LEARN PYTHON PROGRAMMING

Python Slices Part 2: The Deep Cut

Phil Best
4 min read · Apr 14

In this post, we're going to do a slightly deeper dive into slices in Python. If you need a refresher on the basics, be sure to check out our earlier post on the topic!

Without further ado, let's dive right in.

Slice Assignment

In the last post, we talked a lot about grabbing some slice of a sequence, but we can actually do a great deal more with slices.

One interesting thing we can do is replace some slice of a sequence with new values.

```
numbers = [1, 3, 3]
numbers[1:2] = [2]

print(numbers)  # [1, 2, 3]
```

Here we take a slice of `numbers` which includes only the value at index 1. Remember that the stop index for slices is **not inclusive**.

We then use this slice as the left-hand side of an assignment. When we print `numbers`, we can see that it now reads `[1, 2, 3]` showing that the `3` at index 1 has been replaced.

One interesting thing to note is that we assigned another iterable, not just the integer `2`. We could have actually used a tuple or even a set here instead of a list:

```
numbers[1:2] = (2,)  # Don't forget the comma!
numbers[1:2] = {2}
```

However, assigning an integer would have raised a `TypeError`.

```
TypeError: can only assign an iterable
```

Assigning Multiple Values

As we have to assign some iterable type, it stands to reason we can assign multiple values in one go. That's absolutely correct:

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

print(numbers)  # [1, 2, 3]
```

However, what might surprise you is that we can assign an iterable of a different length to our slice.

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

print(numbers) # [1, 2, 3, 4, 5]
```

It actually makes no difference how many items our iterable contains: we can even assign an empty list to some slice without issues. The items at the indexes defined in our slice are simply removed.

```
numbers[1:3] = []
```

Zero Length Slices

We can also exploit the uneven assignment in other ways. For example, we can take an empty slice and use it to insert values in the middle of some sequence.

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

print(numbers)  # [1, 2, 3, 4, 5]
```

Remember that a slice like `[1:1]` is totally valid, but completely empty. It starts at index 1, and ends at index 1, but the stop value is not inclusive, so the value at index 1 is not part of the slice.

A slice like this therefore allows us to insert values at a given index without removing any values in the sequence.

Extended Slicing and Assignment

In the last article, we spoke about extended slicing, using a step

value. We can still perform assignment on slices on this type, but there is a caveat. Unfortunately, if we specify any step value other than the default `1`, we can't use the asymmetrical assignment we've been talking about above. This includes a step of `-1`.

As such, when we're using extended slice syntax, we need to be careful to match the number of values we want to replace.

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

print(numbers)  # [1, 2, 3, 4, 5]
```

The example above works just fine, because we assign `2` to index 1, and `4` to index 3. Everything matches.

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

print(numbers)  # ValueError
```

On the other hand, this second example produces a `ValueError`:

```
ValueError: attempt to assign sequence of size 2 to extended slice
```

Slice Objects and `__getitem__`

When we first started talking about slices, we began by introducing slice objects, and the `slice(1, 2)` syntax. We used this for a little

while before moving on to the shorthand we've been using in the examples above.

We've actually seen a number of ways to take a slice of some sequence:

```
numbers = [1, 2, 3]
new_slice = slice(1, 3)

a = numbers[new_slice]
b = numbers[slice(1, 3)]
c = numbers[1:3]
```

These are all absolutely identical in terms of functionality. If we printed any of `a`, `b`, or `c`, we'd get `[2, 3]` printed to the console.

However, this square brackets syntax is only a convenience. What's actually happening is this:

```
d = numbers.__getitem__(slice(1, 3))

print(d)  # [2, 3]
```

The slightly arcane looking `__getitem__` is one of Python's many special methods, often called "dunder" (double underscore) methods.

What's interesting about this, is that we can use these special methods in our own classes, which means we can also provide slicing support for our own custom types.

`__getitem__` actually accepts more than just slices. We could also pass in a single index to `__getitem__`, which is exactly what happens when we do something like this: `numbers[0]`.

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

print(numbers[2])  # 3
print(numbers.__getitem__(2))  # 3
```

`__getitem__` also has a counterpart called `__setitem__`, which is called when we do slice assignment, or when we try to replace a value at a given index using subscript notation with single indexes:

```
numbers = [1, 2, 3, "hello", 5]
numbers[3] = 4  # numbers.__setitem__(3, 4)

print(numbers)  # [1, 2, 3, 4, 5]
```

Special methods are a large and relatively advanced topic, so don't worry if you're a little confused at this point. Special methods will be getting suitably special treatment in a future blog post, so stick around if you're interested in learning more about them!

Recap

- In addition to using slices to grab a range of values from a sequence, we can also use slices to replace values in a sequence as part of an assignment operation.
- We have to use an iterable type when assigning using a slice. This can be anything from list and tuples to sets.
- Slice assignment can be asymmetrical: the length of the slice can be different to that of the iterable object we want to assign.
- We can define a slice of zero length by specifying a start and stop value at the same index. We can use an empty slice to

insert an arbitrary number of values at a given index.

- When using extended slicing, any step value except `1` imposes some caveats on slice assignment. When using a different step size, the number of items we assign must match the number of items in the slice.
- The subscript notation we use for slicing is actually shorthand for the `__getitem__` and `__setitem__` special methods. This is very useful, because it means we can define slicing behaviour for our own custom types.

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

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

Read More

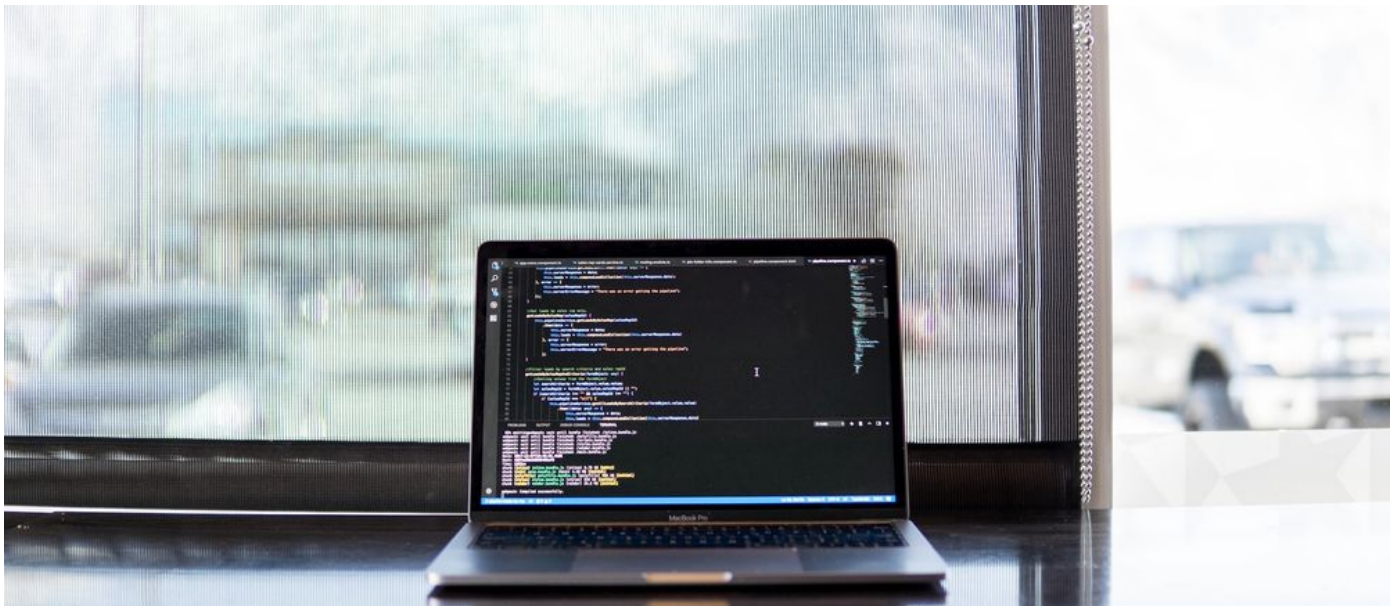
Learn Python Programming

Introduction to Object-Oriented Programming in Python →

Dictionary Merge and Update Operators in Python 3.9 →

Working with Python virtual environments: the complete guide →

→ See all 144 posts



LEARN PYTHON PROGRAMMING

Creating a New Sequence Type in Python - Part 1

Learn to exploit the power of Python's magic methods by creating a new sequence type from scratch!

Phil Best
9 min read · Apr 19

Like what you see? Enter your e-mail to hear when new posts come out!

E-mail*

Name*

Receive our newsletter and get notified about new posts we publish on the site, as well as occasional

special offers.

Sign Up