LEARN PYTHON PROGRAMMING

Python Slices

Phil Best
6 min read · Apr 7

In this post I want to tell you a little bit about slicing in Python. Slicing is a powerful tool, but it's also quite easy to make mistakes with if you're not careful.

So, what exactly is slicing? Slicing is the process of creating a new sequence from some portion of an existing sequence. It's actually quite an intuitive concept, with very clear parallels to something like cutting a cake in order to get a slice of the whole.

We can perform slicing on any *sequence type* in Python. This includes string, lists, tuples, byte objects and byte arrays.

Because slicing relies on the position of items in a sequence, it cannot be used on things like sets, which do not preserve order. More importantly, they aren't indexed by consecutive non-negative integers, which is why dictionaries also cannot be sliced, despite having a reliable ordering in modern Python.

Creating a slice

Let's define our first slice.

While `slice` may look like a function here, it's actually a class. We therefore just bound `slice_instance` to some `slice` object.

We can see this if we print the type of `slice_instance`:

```
print(type(slice_instance))  # <class 'slice'>
```

So far so good. So what are those numbers we passed into slice when we created this slice object?

If we take a look at the documentation for the slice class, we see that slice has three parameters:

> slice(start, stop[, step])

If you're not familiar with the notation in the documentation for parameters, items inside the square brackets are optional.

Great, so we passed in arguments for the `start` and `stop` parameters. We'll come back to step in a little while. These `start` and `stop` values represent indexes in some as yet unspecified sequence.

So how do we use this slice object? Well, we need some sequence to try it out on.

Let's go with a simple list to start:

```
```

We can get a slice of a sequence using subscript syntax:

```
x_slice = x[slice_instance]
print(x_slice)  # [1, 2]
```

Since our `slice_instance` goes from index 0 to index 2, we got that part of the `x` list only.

There are a couple things to note here:

1. `x_slice` is of type list.
2. The item at index 2 of `x` wasn't included in `x_slice`.

This leads us to our first warning regarding slices: the index we provide for the stop parameter is **not** inclusive.

Slicing other sequence types

Slicing other sequence types uses exactly the same syntax.

We can define a slice object and then use it for any sequence type like so:

```
s = slice(1, 4)

t = (1, 2, 3, 4, 5)  # tuple
l = [1, 2, 3, 4, 5]  # list
c = "12345"          # string

print(t[s])  # (2, 3, 4)
print(l[s])  # [2, 3, 4]
print(c[s])  # 234
```

What you might have noticed from the example above is that slicing a sequence gives us a sequence of the same type back.

Remember that strings are just sequences of characters, and are therefore perfect candidates for slicing!

Defining a slice object inline

Instead of going through this process of defining a slice object, binding it to a variable, and then providing that variable name as part of the subscript syntax, we can do the following:

```
t = (1, 2, 3, 4, 5)
print(t[slice(1, 4)])  # (2, 3, 4)
```

However, there is a faster way still, which we're going to cover next.

A faster way

Python has an alternative syntax for defining a slice directly in the square brackets we use as part of the subscript syntax.

Let's define the same slice object as we used above, but using the new syntax:

```
t = (1, 2, 3, 4, 5)
print(t[1:4])  # (2, 3, 4)
```

As you can probably tell, the first number is our starting index, then

the stop index. Just like before, this stop index is not inclusive.

This new syntax functions the same way for all sequence types.

Leaving some values empty

What might surprise you is that each of the following is valid syntax:

```
print(t[:4])
print(t[1:])
print(t[:])
```

So, what exactly do each of these mean?

When we miss off the starting index, this means "start from the beginning of the sequence".

When we miss off the stopping index, this means "stop at the end of the sequence".

In the latter case, the final element is included in the new slice.

Putting these together, we can guess what the final example means: "give me the whole sequence". When you miss off both starting and ending indices, you get everything back.

Using step values

Right at the start of this post I mentioned that we can also provide an optional step value when creating a slice object. This allows us to skip over values by providing a step greater than `1`.

For example:

```
t = (1, 2, 3, 4, 5)
print(t[1:4:2])  # (2, 4)
```

We go from the item at index `1`, and then go straight to the item at index `3`.

Negative step values

Step values don't need to be positive, and this is actually a really useful property. When a step value is negative, we start at the starting index as usual, but then move along the sequence in reverse.

For example, we might want to start at index `4`, stop at index `2`, and move in steps of `-1`.

```
t = (1, 2, 3, 4, 5)
print(t[4:2:-1]  # (5, 4)
```

Notice that the results came back in the reverse order to the original tuple. This is because the values at the end of the tuple were encountered first, and we kept stepping towards the start of the tuple.

Using extended slicing, we can still grab a whole list using the following syntax: `[::]`. It looks a little arcane, but it just means start at the beginning of the sequence, stop at the end, and use the default step value: `1`.

In combination with a negative step values. we can use syntax like this to check if a sequence is a palindrome, for example:

```
def palindrome_check(word):
    if word == word[::-1]:  # check against full sequence in reverse
        return True
    return False

print(palindrome_check("kayak"))  # True
print(palindrome_check("lemon"))  # False
```

A warning about negative step values

One thing about slices is that it's very easy to end up with an empty slice, particularly when negative values come into play.

For example, we might try to use a negative step with one of our older slices:

```
t = (1, 2, 3, 4, 5)
print(t[1:4:-1])  # ()  ← Empty tuple
```

But the example above will give us back an empty tuple. This is because it's impossible to get from index `1` to index `4` in steps of `-1`. What would should have written is `t[4:1:-1]`, starting at a higher index than where we finish, which would print `(5, 4, 3)`.

Negative indices

In addition to providing a negative step value, we can also provide negative numbers for indices.

When using a negative index, we start counting backwards from the end of the sequence. In our tuple `t` above, the index `-1` is the same as index `4`. In other words, the last item in the tuple.

a slice like this:

```
t = (1, 2, 3, 4, 5)
print(t[-1:-5:-1])  # (5, 4, 3, 2)
```

I chose this example for a particular reason, because it highlights another easy trap to fall in when working with slices. When using negative indices, the stop value is still **not** inclusive. In order to include the item at index `0`, we would have to write:

```
t[-1:-6:-1]
```

Recap

- Slices can be used to create sequences from some portion of another sequence.
- Only *sequence types* can be sliced, as slicing relies on the items being indexed by non-negative indices.
- We can define a slice object creating an instance of the `slice` class, which has three parameters: a starting index, a stopping index, and an optional step value. Remember that the item at the stopping index of a given sequence is **not** included in the slice.
- We can create a slice of a specific sequence by passing a slice object into a pair of square brackets directly after that sequence, e.g. `some_sequence[slice(1, 2)]`. We can also use special slice syntax inside these square brackets, removing the need for use to explicitly create a slice object, e.g. `some_sequence[1:2]`.

have to be careful when using negative values, as it's easy to end up with a slice that contains nothing. One use case for a negative step is quickly reversing a sequence like so: `some_sequence[::-1]`.

I hope you learnt something new, and if you're looking to upgrade your Python skills even further, you might want to try out our Complete Python Course.

Make sure to also check out next week's post where we cover slicing in even more depth.

Phil Best
I'm a freelance developer, mostly working in web development. I also create course content for Teclado!

Read More

Learn Python Programming

Introduction to Object-Oriented Programming in Python →

Dictionary Merge and Update Operators in Python 3.9 →

Working with Python virtual environments: the complete guide →

LEARN PYTHON PROGRAMMING

Python Slices Part 2: The Deep Cut

In this post we take a deeper dive into Python slices, and pull back the curtain on the magic methods behind the slice notation we covered in earlier posts.

Phil Best
4 min read · Apr 14

Share this