LEARN PYTHON PROGRAMMING

Creating a New Sequence Type in Python - Part 2

Phil Best
10 min read · Apr 25

Welcome back to our series on creating a new sequence type in Python using classes and Python's magic methods. If you didn't catch part one, you can find it here.

In this post, we're going to focus on adding items to our new `LockableList` sequence, and we'll also be creating the `lock` and `unlock` methods to change the mutability of our `LockableList` objects. We're also going to be implementing the long overdue `__repr__`.

A Quick Recap

First, let's do a quick recap of what we've done so far. At the moment, our `LockableList` class looks like this:

```
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__)
                )
```

We have an `__init__` method which gets called when an instance of our `LockableList` class gets created. This allows us to take in values during the creation of a `LockableList` such as some initial data, and also a lock state.

We also have three other magic methods defined: `__str__`, `__len__`, and `__getitem__`.

`__str__` allows us to return something whenever Python asks for a string representation of our object. An example might be when we pass our object to the built in `print` function.

`__len__` allows us to return the length of our object, indicating how many items the sequence contains. This means we can call `len` on our function without it raising an error.

`__getitem__` is the most complicated method we've implemented so far. `__getitem__` allows us to access specific items or ranges of items from the list. If we create a lockable list like this:

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

We can access items like we might access a list:

```
print(friends[0])    # Rolf
print(friends[1:3])  # ['John', 'Anna']
```

We can also loop over our `LockableList` object using a for or while loop, or by using a comprehension.

Adding the `lock` and `unlock` Methods

Let's start by adding the methods which make our type unique: `lock` and `unlock`.

The intention here is for us to modify the lock state of our `LockableList` object. When a list is locked, attempting to modify the sequence will raise an error, much like when we try to modify a tuple.

The `lock` and `unlock` methods themselves are quite simple. All of the logic for checking the lock state will be done elsewhere. `lock` and `unlock` are concerned with one thing: changing the value of `self._locked` from `True` to `False` and *vice versa*.

```
def lock(self):
    self._locked = True

def unlock(self):
    self._locked = False
```

We can test this works like so:

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

friends.lock()
print(friends._locked)  # True

friends.unlock()
print(friends._locked)  # False
```

Adding `__repr__`

`__repr__` is an important dunder method that we should be implementing for every class we make. Much like `__str__`, `__repr__` returns a string representation of a particular object, but it serves a different function.

An easy way to think about `__str__` vs `__repr__` is that `__str__` is for showing information to the user, while `__repr__` is information for the developer. When we implement `__repr__` we want to provide all the information needed to reproduce the object, being as specific as we can. This might involve including the module name, for example.

In our case, we're going to show how to define a `LockableList` object, and we're going to show the arguments required to recreate the specific `LockableList` instance `__repr__` was called on.

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

# LockableList(['Rolf', 'John', 'Anna'])
```

However, the representation above doesn't accurately describe how to create the `LockableList` object we're concerned with. Instead, we pass in each string individually, without the square brackets, and any non-string entry would be added just the same. We can easily store a list, integers, or a set as members of our `LockableList`, each passed in as a separate argument.

As such, we're going to call `__repr__` on each item in `self.values`, getting back an appropriate representation for each object in values, based on their type. We'll then `join` each value with `", "` as the joining string.

```
def __repr__(self):
    values = ", ".join([value.__repr__() for value in self.values])
    return f"LockableList({values})"

# LockableList('Rolf', 'John', 'Anna')
```

With that, our `LockableList` class looks something like this:

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

    def __str__(self):
```

```python
        return f"{self.values}"

    def __repr__(self):
        values = ", ".join([value.__repr__() for value in self.values])
        return f"LockableList({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__)
            )

    def lock(self):
        self._locked = True

    def unlock(self):
        self._locked = False


test = LockableList("Rolf", "John", "Anna", 3, ["Hello", "World"])
print(repr(test))  # LockableList('Rolf', 'John', 'Anna', 3, ['Hello', 'Wor
```

Implementing `__setitem__`

Now it's time to start implementing `__setitem__`, which will let us assign new objects to specific indices of our `LockableList`. In other words, we'll be able to do something like this:

```
friends = LockableList("Rolf", "John", "Anna")
friends[1] = "Jose"
```

At the moment, Python will raise an exception if we try to do this:

```
TypeError: 'LockableList' object does not support item assignment
```

Implementing `__setitem__` will also allow us to carry out slice assignment. If you're not familiar with this topic, it's a good idea to read through our advanced slicing post to get up to speed.

Once again, we're going to need a different process for dealing with item assignment and slice assignment, and we're also going to have to handle invalid indices, whether they're simply out of range, or whether they're the result of a `TypeError`.

In the case of `__setitem__`, we also have to pay attention to the lock state of our `LockableList` and we have to make sure to raise an exception when a user tries to modify a locked sequence.

Checking the lock state

First things first, let's define our new `__setitem__` method and add some logic to check whether or not the `LockableList` is indeed locked. If it is, we can raise a `RuntimeError`.

For `__setitem__` we're going to specify three parameters: `self`, since we need access to `self._locked`; `i` which represent some index or range of indices; and `values`, which represents the items we want to add to our `LockableList` object.

```
def __setitem__(self, i, value):
    if self._locked == True:
        raise RuntimeError(
            "LockableList object does not support item assignment while loc
        )
```

Checking the lock state is as simple as using an if statement to check the current value of `self._locked`. If `self._locked` is `True`, we immediately raise our `RuntimeError`, otherwise we're going to try to add the specified items to the sequence.

Adding single items

Once we've established that our `LockableList` is unlocked and capable of mutation, we can start assigning items to the sequence.

Our implementation of `__setitem__` is going to look very similar to our implementation of `__getitem__`. First we check if the index is negative, and if it is, we attempt to convert it to a positive index. We then check if the index lies within the valid range for this particular `LockableList` instance, raising an `IndexError` if not.

Where our implementation will differ is in how we handle the case where no errors occurred. Instead of returning the item at a given index, we're going to assign to it. As we use a list internally to store our values, this should be relatively easy:

```
def __setitem__(self, i, values):
    if self._locked == True:
```

```
        raise RuntimeError(
            "LockableList object does not support item assignment while loc
        )

    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 assign value if possi
        if i < 0 or i >= len(self.values):
            raise IndexError("LockableList index out of range")
        else:
            self.values[i] = values
```

Implementing slice assignment

There are a few things we need to keep in mind when implementing `__setitem__` for slices:

1. Slices allow for asymmetrical assignment. We can assign more values than there is space to accommodate them.
2. We can also assign fewer items than we replace, and the proceeding values in the sequence will move to fill the remaining space.
3. When using extended slices, these special properties do not apply unless the step value is 1.

Strictly speaking, these limitations are not imposed by slices themselves, but by how list implements slice assignment. Since we're trying to imitate the functionality of the built in list type, we will also be imposing these limitations.

We could take the easy way out and let the underlying list do all the work for us, but where's the fun in that? Let's do as much as we can from scratch.

Extended slicing

Let's start by creating an `elif` branch to capture slice objects. We can use `indices` to get a `start`, `stop`, and `step` value corresponding to this particular sequence, just like we did for `__getitem__` in part 1.

`rng` will contain the specific indices in our sequence that correspond to the slice object provided, again, just like before.

We can now perform a check to see if the step value is anything other than 1. If it is, we know that the `len` of `rng` must be equal to the `len` of the `values` passed into `__setitem__`. If this condition isn't satisfied, we'll raise a `ValueError`.

```
elif isinstance(i, slice):
    start, stop, step = i.indices(len(self.values))
    rng = range(start, stop, step)
    if step != 1:
        if len(rng) != len(values):
            raise ValueError(
                "attempt to assign a sequence of size {} to extended slice
                .format(len(values), len(rng))
            )
```

If we don't encounter any exceptions, we need to loop over the indices in `rng` and assign each item in `values` one at a time.

In this case, I'm going to use `zip` to create a series of tuples, with the first value being an index in `rng` and the second value being an item in `values`.

```
    elif isinstance(i, slice):
        start, stop, step = i.indices(len(self.values))
        rng = range(start, stop, step)
        if step != 1:
            if len(rng) != len(values):
                raise ValueError(
                    "attempt to assign a sequence of size {} to extended slice
                    .format(len(values), len(rng))
                )
            else:
                for index, value in zip(rng, values):
                    self.values[index] = value
```

Now we can assign to an extended slice with a step value other than 1, like so:

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

friends[0:3:2] = ["Jose", "Mary"]
print(friends)  # ["Jose", "John", "Mary"]

friends[2::-2] = ["Rolf", "Anna"]
print(friends)  # ["Anna", "John", "Rolf"]
```

Standard slice assignment

Finally we can handle the standard assignment case for slices. This is a little bit complicated, as we'll potentially have to chop up the underlying list to fit in values.

Fortunately, we can use some properties of slices to help us greatly here. We already have the `start` and `stop` values for the provided slice object, `i`. We also know that the `start` index of a slice is inclusive, while `stop` is not.

We can therefore do something like this:

```
self.values = self.values[:start] + values + self.values[stop:]
```

Here we take a slice of all the values from the start of the internal list up to the start of our slice, not inclusive. In other words, everything up to the first slice item. We append to this range of values the values we want to add to the sequence.

We then append another set of values, this time from the `stop` index of our slice up to the end of the sequence.

All of this is then assigned to `self.values`, replacing the internal list with this new sequence.

One last thing we need to do is handle the case where the user passes in an invalid index, just like we did for `__getitem__`.

With all our new code, we end up with something like this:

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

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

    def __repr__(self):
        values = ", ".join([value.__repr__() for value in self.values])
        return f"LockableList({values})"

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

    def __getitem__(self, i):
```

```python
        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__)
            )

    def __setitem__(self, i, values):
        if self._locked == True:
            raise RuntimeError(
                "LockableList object does not support item assignment while
            )

        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 assign value if p
            if i < 0 or i >= len(self.values):
                raise IndexError("LockableList index out of range")
            else:
                self.values[i] = values
        elif isinstance(i, slice):
            start, stop, step = i.indices(len(self.values))
            rng = range(start, stop, step)
            if step != 1:
                if len(rng) != len(values):
                    raise ValueError(
                        "attempt to assign a sequence of size {} to extende
                        .format(len(values), len(rng))
```

```
                    )
                else:
                    for index, value in zip(rng, values):
                        self.values[index] = value
            else:
                self.values = self.values[:start] + values + self.values[st
        else:
            invalid_type = type(i)
            raise TypeError(
                "LockableList indices must be integers or slices, not {}"
                .format(invalid_type.__name__)
            )

    def lock(self):
        self._locked = True

    def unlock(self):
        self._locked = False


friends = LockableList("Rolf", "John", "Anna")
friends[1:2] = ["Jose", "Mary"]

print(friends)  # ['Rolf', 'Jose', 'Mary', 'Anna']
```

I hope you're enjoying this series. We'll be back soon to add *even more* functionality to our new sequence type. Until then, have a look into the `__add__`, `__radd__`, and `append` methods and see if you can implement them yourself from scratch. You can find information about the `__add__` and `__radd__` methods in the official documentation.

Wrapping up

If you're looking for something a little more comprehensive, you might want to try out our Complete Python Course. It's over 35 hours long, with exercises, projects, and quizzes to help you get a really good understanding of Python. We think it's awesome, and we're sure you will too!

You might want to also think about signing up to our mailing list below!

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

Read More

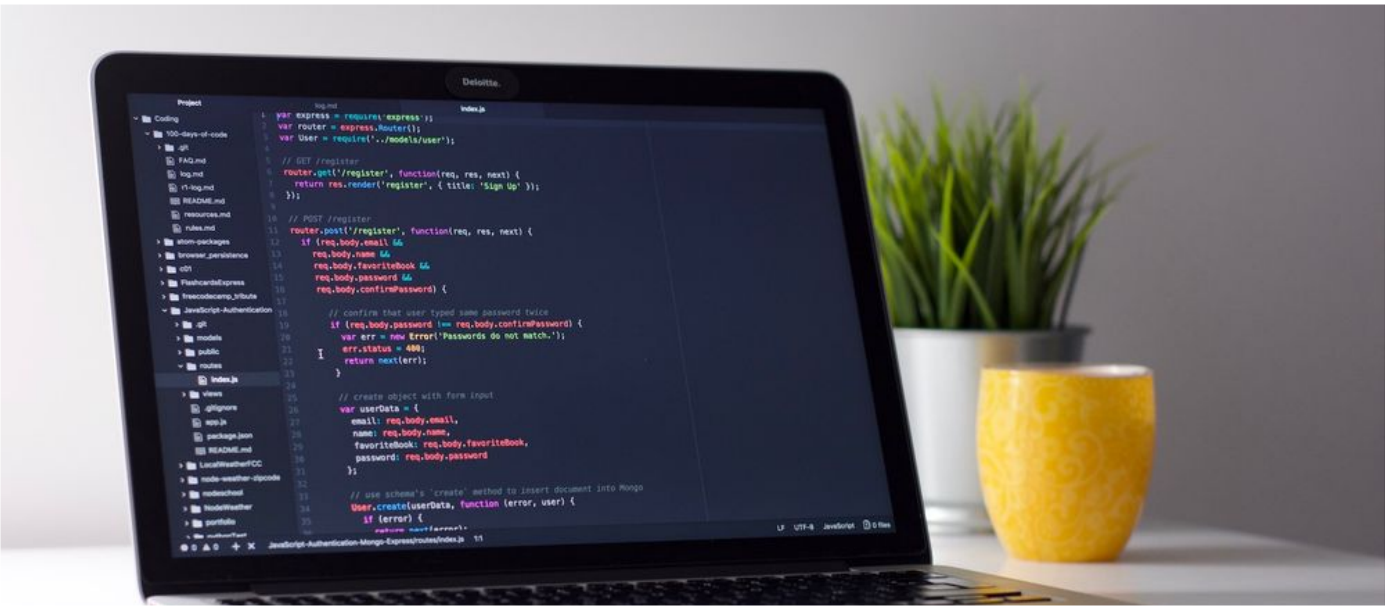
Learn Python Programming

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

Quick Sequence Reversal in Python

Python lists have a handy method called reverse, but it's not always what we want. For a start, we can't use it on other sequence types, like tuples and strings, and it also

Phil Best
1 min read · Apr 29