LEARN PYTHON PROGRAMMING

Creating a New Sequence Type in Python – Part 1

Phil Best

9 min read · Apr 19

In this post, we're going to be looking at Python's magic or "dunder" methods, and we're going to use these magic methods to start creating a brand new sequence type from scratch.

The new sequence type we're going to make is called a `LockableList`. It's going to have all the core features of a regular list, but we're going to have the option of making it immutable by calling a `lock` method on a `LockableList` object. If we want to make it mutable again, we just need to call `unlock` on our `LockableList`, after which we'll be able to modify values once again.

Here is an example of how it will work:

```
friends = LockableList("Rolf", "Bob", "Jen")
friends.append("Adam")
print(friends)  # ['Rolf', 'Bob', 'Jen', 'Adam']

friends.lock()
friends.append("Anne")  # Error
```

This post and the subsequent parts are going to cover some relatively advanced material, but we have a number of resources available to help you along the way. We'll be working with classes, so if you need a brief refresher on those, you can look at this post. We're also going to implementing slicing for our `LockableList`, so if you're not very familiar with how slicing works, we have an introductory and more advanced post on this topic. If you want something a little more comprehensive, you might be interested in our Complete Python Course.

With all that out of the way, let's get started!

Defining Our `LockableList` Class

First things first, if we need to define a class for our `LockableList`. Inside the class we're going to create properties to store the list's values and keep track of the the object's locked state. We're also going to be defining a large number of methods which will allow us to recreate the behaviour of Python's standard list type, just with our own special twist.

```
class LockableList:
    def __init__(self):
        pass
```

We've already run into our first special method: `__init__`. Unlike the other methods we'll be defining, `__init__` gets called automatically when an object instance gets created and it allows us to set some initial values. Right now, our `__init__` method isn't doing very much, so let's fix that.

```
class LockableList:
```

```
    def __init__(self, values):
        self.values = values
```

We've now defined a second parameter for `__init__` called `values`, meaning we can now pass in some argument when creating an instance of our `LockableList`. The value we pass in is then stored in `self.values`. We can check this works like so:

```
l = LockableList([1, 2, 3, 4, 5])
print(l.values)  # [1, 2, 3, 4, 5]
```

In addition to storing a set of values, we also need to keep track of our lock state. By default, I think it makes sense to create our `LockableList` unlocked, with the option for users to specify a locked state if they so wish.

```
class LockableList:
    def __init__(self, values, locked=False):
        self.values = values
        self._locked = locked
```

Here I've used a default parameter value of `False` for our lock state for the reasons stated above. Note that I've also prefaced the the `locked` property with an underscore. This is just a convention used to indicate that this variable is intended for internal use only.

Overall, this is a good start, but I think we can do a little better.

Improving How We Create a `LockableList`

At the moment we have a bit of a niggling issue. What happens if a user creates a `LockableList` like this:

```
l = LockableList(2, 6)
```

We've been assuming that our users are just going to pass in a list or some other collection to populate our `LockableList`, but there's nothing stopping them doing something like we did in the example above.

In this case, our `_locked` state is going to take a value of `6`, but we're expecting a Boolean value. Later on we're going to be checking the truthiness of `_locked`, and a value like `6` will evaluate to `True`, even though the initial `_locked` value is meant to be `False`.

Even worse, the values variable only contains `2`, so we've accidentally thrown away some of our user's data.

Now imagine the user enters a third value when initialising their `LockableList`. Now they're going to get a `TypeError`, because they've provided too many arguments. Not great.

One solution is to get rid of the `values` parameter and replace it with `*values`. The important part here is the `*`, which actually does two things for us:

1. It allows us to take in an arbitrary number of positional arguments (the main use of the `*` syntax).
2. It makes our `locked` parameter keyword only, as it occurs after the `*values`.

Great, so we've prevented users from accidentally overwriting the default `_locked` state, and we've also made the setup of our `LockableList` a little easier too. Users can now write `LockableList(1, 4, 6, 2)` instead of needing to pass in a collection. So how do we access the values in `*values`?

By default, the values collected by `*values` are going to be placed in a tuple, and can be accessed by referring to the `values` parameter. This isn't really what we want. For one, tuples are immutable objects, and using a mutable type for our internal storage is going to make our lives much easier.

Instead of using the standard `values` tuple, I'm going to convert it to a list:

```
class LockableList:
    def __init__(self, *values, locked=False):
        self.values = list(values)
        self._locked = locked
```

With that, we're now storing our user's values internally as a standard Python list.

Printing the Content of Our `LockableList`

Earlier we were successful in printing the contents of our `LockableList` instance, `l`, but it certainly felt pretty clunky. When we print a normal list, we can just use a variable name, we don't have to refer to some internal structure of the `list` class. I want our `LockableList` to work the same way.

As it turns out, we can use another special method to get the job done. When we call `print`, Python goes looking inside the object we

passed in as an argument for a method called `__str__`. If it can't find one, it looks for one called `__repr__` instead, which we'll get to in a little bit.

`__str__` allows us to define a user-friendly representation of a given object. In our case, I think it's fairly safe to use the standard list syntax to represent our `LockableList`, but we could really return anything we want. The only rule we need to keep in mind is that it has to return a string.

```
class LockableList:
    def __init__(self, *values, locked=False):
        self.values = list(values)
        self._locked = locked

    def __str__(self):
        return f"{self.values}"
```

As we already use a list internally for storing the values in a `LockableList` we can grab that list and put it in an f-string using `{self.values}`. Now we can do this:

```
friends = LockableList("John", "Rolf", "Mary")
print(friends)  # ['John', 'Rolf', 'Mary']
```

Much better!

Returning the Length of a `LockableList`

Python has a handy built-in function called `len` which lets us get some measurement of the size of an object. For collections, this is generally the amount of objects they contain.

What happens if we try to get the `len` of a `LockableList`?

```
friends = LockableList("John", "Rolf", "Mary")
print(len(friends))  # TypeError: object of type 'LockableList' has no len(
```

That just won't do.

Luckily, Python once again has a special method we can implement to give us this functionality: `__len__`.

At this point, you should be noticing something of a pattern. There's some functionality we want our sequence type to have, and to provide this functionality we turn to a special method. The names of these methods are also generally very appropriate and strongly tied to the built in method or operation we'd like to use.

Implementing `__len__` is a relatively simple task. Internally we use a standard list for storing our user's values, and a list already implements `len`, so there's no need for us to count the values ourselves.

```
class LockableList:
    def __init__(self, *values, locked=False):
        self.values = list(values)
        self._locked = locked

    def __str__(self):
        return f"{self.values}"

    def __len__(self):
        return len(self.values)
```

```
friends = LockableList("John", "Rolf", "Mary")
print(len(friends))  # 3
```

Getting Items By Index

So far we can print all the items in a `LockableList` and we can find out how many items are in the collection, but we can't yet access specific items by their index.

If you read our blog posts on slices, you'll know that when we write `friends[2]`, Python uses a method called `__getitem__` to retrieve the item at that index. It uses the same method for retrieving a slice of some sequence.

As we can see, this is another dunder method, and we can implement it in our `LockableList`.

Before we write any code, there are a few things we need to think about when it comes to our implementation of `__getitem__`.

1. Firstly, we need to distinguish between a slice and a reference to a single index, because we need to handle these two uses of `__getitem__` differently.
2. Secondly, we need to consider how to handle negative indexes.

Technically we can just pass the negative indexes straight to the internal list, but we're going to imagine we don't have that luxury. The same goes for handling slices.

Let's first consider how to handle negative indexes. If a user asks for the item at index `-1`, they want the final item in the sequence. In a sequence of length `5`, the index `-1` and the index `4` are the same thing. There is a very clear connection between these numbers which

will allow us to handle negative indexes: the length of the sequence plus the negative index equals the positive index version of that negative index.

Getting Individual Items

With this, we can implement the first part of `__getitem__` for handling the retrieval of items at a specific index:

```
class LockableList:
    def __init__(self, *values, locked=False):
        self.values = list(values)
        self._locked = locked

    def __str__(self):
        return f"{self.values}"

    def __len__(self):
        return len(self.values)

    def __getitem__(self, i):
        if isinstance(i, int):
            # Perform conversion to positive index if necessary
            if i < 0:
                i = len(self.values) + i

            # Check index lies within the valid range and return value if p
            if i < 0 or i >= len(self.values):
                raise IndexError("LockableList index out of range")
            else:
                return self.values[i]
```

First we use `isinstance` to check the type of the value passed in for `i`. Specifically, we check to see if it's an integer. If it is, we know we can use it to access a specific item in our sequence.

For now, we're just going to ignore any other input types, but we'll

handle them later on.
Once we know we're working with an integer, we check to see if the index we were given is negative. If it is, we know we need to convert it to a positive index using the method we outlined above.

If the index we were provided was greater than or equal to the length of our sequence, we know it must be outside the range of values we have, because our indexes start at `0`. Our highest index is therefore the length of our sequence - 1. Similarly, if they provide a large negative index it might exceed the length of our sequence.

If a user tries to access a value at an index which doesn't exist, we will raise an `IndexError`.

Finally, if we didn't run into any problems, we'll return the item in our internal list corresponding to the index the user requested. Yay!

This is actually incredibly significant, because we can now do something else with our `LockableList` objects: we can loop over them!

```
friends = LockableList("John", "Rolf", "Mary")

for friend in friends:
    print(friend)
```

Getting Back a Slice of Our `LockableList`

Now that we can get back individual items from our `LockableList`, we need to handle slices.

Remember that when grab a slice of some sequence, such as by doing `friends[1:3]`, Python passes in the `1:3` slice object to

`__getitem__` .

The first thing we should do is check that `index` is of type `slice` . If it isn't (and it's not an `int`), then we need to raise a `TypeError` .

Once we know we're working with a slice, we can make use of the `indices` method, which will return concrete range values for a specific sequence we pass in for the slice object we call it on.

```
friends = ["John", "Rolf", "Mary", "Adam", "Jose"]
my_slice = slice(-3, -1, 1)  # akin to [-3:-1]
my_slice.indices(len(friends))  # (2, 4, 1)
```

In the example above we have a slice object, `slice(-3, -1, 1)` . We then call `indices` on this slice object and pass `friends` as an argument. Because `friends` has a length of `5` , we get a tuple back like this: `(2, 4, 1)` . It contains a start index, a non-inclusive stop index, and a step value.

We can put these values straight into `range` to create a sequence of indexes we should return values for. We can then loop over the required indexes and return the values as a new `LockableList` by unpacking a list comprehension.

```
class LockableList:
    def __init__(self, *values, locked=False):
        self.values = list(values)
        self._locked = locked

    def __str__(self):
        return f"{self.values}"

    def __len__(self):
        return len(self.values)
```

```
def __getitem__(self, i):
    if isinstance(i, int):
        # Perform conversion to positive index if necessary
        if i < 0:
            i = len(self.values) + i

        # Check index lies within the valid range and return value if p
        if i < 0 or i >= len(self.values):
            raise IndexError("LockableList index out of range")
        else:
            return self.values[i]
    elif isinstance(i, slice):
        start, stop, step = i.indices(len(self.values))
        rng = range(start, stop, step)
        return LockableList(*[self.values[index] for index in rng])
    else:
        invalid_type = type(i)
        raise TypeError(
            "LockableList indices must be integers or slices, not {}"
            .format(invalid_type.__name__)
        )
```

With that we can now accept slice objects as arguments to `__getitem__` !

Wrapping up

If you enjoyed this post, make sure to check back next week when I'll be implementing the counterpart to `__getitem__` used for modifying a sequence: `__setitem__` . Until then, play around with the code and see what else you can implement on your own. You can find details of many of Python special methods here.

Phil Best

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

Learn Python Programming

Using "match...case" in Python 3.10 →

Introduction to Object-Oriented Programming in Python →

Dictionary Merge and Update Operators in Python 3.9 →

→ See all 145 posts

PYTHON SNIPPETS

More Uses for Else in Python

Get familiar with the different uses for else in Python. It's not just for if statements: we can use it with loops, and for error handling too!

Phil Best

2 min read · Apr 22

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