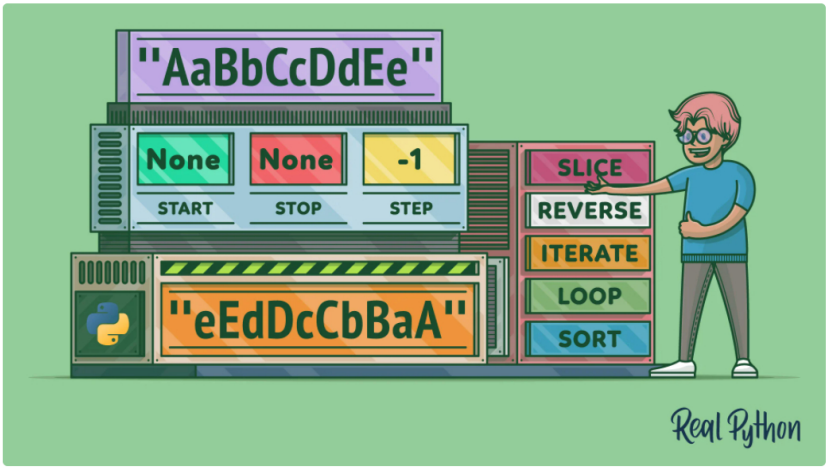

Reverse Strings in Python: reversed(), Slicing, and More

by Leodanis Pozo Ramos Sep 27, 2021 0 Comments basics best-practices python

Mark as Completed

Table of Contents

When you're using Python strings often in your code, you may face the need to work with them in **reverse order**. Python includes a few handy tools and techniques that can help you out in these situations. With them, you'll be able to build reversed copies of existing strings quickly and efficiently.

Knowing about these tools and techniques for reversing strings in Python will help you improve your proficiency as a Python developer.

In this tutorial, you'll learn how to:

- Quickly build reversed strings through **slicing**
- Create **reversed copies** of existing strings using `reversed()` and `.join()`
- Use **iteration** and **recursion** to reverse existing strings manually
- Perform **reverse iteration** over your strings
- **Sort** your strings in reverse order using `sorted()`

Table of Contents
- Reversing Strings With Core Python Tools
- Generating Reversed Strings by Hand
- Iterating Through Strings in Reverse
- Creating a Custom Reversible String
- Sorting Python Strings in Reverse Order
- Conclusion

Mark as Completed

Tweet Share Email

To make the most out of this tutorial, you should know the basics of strings, `for` and `while` loops, and recursion.

> **Free Download: Get a sample chapter from Python Basics: A Practical Introduction to Python 3** to see how you can go from beginner to intermediate in Python with a complete curriculum, up-to-date for Python 3.8.

Reversing Strings With Core Python Tools

Working with Python strings in **reverse order** can be a requirement in some particular situations. For example, say you have a string `"ABCDEF"` and you want a fast way to reverse it to get `"FEDCBA"`. What Python tools can you use to help?

Strings are immutable in Python, so reversing a given string in place isn't possible. You'll need to create reversed **copies** of your target strings to meet the requirement.

Python provides two straightforward ways to reverse strings. Since strings are sequences, they're **indexable**, **sliceable**, and **iterable**. These features allow you to use slicing to directly generate a copy of a given string in reverse order. The second option is to use the built-in function `reversed()` to create an iterator that yields the characters of an input string in reverse order.

Reversing Strings Through Slicing

Slicing is a useful technique that allows you to extract items from a given sequence using different combinations of **integer indices** known as offsets. When it comes to slicing strings, these offsets define the index of the first character in the slicing, the index of the character that stops the slicing, and a value that defines how many characters you want to jump through in each iteration.

To slice a string, you can use the following syntax:

```
Python
a_string[start:stop:step]
```

Your offsets are `start`, `stop`, and `step`. This expression extracts all the characters from `start` to `stop - 1` by `step`. You're going to look more deeply at what all this means in just a moment.

All the offsets are optional, and they have the following default values:

Offset	Default Value
`start`	`0`
`stop`	`len(a_string)`
`step`	`1`

Here, `start` represents the index of the first character in the slice, while `stop` holds the index that stops the slicing operation. The third offset, `step`, allows you to decide how many characters the slicing will jump through on each iteration.

> **Note:** A slicing operation finishes when it reaches the index equal to or greater than `stop`. This means that it never includes the item at that index, if any, in the final slice.

The `step` offset allows you to fine-tune how you extract desired characters from a string while skipping others:

while skipping others.

```
>>> letters = "AaBbCcDd"

>>> # Get all characters relying on default offsets
>>> letters[::]
'AaBbCcDd'
>>> letters[:]
'AaBbCcDd'

>>> # Get every other character from 0 to the end
>>> letters[::2]
'ABCD'

>>> # Get every other character from 1 to the end
>>> letters[1::2]
'abcd'
```

Here, you first slice `letters` without providing explicit offset values to get a full copy of the original string. To this end, you can also use a slicing that omits the second colon (`:`). With `step` equal to `2`, the slicing gets every other character from the target string. You can play around with different offsets to get a better sense of how slicing works.

Why are slicing and this third offset relevant to reversing strings in Python? The answer lies in how `step` works with negative values. If you provide a negative value to `step`, then the slicing runs backward, meaning from right to left.

For example, if you set `step` equal to `-1`, then you can build a slice that retrieves all the characters in reverse order:

```
>>> letters = "ABCDEF"

>>> letters[::-1]
'FEDCBA'

>>> letters
'ABCDEF'
```

This slicing returns all the characters from the right end of the string, where the index is equal to `len(letters) - 1`, back to the left end of the string, where the index is `0`. When you use this trick, you get a copy of the original string in reverse order without affecting the original content of `letters`.

Another technique to create a reversed copy of an existing string is to use `slice()`. The signature of this built-in function is the following:

```
slice(start, stop, step)
```

This function takes three arguments, with the same meaning of the offsets in the slicing operator, and returns a slice object representing the set of indices that result from calling `range(start, stop, step)`.

You can use `slice()` to emulate the slicing `[::-1]` and reverse your strings quickly. Go ahead and run the following call to `slice()` inside square brackets:

```
>>> letters = "ABCDEF"

>>> letters[slice(None, None, -1)]
'FEDCBA'
```

Passing `None` to the first two arguments of `slice()` tells the function that you want to rely on its internal default behavior, which is the same as a standard slicing with no values for `start` and `stop`. In other words, passing `None` to `start` and `stop` means that you want a slice from the left end to the right end of the underlying sequence.

Reversing Strings With `.join()` and `reversed()`

The second and arguably the most Pythonic approach to reversing strings is to use `reversed()` along with `str.join()`. If you pass a string to `reversed()`, you get an iterator that yields characters in reverse order:

```
Python >>>
>>> greeting = reversed("Hello, World!")

>>> next(greeting)
'!'
>>> next(greeting)
'd'
>>> next(greeting)
'l'
```

When you call `next()` with `greeting` as an argument, you get each character from the right end of the original string.

An important point to note about `reversed()` is that the resulting iterator yields characters directly from the original string. In other words, it doesn't create a new reversed string but reads characters backward from the existing one. This behavior is fairly efficient in terms of memory consumption and can be a fundamental win in some contexts and situations, such as iteration.

You can use the iterator that you get from calling `reversed()` directly as an argument to `.join()`:

```
Python >>>
>>> "".join(reversed("Hello, World!"))
'!dlroW ,olleH'
```

In this single-line expression, you pass the result of calling `reversed()` directly as an argument to `.join()`. As a result, you get a reversed copy of the original input string. This combination of `reversed()` and `.join()` is an excellent option for reversing strings.

Generating Reversed Strings by Hand

So far, you've learned about core Python tools and techniques to reverse strings quickly. Most of the time, they'll be your way to go. However, you might need to reverse a string by hand at some point in your coding adventure.

In this section, you'll learn how to reverse strings using explicit loops and recursion. The final technique uses a functional programming approach with the help of Python's `reduce()` function.

Reversing Strings in a Loop

The first technique you'll use to reverse a string involves a `for` loop and the concatenation operator (`+`). With two strings as operands, this operator returns a new string that results from joining the original ones. The whole operation is known as **concatenation**.

> **Note:** Using `.join()` is the recommended approach to concatenate strings in Python. It's clean, efficient, and Pythonic.

Here's a function that takes a string and reverses it in a loop using concatenation:

```
Python >>>
>>> def reversed_string(text):
```

```
...     result = ""
...     for char in text:
...         result = char + result
...     return result
...

>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:


```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

```
>>> reversed_string("Hello, World!")
'!dlroW ,olleH'
```

In every iteration, the loop takes a subsequent character, `char`, from `text` and concatenates it with the current content of `result`. Note that `result` initially holds an empty string (`""`). The new intermediate string is then reassigned to `result`. At the end of the loop, `result` holds a new string as a reversed copy of the original one.

> **Note:** Since Python strings are immutable data types, you should keep in mind that the examples in this section use a wasteful technique. They rely on creating successive intermediate strings only to throw them away in the next iteration.

If you prefer using a `while` loop, then here's what you can do to build a reversed copy of a given string:

Here's how you can iterate over a string in reverse order with `reversed()`:

```
Python >>>
>>> greeting = "Hello, World!"

>>> for char in reversed(greeting):
...     print(char)
...
!
d
l
r
o
W

,
o
l
l
e
H

>>> reversed(greeting)
<reversed object at 0x7f17aa89e070>
```

The `for` loop in this example is very readable. The name of `reversed()` clearly expresses its intent and communicates that the function doesn't produce any side effects on the input data. Since `reversed()` returns an iterator, the loop is also efficient regarding memory usage.

The Slicing Operator, `[::-1]`

The second approach to perform reverse iteration over strings is to use the extended slicing syntax you saw before in the `a_string[::-1]` example. Even though this approach won't favor memory efficiency and readability, it still provides a quick way to iterate over a reversed copy of an existing string:

```
Python >>>
>>> greeting = "Hello, World!"

>>> for char in greeting[::-1]:
...     print(char)
...
!
d
l
r
```

The Slicing Operator, `[::-1]`

The second approach to perform reverse iteration over strings is to use the extended slicing syntax you saw before in the `a_string[::-1]` example. Even though this approach won't favor memory efficiency and readability, it still provides a quick way to iterate over a reversed copy of an existing string:

```
Python >>>
>>> greeting = "Hello, World!"

>>> for char in greeting[::-1]:
...     print(char)
...
!
d
```

```
l
r
```

The Slicing Operator, `[::-1]`

The second approach to perform reverse iteration over strings is to use the extended slicing syntax you saw before in the `a_string[::-1]` example. Even though this approach won't favor memory efficiency and readability, it still provides a quick way to iterate over a reversed copy of an existing string:

Python >>>

```
>>> greeting = "Hello, World!"

>>> for char in greeting[::-1]:
...     print(char)
...
!
d
l
r
```

The Slicing Operator, `[::-1]`

The second approach to perform reverse iteration over strings is to use the extended slicing syntax you saw before in the `a_string[::-1]` example. Even though this approach won't favor memory efficiency and readability, it still provides a quick way to iterate over a reversed copy of an existing string:

Python >>>

```
>>> greeting = "Hello, World!"

>>> for char in greeting[::-1]:
...     print(char)
...
!
d
l
r
>>> text
'Hello, World!'

>>> # Reverse the string in place
>>> text.reverse()
>>> text
'!dlroW ,olleH'
```

When you call `.reverse()` on `text`, the method acts as if you're doing an in-place mutation of the underlying string. However, you're actually creating a new string and assigning it back to the wrapped string. Note that `text` now holds the original string in reverse order.

Since `UserString` provides the same functionality as its superclass `str`, you can use `reversed()` out of the box to perform reverse iteration:

Python >>>

```
>>> text = ReversibleString("Hello, World!")

>>> # Support reverse iteration out of the box
>>> for char in reversed(text):
...     print(char)
...
!
d
l
r
o
W

,
o
l
l
e
```

```
H

>>> text
"Hello, World!"
```

Here, you call `reversed()` with `text` as an argument to feed a `for` loop. This call works as expected and returns the corresponding iterator because `UserString` inherits the required behavior from `str`. Note that calling `reversed()` doesn't affect the original string.

Sorting Python Strings in Reverse Order

The last topic you'll learn about is how to sort the characters of a string in reverse order. This can be handy when you're working with strings in no particular order and you need to sort them in reverse alphabetical order.

To approach this problem, you can use `sorted()`. This built-in function returns a list containing all the items of the input iterable in order. Besides the input iterable, `sorted()` also accepts a `reverse` keyword argument. You can set this argument to `True` if you want the input iterable to be sorted in descending order:

Python >>>

```
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use `.join()` just like you did in earlier sections:

input iterable to be sorted in descending order:

Python >>>

```
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use `.join()` just like you did in earlier sections:

input iterable to be sorted in descending order:

Python >>>

```
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use

.join() just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use .join() just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use .join() just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use .join() just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use .join() just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use `.join()` just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

When you call `sorted()` with a string as an argument and `reverse` set to `True`, you get a list containing the characters of the input string in reverse or descending order. Since `sorted()` returns a `list` object, you need a way to turn that list back into a string. Again, you can use `.join()` just like you did in earlier sections:

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
>>> sorted(vowels, reverse=True)
```

input iterable to be sorted in descending order:

```
Python                                                    >>>
>>> vowels = "eauoi"

>>> # Sort in ascending order
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']

>>> # Sort in descending order
```