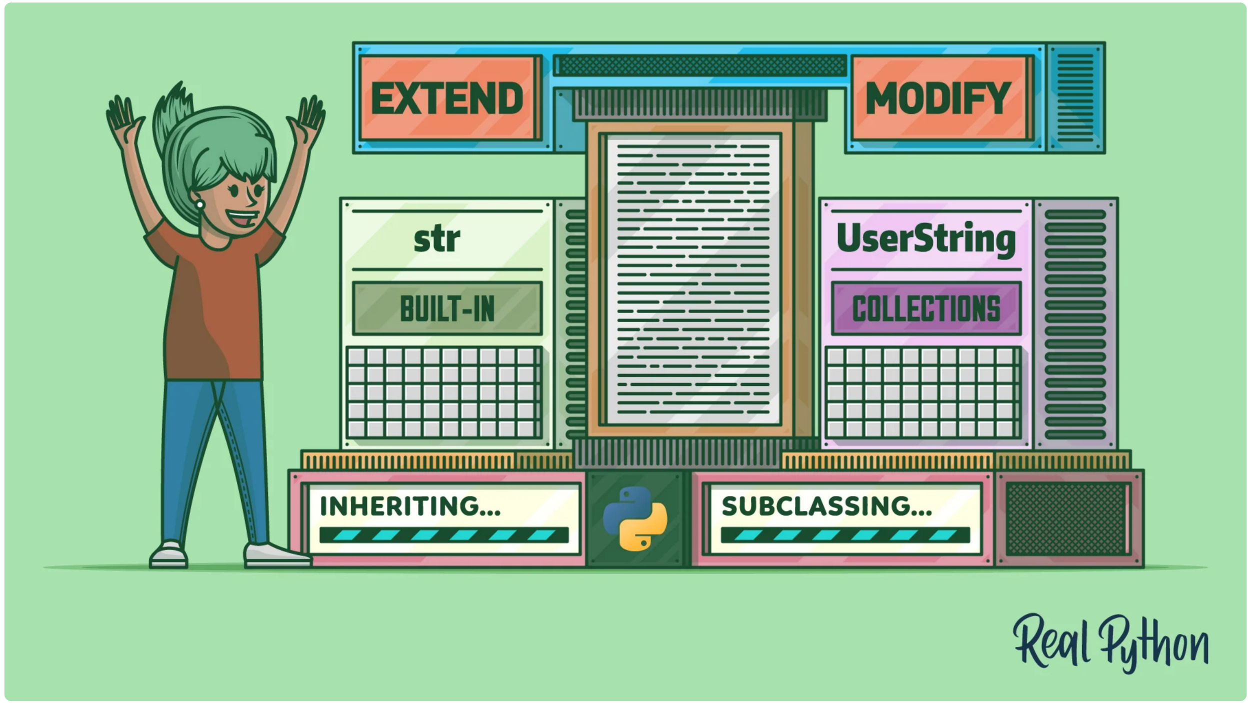

Custom Python Strings: Inheriting From str vs UserString

by Leodanis Pozo Ramos Sep 28, 2022 0 Comments intermediate python

Table of Contents

- Creating String-Like Classes in Python
- Inheriting From Python’s Built-in str Class
  - Extending the String’s Standard Behavior

- Modifying the String's Standard Behavior
- Tweaking the Instantiation Process of str
- Subclassing UserString From collections
  - Extending and Modifying the String's Standard Behavior
  - Tweaking the Instantiation Process of UserString
- Simulating Mutations in Your String-Like Classes
- Conclusion

The Python **`str`** class has many useful features that can help you out when you're processing **text** or **strings** in your code. However, in some situations, all these great features may not be enough for you. You may need to create **custom string-like classes**. To do this in Python, you can inherit from the built-in `str` class directly or subclass `UserString`, which lives in the `collections` module.

In this tutorial, you'll learn how to:

- Create custom string-like classes by inheriting from the **built-in `str` class**
- Build custom string-like classes by subclassing **`UserString`** from the **`collections` module**
- Decide when to use `str` or `UserString` to create **custom string-like classes**

Meanwhile, you'll write a few examples that'll help you decide whether to use `str` or `UserString` when you're creating your custom string classes. Your choice will mostly depend on your specific use case.

To follow along with this tutorial, it'll help if you're familiar with Python's built-in `str` class and its standard features. You'll also need to know the basics of object-oriented

Table of Contents

- Creating String-Like Classes in Python
- Inheriting From Python's Built-in str Class
- Subclassing UserString From collections
- Simulating Mutations in Your String-Like Classes
- Conclusion

programming and inheritance in Python.

Sample Code: Click here to download the free sample code that you'll use to create custom string-like classes.

Creating String-Like Classes in Python

The built-in `str` class allows you to create **strings** in Python. Strings are sequences of characters that you'll use in many situations, especially when working with textual data. From time to time, the standard functionalities of Python's `str` may be insufficient to fulfill your needs. So, you may want to create custom string-like classes that solve your specific problem.

You'll typically find at least two reasons for creating custom string-like classes:

1. **Extending** the regular string by adding new functionality
2. **Modifying** the standard string's functionality

You can also face situations in which you need to both extend *and* modify the standard functionality of strings at the same time.

In Python, you'll commonly use one of the following techniques to create your string-like classes. You can inherit from the Python built-in `str` class directly or subclass `UserString` from `collections`.

Note: In object-oriented programming, it's common practice to use the verbs **inherit** and **subclass** interchangeably.

One relevant feature of Python strings is immutability, which means that you can't modify them in place. So, when selecting the appropriate technique to create your own custom string-like classes, you need to consider whether your desired features will affect immutability or not.

For example, if you need to modify the current behavior of existing string methods, then you'll probably be okay subclassing `str`. In contrast, if you need to change how strings are created, then inheriting from `str` will demand advanced knowledge. You'll have to override the `.__new__()` method. In this latter case, inheriting from `UserString` may make your life easier because you won't have to touch `.__new__()`.

In the upcoming sections, you'll learn the pros and cons of each technique so that you can decide which is the best strategy to use for your specific problem.

Inheriting From Python's Built-in `str` Class

For a long time, it was impossible to inherit directly from Python types implemented in C. Python 2.2 fixed this issue. Now you can subclass built-in types, including `str`. This new feature is quite convenient when you need to create custom string-like classes.

By inheriting from `str` directly, you can extend and modify the standard behavior of this built-in class. You can also tweak the instantiation process of your custom string-like classes to perform transformations before new instances are ready.

Extending the String's Standard Behavior

An example of requiring a custom string-like class is when you need to extend the standard Python strings with new behavior. For example, say that you need a string-like class that implements a new method to count the number of words in the underlying string.

In this example, your custom string will use the whitespace character as its default word separator. However, it should also allow you to provide a specific separator character. To code a class that fulfills these needs, you can do something like this:

```python
>>> class WordCountString(str):
...     def words(self, separator=None):
...         return len(self.split(separator))
...
```

This class inherits from `str` directly. This means that it provides the same interface as its parent class.

On top of this inherited interface, you add a new method called `.words()`. This method takes a `separator` character as an argument that is passed on to `.split()`. Its default value is `None` which will split on runs of consecutive whitespace. Then you call `.split()` with the target separator to split the underlying string into words. Finally, you use the `len()` function to determine the word count.

Here's how you can use this class in your code:

```python
>>> sample_text = WordCountString(
...     """Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime
...     mollitia, molestiae quas vel sint commodi repudiandae consequuntur
...     voluptatum laborum numquam blanditiis harum quisquam eius sed odit
...     fugiat iusto fuga praesentium optio, eaque rerum! Provident similique
...     accusantium nemo autem. Veritatis obcaecati tenetur iure eius earum
...     ut molestias architecto voluptate aliquam nihil, eveniet aliquid
...     culpa officia aut! Impedit sit sunt quaerat, odit, tenetur error,
...     harum nesciunt ipsum debitis quas aliquid."""
... )

>>> sample_text.words()
```

```
68
```

Cool! Your `.words()` methods works fine. It splits the input text into words and then returns the word count. You can modify how this method delimits and processes words, but the current implementation works okay for this demonstrative example.

In this example, you haven't modified the standard behavior of Python's `str`. You've just added new behavior to your custom class. However, it's also possible to change the default behavior of `str` by overriding any of its default methods, as you'll explore next.

Modifying the String's Standard Behavior

To learn how to modify the standard behavior of `str` in a custom string-like class, say that you need a string class that always prints its letters in uppercase. You can do this by overriding the `.__str__()` special method, which takes care of how string objects are printed.

Here's an `UpperPrintString` class that behaves as you need:

Python >>>

```python
>>> class UpperPrintString(str):
...     def __str__(self):
...         return self.upper()
...
```

Again, this class inherits from `str`. The `.__str__()` method returns a copy to the underlying string, `self`, with all of its letters in uppercase. To transform the letters, you use the `.upper()` method.

To try out your custom string-like class, go ahead and run the following code:

Python >>>

```
>>> sample_string = UpperPrintString("Hello, Pythonista!")

>>> print(sample_string)
HELLO, PYTHONISTA!

>>> sample_string
'Hello, Pythonista!'
```

When you print an instance of `UpperPrintString`, you get the string in uppercase letters on your screen. Note that the original string wasn't modified or affected. You only changed the standard printing feature of `str`.

Tweaking the Instantiation Process of `str`

In this section, you'll do something different. You'll create a string-like class that transforms the original input string before making the final string object. For example, say that you need a string-like class that stores all of its letters in lowercase. To do this, you'll try to override the class initializer, `.__init__()`, and do something like this:

Python >>>

```
>>> class LowerString(str):
...     def __init__(self, string):
...         super().__init__(string.lower())
...
```

In this code snippet, you provide an `.__init__()` method that overrides the default `str` initializer. Inside this `.__init__()` implementation, you use `super()` to access the parent class's `.__init__()` method. Then you call `.lower()` on the input string to convert all of its letters into lowercase letters before initializing the current string.

However, the above code doesn't work, as you'll confirm in the following example:

Python >>>

```python
>>> sample_string = LowerString("Hello, Pythonista!")
Traceback (most recent call last):
    ...
TypeError: object.__init__() takes exactly one argument...
```

Since `str` objects are immutable, you can't change their value in `.__init__()`. This is because the value is set during object creation and not during object initialization. The only way to transform the value of a given string during the instantiation process is to override the `.__new__()` method.

Here's how to do this:

Python >>>

```python
>>> class LowerString(str):
...     def __new__(cls, string):
...         instance = super().__new__(cls, string.lower())
...         return instance
...

>>> sample_string = LowerString("Hello, Pythonista!")
>>> sample_string
'hello, pythonista!'
```

In this example, your `LowerString` class overrides the super class's `.__new__()` method to customize how instances are created. In this case, you transform the input string before

creating the new `LowerString` object. Now your class works as you need it to. It takes a string as input and stores it as a lowercase string.

If you ever need to transform the input string at instantiation time, then you'll have to override `.__new__()`. This technique will require advanced knowledge of Python's data model and special methods.

Subclassing `UserString` From `collections`

The second tool that allows you to create custom string-like classes is the `UserString` class from the `collections` module. This class is a wrapper around the built-in `str` type. It was designed to develop string-like classes when it wasn't possible to inherit from the built-in `str` class directly.

The possibility of directly subclassing `str` means you might have less need for `UserString`. However, this class is still available in the standard library, both for convenience and backward compatibility. In practice, this class also has some hidden features that can be helpful, as you'll learn soon.

The most relevant feature of `UserString` is its `.data` attribute, which gives you access to the wrapped string object. This attribute can facilitate the creation of custom strings, especially in cases where your desired customization affects the string mutability.

In the following two sections, you'll revisit the examples from previous sections, but this time you'll be subclassing `UserString` instead of `str`. To kick things off, you'll start by extending and modifying the standard behavior of Python strings.

Extending and Modifying the String's Standard Behavior

Instead of subclassing the built-in `str` class, you could implement `WordCountString` and

Instead of subclassing the built-in `str` class, you could implement `WordCountString` and `UpperPrintString` by inheriting from the `UserString` class. This new implementation will

only require you to change the superclass. You won't have to change the original internal implementation of your classes.

Here are new versions of `WordCountString` and `UpperPrintString`:

```python
>>> from collections import UserString

>>> class WordCountString(UserString):
...     def words(self, separator=None):
...         return len(self.split(separator))
...

>>> class UpperPrintString(UserString):
...     def __str__(self):
...         return self.upper()
...
```

The only difference between these new implementations and the original ones is that now you're inheriting from `UserString`. Note that inheriting from `UserString` requires you to import the class from the `collections` module.

If you try out these classes with the same examples as before, then you'll confirm that they work the same as their equivalent classes based on `str`:

```
Python                                                                  >>>

>>> sample_text = WordCountString(
...     """Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime
...     mollitia, molestiae quas vel sint commodi repudiandae consequuntur
...     voluptatum laborum numquam blanditiis harum quisquam eius sed odit
...     fugiat iusto fuga praesentium optio, eaque rerum! Provident similique
...     accusantium nemo autem. Veritatis obcaecati tenetur iure eius earum
...     ut molestias architecto voluptate aliquam nihil, eveniet aliquid
...     culpa officia aut! Impedit sit sunt quaerat, odit, tenetur error,
...     harum nesciunt ipsum debitis quas aliquid."""
... )

>>> sample_text.words()
68

>>> sample_string = UpperPrintString("Hello, Pythonista!")
>>> print(sample_string)
HELLO, PYTHONISTA!

>>> sample_string
'Hello, Pythonista!'
```

In these examples, your new implementations of `WordCountString` and `UpperPrintString` work the same as the old ones. So, why should you use `UserString` rather than `str`? Up to this point, there’s no apparent reason for doing this. However, `UserString` comes in handy when you need to modify how your strings are created.

Tweaking the Instantiation Process of `UserString`

You can code the `LowerString` class by inheriting from `UserString`. By changing the parent class, you'll be able to customize the initialization process in the instance initializer, `.__init__()`, without overriding the instance creator, `.__new__()`.

Here's your new version of `LowerString` and how it works in practice:

```python
>>> from collections import UserString

>>> class LowerString(UserString):
...     def __init__(self, string):
...         super().__init__(string.lower())
...

>>> sample_string = LowerString("Hello, Pythonista!")
>>> sample_string
'hello, pythonista!'
```

In the example above, you've made running transformations on the input string possible by using `UserString` instead of `str` as your superclass. The transformations are possible because `UserString` is a wrapper class that stores the final string in its `.data` attribute, which is the real immutable object.

Because `UserString` is a wrapper around the `str` class, it provides a flexible and straightforward way to create custom strings with mutable behaviors. Providing mutable behaviors by inheriting from `str` is complicated because of the class's natural immutability condition.

In the following section, you'll use `UserString` to create a string-like class that simulates a **mutable** string data type.

Simulating Mutations in Your String-Like Classes

As a final example of why you should have `UserString` in your Python tool kit, say that you need a mutable string-like class. In other words, you need a string-like class that you can modify in place.

Unlike lists and dictionaries, strings don't provide the `.__setitem__()` special method, because they're immutable. Your custom string will need this method to allow you to update characters and slices by their indices using an assignment statement.

Your string-like class will also need to change the standard behavior of common string methods. To keep this example short, you'll only modify the `.upper()` and `.lower()` methods. Finally, you'll provide a `.sort()` method to sort your string in place.

Standard string methods don't mutate the underlying string. They return a new string object with the required transformation. In your custom string, you need the methods to perform their changes in place.

To achieve all these goals, fire up your favorite code editor, create a file named `mutable_string.py`, and write the following code:

Python

```python
 1 # mutable_string.py
 2 
 3 from collections import UserString
 4 
 5 class MutableString(UserString):
 6     def __setitem__(self, index, value):
 7         data_as_list = list(self.data)
 8         data_as_list[index] = value
 9         self.data = "".join(data_as_list)
10 
11     def __delitem__(self, index):
12         data_as_list = list(self.data)
13         del data_as_list[index]
14         self.data = "".join(data_as_list)
15 
16     def upper(self):
17         self.data = self.data.upper()
18 
19     def lower(self):
20         self.data = self.data.lower()
21 
22     def sort(self, key=None, reverse=False):
23         self.data = "".join(sorted(self.data, key=key, reverse=reverse))
```

Here’s how this code works line by line:

- **Line 3** imports `UserString` from `collections`.
- **Line 5** creates `MutableString` as a subclass of `UserString`.
- **Line 6** defines `.__setitem__()`. Python calls this special method whenever you run an assignment operation on a sequence using an index, like in `sequence[0] = value`. This implementation of `.__setitem__()` turns `.data` into a list, replaces the item at `index`

implementation of `.__setitem__()` turns `.data` into a list, replaces the item at `index` with `value`, builds the final string using `.join()`, and assigns its value back to `.data`. The whole process simulates an in-place transformation or mutation.

- **Line 11** defines `.__delitem__()`, the special method that allows you to use the `del` statement for removing characters by index from your mutable string. It's implemented similar to `.__setitem__()`. On line 13, you use `del` to delete items from the temporary list.

- **Line 16** overrides `UserString.upper()` and calls `str.upper()` on `.data`. Then it stores the result back in `.data`. Again, this last operation simulates an in-place mutation.

- **Line 19** overrides `UserString.lower()` using the same technique as in `.upper()`.

- **Line 22** defines `.sort()`, which combines the built-in `sorted()` function with the `str.join()` method to create a sorted version of the original string. Note that this method has the same signature as `list.sort()` and the built-in `sorted()` function.

That's it! Your mutable string is ready! To try it out, get back to your Python shell and run the following code:

```
Python                                                                >>>
>>> from mutable_string import MutableString

>>> sample_string = MutableString("ABC def")
>>> sample_string
'ABC def'

>>> sample_string[4] = "x"
>>> sample_string[5] = "y"
>>> sample_string[6] = "z"
>>> sample_string
'ABC xyz'

>>> del sample_string[3]
>>> sample_string
'ABCxyz'

>>> sample_string.upper()
>>> sample_string
'ABCXYZ'

>>> sample_string.lower()
>>> sample_string
'abcxyz'

>>> sample_string.sort(reverse=True)
>>> sample_string
'zyxcba'
```

Great! Your new mutable string-like class works as expected. It allows you to modify the underlying string in place, as you would do with a mutable sequence. Note that this example covers a few string methods only. You can play with other methods and continue providing your class with new mutability features.

Conclusion

You've learned to create **custom string-like classes** with new or modified behaviors. You've done this by subclassing the built-in `str` class directly and by inheriting from `UserString`, which is a convenient class available in the `collections` module.

Inheriting from `str` and subclassing `UserString` are both suitable options when it comes to creating your own string-like classes in Python.

In this tutorial, you've learned how to:

- Create string-like classes by inheriting from the **built-in `str` class**
- Build string-like classes by subclassing **`UserString`** from the **`collections` module**
- Decide when to subclass `str` or `UserString` to create your **custom string-like classes**

Now you're ready to write custom string-like classes, which will allow you to leverage the full power of this valuable and commonplace data type in Python.

Sample Code: Click here to download the free sample code that you'll use to create custom string-like classes.

```
>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Bartosz

Geir Arne

Kate

Philipp

What Do You Think?

Rate this article:

Tweet Share in Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate python