LEARN PYTHON PROGRAMMING

Creating a New Sequence Type in Python - Part 3

Phil Best
9 min read · May 16

Welcome to part 3 of our series on Python's special methods. In this post we'll be covering a feature called operator overloading, and how we can achieve this using Python's special methods. To do so we'll be continuing our journey to create our new sequence type (the `LockableList`), so if you haven't been keeping up with this series, be sure to check out part 1 and part 2.

A Quick Recap

While I recommend taking a look at our previous posts for a full breakdown of the code, we've so far created a class to describe our new `LockableList` sequence type. In the class we've defined a number of special or "dunder" methods, starting with `__init__`.

`__init__` is called automatically when an instance of our class is created, and it's here that we take in some initial values, as well as some configuration regarding the lock state of our list.

We next have the `__str__` and `__repr__` methods, which return string representations of a given `LockableList` object, with `__str__` providing user friendly output, while `__repr__` is more aimed at developers using our code.

We also defined `__getitem__` and `__setitem__`, which are responsible for retrieving, updating, and adding values in our sequence using either indexes or slice objects.

Finally we have defined two normal methods called `lock` and `unlock` which allow us to update the lock state of a given `LockableList` object. While locked, a `LockableList` cannot be mutated, and acts very much like a tuple.

Here is the complete class definition so far:

```
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):
        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:
            raise RuntimeError(
                "LockedList object does not support item assignment while l
            )

        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 {}"
```

```python
                .format(invalid_type.__name__)
            )

    def lock(self):
        self._locked = True

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

What is Operator Overloading?

Operator overloading is a feature of many languages which allows us to define new behaviour for existing operators with regards to certain operand types. In Python, we accomplish this behaviour using special methods, and we've already touched on this behaviour previously. Note that we can access `LockableList` items by index using subscript notation.

```python
l = LockableList(1, 2, 3)
print(l[2])  # 3
```

This is an example of operator overloading, as we we have redefined the behaviour of these square brackets when used in conjunction with a `LockableList` object.

We can define behaviour for just about any operator we can think of, with each corresponding to a generally sensibly named special method. The `*` operator corresponds to `__mul__`, for example, while `+` corresponds to `__add__`.

This isn't the end of the story, however. If you were to peruse the documentation, you'd also find `__radd__` and `__iadd__`, so what's going on here?

When Python encounters a binary operator like `+`, Python first checks the left-hand operand for information on how to perform the operation for the relevant types. If no such information exists, Python tries the right hand operand instead, but this time looks for an `r` version of the relevant special method. We therefore have special methods like `__radd__`, `__rmul__`, and `__rsub__`.

`i` versions of the special methods represent an in-place operation. This is an operation that doesn't require an assignment.

```
a = 5
a = a + 1  # __add__

b = 5
b += 1  # __iadd__
```

These `i` versions of the special methods are very important for us, because while we want to implement in place modifications to our `LockableList` objects, we want to prevent such operations while a `LockableList` is locked.

`__add__`, `__radd__`, and `__iadd__`

Let's start off with `__add__`. First we have to think about what we want to happen when we try to concatenate `LockableList` objects, and indeed objects of other types. One important question is, which types will we exclude?

In my implementation, I'm going to follow the example of the list type, except I will accept lists and `LockableList` objects. The resulting type will always be a new `LockableList` object, and the new object will use the default lock state.

Ultimately these kinds of decisions are entirely up to you, so you can write your own classes to function differently to mine if you don't like my choices in this regard. You might want to only allow concatenation with other `LockableList` objects, or you might want to set the new object to locked if either of the operands were locked. You really can do whatever you want.

Implementing `__add__`

```
def __add__(self, other):
    if isinstance(other, (list, LockableList)):
        return LockableList(*(self.values + other))

    invalid_type = type(other)
    raise TypeError(
        'can only concatenate list or LockableList (not "{}") to LockableLi
        .format(invalid_type.__name__)
    )
```

Here we define `__add__` with two parameters: `self` and `other`. These names are entirely dictated by convention, and you can actually name them whatever you like.

In our case, `self` as usual refers to the current `LockableList` object, while `other` refers to the right hand operand of the `+` operator. We have no idea what wacky things our users are going to try to concatenate to our `LockableList` objects, so we need to carry out some checks.

A simple if statement is enough here in conjunction with the `isinstance` built in function. If you don't know about `isinstance`, you can read more about it in the official documentation.

Essentially, `isinstance` will allow us to verify that a given object is of a certain type. We can specify a number of types using a tuple, and as you can see above, we specify `list` and `LockableList`. If `other` is an instance of `list` or `LockableList`, `isinstance` will return `True`.

In the event that we find a valid type, we can simply concatenate the values of `other` to the internal list we use for storage, and unpack the new list into a new `LockableList` object. There is a problem with this which we'll look at in a moment.

In the event that the user tries to concatenate an invalid type, we raise a `TypeError`. This is very similar to the errors we've used previously in this project.

With that, we can do some cool stuff:

```
numbers_one = LockableList(1, 2, 3)
numbers_two = [4, 5, 6]

print(numbers_one + numbers_two)  # [1, 2, 3, 4, 5, 6]
```

But as I mentioned, there is a problem:

```
numbers_one = LockableList(1, 2, 3)
numbers_two = LockableList(4, 5, 6)

print(numbers_one + numbers_two)  # TypeError
```

Remember that internally we use a list for storing values in our `LockableList`. Our return value for `__add__` is `self.values + other`.

`self.values` is a list; `other` is a `LockableList`, so when we use `+`, we end up calling the list's `__add__` method, and lists only allow concatenation with other lists. We therefore get a `TypeError`.

The solution is to implement `__radd__`, so that Python can fall back on the methods defined `LockableList` when the list's method raises an error.

Implementing `__radd__`

Our implementation of `__radd__` is going to be near identical to `__add__`, but with the order of `other` and `self.values` reversed. This is to preserve the order of the values based on the order of the operands.

```
def __radd__(self, other):
    if isinstance(other, (list, LockableList)):
        return LockableList(*(other + self.values))

    invalid_type = type(other)
    raise TypeError(
        'can only concatenate list or LockableList (not "{}") to LockableLi
        .format(invalid_type.__name__)
    )
```

Now we can avoid the error we were getting before:

```
numbers_one = LockableList(1, 2, 3)
numbers_two = LockableList(4, 5, 6)

print(numbers_one + numbers_two)  # [1, 2, 3, 4, 5, 6]
```

Implementing `__iadd__`

Finally, let's take care of `__iadd__`.

`__iadd__` is a little different, since we have to take note of our lock state, but the rest of our implementation will be fairly similar.

```
def __iadd__(self, other):
    if self._locked:
        raise RuntimeError(
            "LockedList object does not support in-place concatenation whil
        )

    if isinstance(other, (list, LockableList)):
        self.values = self.values + other
        return self

    invalid_type = type(other)
    raise TypeError(
        'can only concatenate list or LockableList (not "{}") to LockableLi
        .format(invalid_type.__name__)
    )
```

In the case of `__iadd__`, we update the internal list directly, and then return a reference to the current object. This allows us to preserve the same object, while updating the values contained within the given `LockableList`.

```
numbers_one = LockableList(1, 2, 3)
numbers_two = LockableList(4, 5, 6)
numbers_one += numbers_two

print(numbers_one)  # [1, 2, 3, 4, 5, 6]
```

A Note about `__iadd__`

An interesting thing to note is that our class was already capable of handling the `+=` augmented arithmetic operator before we implemented `__iadd__`. Try it out for yourself.

If this is the case, why did we bother implementing `__iadd__` at all? `__iadd__` is really an optimisation feature. In the absence of `__iadd__` Python will fall back to the `__add__` method, but our `__add__` method creates a new `LockableList` object, and creating an object isn't free. By implementing `__iadd__` we can bypass the creation of this new object, saving us a little bit of computing time.

The Completed Class

With that, our class definition is getting very large, but we've accomplished an awful lot:

```
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):
        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:
            raise RuntimeError(
                "LockedList object does not support item assignment while l
            )

        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(
```

```python
                "LockableList indices must be integers or slices, not {}"
                .format(invalid_type.__name__)
            )

    def __add__(self, other):
        if isinstance(other, (list, LockableList)):
            return LockableList(*(self.values + other))

        invalid_type = type(other)
        raise TypeError(
            'can only concatenate list or LockableList (not "{}") to Lockab
            .format(invalid_type.__name__)
        )

    def __radd__(self, other):
        if isinstance(other, (list, LockableList)):
            return LockableList(*(other + self.values))

        invalid_type = type(other)
        raise TypeError(
            'can only concatenate list or LockableList (not "{}") to Lockab
            .format(invalid_type.__name__)
        )

    def __iadd__(self, other):
        if self._locked:
            raise RuntimeError(
                "LockedList object does not support in-place concatenation
            )

        if isinstance(other, (list, LockableList)):
            self.values = self.values + other
            return self

        invalid_type = type(other)
        raise TypeError(
            'can only concatenate list or LockableList (not "{}") to Lockab
            .format(invalid_type.__name__)
        )

    def lock(self):
        self._locked = True

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

Where to Go From Here

If you've found this project interesting, I'd recommend trying to implement `__mul__` for yourself, along with its variant methods. Of course you can go much further than this if you like: it's entirely up to you! You can find plenty of information about Python's special methods in the official documentation.

I hope you've learnt something about how Python's special methods work over the course of this short series, and I hope you find new and interesting ways to use them in your future projects.

If you're interested a more comprehensive look at object oriented programming in Python, you might want to check out our Complete Python Course! There's over 35 hours of material, so there's plenty to sink your teeth into.

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

List Comprehensions in Python

Learn how to use Python's list comprehension syntax: an awesome, Pythonic tool for creating new lists from existing iterables.

Phil Best

1 min read · May 20

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

The Teclado Blog © 2022

Privacy Policy