# studio: Warner Bros
# title: The Matrix
# director: Wachowski
# year: 1999
```

There we have used `**kwargs` to collect both the `studio` argument as well as all the `movie` arguments! It's not something we could do with just a single parameter all that easily.

There are plenty more uses for a technique like this, including merging dictionaries, and saving us a lot of typing when using `format`.

For example, let's look back at an example from the [day 14 project](), where we needed to print some output about our books from a list of dictionaries.

```
def show_books(books):
        # Adds an empty line before the output
        print()

        for book in books:
                print(f"{book['title']}, by {book['author']}
 ({book['year']})")

        print()
```

Instead of using an f-string here, we could use the `format` method with named placeholders. We could then pass in `**book` to `format`.

```
def show_books(books):
        # Adds an empty line before the output
        print()

        for book in books:
                print("{title}, by {author} ({year})".format
(**book))
```

```
        print()
```

This creates a series of keyword arguments from each book, like this:

```
title="1Q84", author="Haruki Murakami", year="2004"
```

One of the nice things about this approach is that we can define the template elsewhere, which also means we can refer to it in multiple places in our code.

```
book_template = "{title}, by {author} ({year})"

def show_books(books):
        # Adds an empty line before the output
        print()

        for book in books:
                print(book_template.format(**book))

        print()
```

It also becomes more and more effective the more keys you need to access in the dictionary.

Exercises

1) Create a function that accepts any number of numbers as positional arguments and prints the sum of those numbers. Remember that we can use the `sum` function to add the values in an iterable.

2) Create a function that accepts any number of positional and keyword arguments, and that prints them back to the user. Your output should indicate which values were provided as positional arguments, and which were provided as keyword arguments.

3) Print the following dictionary using the `format` method and `**`

unpacking.

```
country = {
        "name": "Germany",
        "population": "83 million",
        "capital": "Berlin",
        "currency": "Euro"
}
```

4) Using `*` unpacking and `range`, print the numbers `1` to `20`, separated by commas. You will have to provide an argument for `print` function's `sep` parameter for this exercise.

5) Modify your code from exercise `4` so that each number prints on a different line. You can only use a single `print` call.

Additional Resources

We have a dedicate post on destructuring where we talk about yet another use for `*`, this time for gathering values when assigning to variables.

© 2021 Teclado Ltd.