- Convert a string to upper or lower case
- Remove whitespace from string
- Determine if a string begins and ends with certain characters

Let's go!

Converting String Case

To convert a string to all lower case letters, you use the string's `.lower()` method. This is done by tacking `.lower()` on to the end of the string itself:

```
>>> "Jean-luc Picard".lower()
'jean-luc picard'
```

The dot (`.`) tells Python that what follows is the name of a method—the `lower()` method in this case.

Note

We will refer to the names of string methods with a dot at the beginning of them. So, for example, the `.lower()` method is written with a dot, instead of `lower()`.

The reason we do this is to make it easy to spot functions that are string methods, as opposed to built-in functions like `print()` and `type()`.

String methods don't just work on string literals. You can also use the `.lower()` method on a string assigned to a variable:

```
>>> name = "Jean-luc Picard"
>>> name.lower()
'jean-luc picard'
```

The opposite of the `.lower()` method is the `.upper()` method, which converts every character in a string to upper case:

```
>>> loud_voice = "Can you hear me yet?"
>>> loud_voice.upper()
'CAN YOU HEAR ME YET?'
```

Compare the `.upper()` and `.lower()` string methods to the general-purpose `len()` function you saw in the last section. Aside from the different results of these functions, the important distinction here is how they are used.

The `len()` function is a stand-alone function. If you want to determine the length of the `loud_voice` string, you call the `len()` function directly, like this:

```
>>> len(loud_voice)
20
```

On the other hand, `.upper()` and `.lower()` must be used in conjunction with a string. They do not exist independently.

Removing Whitespace From a String

Whitespace is any character that is printed as blank space. This includes things like spaces and **line feeds**, which are special characters that move output to a new line.

Sometimes you need to remove whitespace from the beginning or end of a string. This is especially useful when working with strings that come from user input, where extra whitespace characters may have been introduced by accident.

There are three string methods that you can use to remove whitespace from a string:

1. `.rstrip()`
2. `.lstrip()`
3. `.strip()`

`.rstrip()` removes whitespace from the right side of a string:

```
>>> name = "Jean-luc Picard     "
>>> name
'Jean-luc Picard     '
>>> name.rstrip()
'Jean-luc Picard'
```

In this example, the string `"Jean-luc Picard     "` has five trailing spaces. Python doesn't remove any trailing spaces in a string automatically when the string is assigned to a variable. The `.rstrip()` method removes trailing spaces from the right-hand side of the string and returns a new string `"Jean-luc Picard"`, which no longer has the spaces at the end.

The `.lstrip()` method works just like `.rstrip()`, except that it removes whitespace from the left-hand side of the string:

```
>>> name = "     Jean-luc Picard"
>>> name
'     Jean-luc Picard'
>>> name.lstrip()
'Jean-luc Picard'
```

To remove whitespace from both the left and the right sides of the string at the same time, use the `.strip()` method:

```
>>> name = "     Jean-luc Picard     "
>>> name
'     Jean-luc Picard     '
>>> name.strip()
'Jean-luc Picard'
```

Note

None of the `.rstrip()`, `.lstrip()`, and `.strip()` methods remove whitespace from the middle of the string. In each of the previous examples the space between "Jean-luc" and "Picard" is always preserved.

Determine if a String Starts or Ends With a Particular String

When you work with text, sometimes you need to determine if a given string starts with or ends with certain characters. You can use two string methods to solve this problem: `.startswith()` and `.endswith()`.

Let's look at an example. Consider the string `"Enterprise"`. Here's how you use `.startswith()` to determine if the string starts with the letters `e` and `n`:

```
>>> starship = "Enterprise"
>>> starship.startswith("en")
False
```

You must tell `.startswith()` what characters to search for by providing a string containing those characters. So, to determine if `"Enterprise"` starts with the letters `e` and `n`, you call `.startswith("en")`. This returns `False`. Why do you think that is?

If you guessed that `.startswith("en")` returns `False` because `"Enterprise"` starts with a capital `E`, you're absolutely right! The `.startswith()` method is **case-sensitive**. To get `.startswith()` to return `True`, you need to provide it with the string `"En"`:

```
>>> starship.startswith("En")
True
```

The `.endswith()` method is used to determine if a string ends with certain characters:

```
>>> starship.endswith("rise")
True
```

Just like `.startswith()`, the `.endswith()` method is case-sensitive:

```
>>> starship.endswith("risE")
False
```

Note

The `True` and `False` values are not strings. They are a special kind of data type called a **Boolean value**. You will learn more about Boolean values in Chapter 8.

String Methods and Immutability

Recall from the previous section that strings are immutable—they can't be changed once they have been created. Most string methods that alter a string, like `.upper()` and `.lower()`, actually return copies of the original string with the appropriate modifications.

If you aren't careful, this can introduce subtle bugs into your program. Try this out in IDLE's interactive window:

```
>>> name = "Picard"
>>> name.upper()
'PICARD'
>>> name
'Picard'
```

When you call `name.upper()`, nothing about `name` actually changes. If you need to keep the result, you need to assign it to a variable:

```
>>> name = "Picard"
>>> name = name.upper()
>>> name
'PICARD'
```

`name.upper()` returns a new string `"PICARD"`, which is re-assigned to the `name` variable. This **overrides** the original string `"Picard"` assigned to `"name"`.

Use IDLE to Discover Additional String Methods

Strings have lots of methods associated to them. The methods introduced in this section barely scratch the surface. IDLE can help you

find new string methods. To see how, first assign a string literal to a variable in the interactive window:

```
>>> starship = "Enterprise"
```

Next, type `starship` followed by a period, but do not hit `Enter`. You should see the following in the interactive window:

```
>>> starship.
```

Now wait for a couple of seconds. IDLE displays a list of every string method that you can scroll through with the arrow keys.

A related shortcut in IDLE is the ability to fill in text automatically without having to type in long names by hitting `Tab`. For instance, if you only type in `starship.u` and then hit the `Tab` key, IDLE automatically fills in `starship.upper` because there is only one method belonging to `starship` that begins with a `u`.

This even works with variable names. Try typing in just the first few letters of `starship` and, assuming you don't have any other names already defined that share those first letters, IDLE completes the name `starship` for you when you hit the `Tab` key.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a script that converts the following strings to lowercase: `"Animals"`, `"Badger"`, `"Honey Bee"`, `"Honeybadger"`. Print each lowercase string on a separate line.
2. Repeat Exercise 1, but convert each string to uppercase instead of lowercase.
3. Write a script that removes whitespace from the following strings:

```
string1 = "    Filet Mignon"
string2 = "Brisket    "
```

```
string3 = "  Cheeseburger   "
```

Print out the strings with the whitespace removed.

4. Write a script that prints out the result of `.startswith("be")` on each of the following strings:

```
string1 = "Becomes"
string2 = "becomes"
string3 = "BEAR"
string4 = "  bEautiful"
```

5. Using the same four strings from Exercise 4, write a script that uses string methods to alter each string so that `.startswith("be")` returns `True` for all of them.

Leave feedback on this section »

4.4 Interact With User Input

Now that you've seen how to work with string methods, let's make things interactive. In this section, you will learn how to get some input from a user with the `input()` function. You'll write a program that asks a user to input some text and then display that text back to them in uppercase.

Enter the following into IDLE's interactive window:

```
>>> input()
```

When you press Enter, it looks like nothing happens. The cursor moves to a new line, but a new `>>>` doesn't appear. Python is waiting for you to enter something!

Go ahead and type some text and press Enter:

```
>>> input()
Hello there!
'Hello there!'
>>>
```

The text you entered is repeated on a new line with single quotes. That's because `input()` returns any text entered by the user as a string.

To make `input()` a bit more user friendly, you can give it a **prompt** to display to the user. The prompt is just a string that you put in between the parentheses of `input()`. It can be anything you want: a word, a symbol, a phrase—anything that is a valid Python string.

The `input()` function displays the prompt and waits for the user to type something on their keyboard. When the user hits `Enter`, `input()` returns their input as a string that can be assigned to a variable and used to do something in your program.

To see how `input()` works, save and run the following script:

```
prompt = "Hey, what's up? "
user_input = input(prompt)
print("You said:", user_input)
```

When you run this script, you'll see `Hey, what's up?` displayed in the interactive window with a blinking cursor.

The single space at the end of the string `"Hey, what's up? "` makes sure that when the user starts to type, the text is separated from the prompt with a space. When you type a response and press Enter, your response is assigned to the `user_input` variable.

Here's a sample run of the program:

```
Hey, what's up? Mind your own business.


You said: Mind your own business.
```

Once you have input from a user, you can do something with it. For example, the following script takes user input and converts it to uppercase with `.upper()` and prints the result:

```
response = input("What should I shout? ")
shouted_response = response.upper()
print("Well, if you insist...", shouted_response)
```

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a script that takes input from the user and displays that input back.
2. Write a script that takes input from the user and displays the input in lowercase.
3. Write a script that takes input from the user and displays the number of characters inputted.

Leave feedback on this section »

4.5 Challenge: Pick Apart Your User's Input

Write a script named `first_letter.py` that first prompts the user for input by using the string `"Tell me your password:"` The script should then determine the first letter of the user's input, convert that letter to upper-case, and display it back.

For example, if the user input is `"no"` then the program should respond like this:

```
The first letter you entered was: N
```

For now, it's okay if your program crashes when the user enters nothing as input—that is, they just hit `Enter` instead of typing something in. You'll learn about a couple of ways you can deal with this situation in an upcoming chapter.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources.

Leave feedback on this section »

4.6 Working With Strings and Numbers

When you get user input using the `input()` function, the result is always a string. There are many other times when input is given to a program as a string. Sometimes those strings contain numbers that need to be fed into calculations.

In this section you will learn how to deal with strings of numbers. You will see how arithmetic operations work on strings, and how they often lead to surprising results. You will also learn how to convert between strings and number types.

Strings and Arithmetic Operators

You've seen that string objects can hold many types of characters, including numbers. However, don't confuse numerals in a string with actual numbers. For instance, try this bit of code out in IDLE's interactive window:

```
>>> num = "2"
>>> num + num
'22'
```

The `+` operator concatenates two string together. So, the result of `"2" + "2"` is `"22"`, not `"4"`.

Strings can be "multiplied" by a number as long as that number is an integer, or whole number. Type the following into the interactive window:

```
>>> num = "12"
>>> num * 3
```

```
'121212'
```

`num * 3` concatenates the string `"12"` with itself three times and returns the string `"121212"`. To compare this operation to arithmetic with numbers, notice that `"12" * 3 = "12" + "12" + "12"`. In other words, multiplying a string by an integer `n` concatenates that string with itself `n` times.

The number on the right-hand side of the expression `num * 3` can be moved to the left, and the result is unchanged:

```
>>> 3 * num
'121212'
```

What do you think happens if you use the `*` operator between two strings? Type `"12" * "3"` in the interactive window and press `Enter`:

```
>>> "12" * "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Python raises a `TypeError` and tells you that you can't multiply a sequence by a non-integer. When the `*` operator is used with a string on either the left or the right side, it always expects an integer on the other side.

Note

A **sequence** is any Python object that supports accessing elements by index. Strings are sequences. You will learn about other sequence types in Chapter 9.

What do you think happens when you try to add a string and a number?

```
>>> "3" + 3
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Again, Python throws a `TypeError` because the `+` operator expects both things on either side of it to be of the same type. If any one of the objects on either side of `+` is a string, Python tries to perform string concatenation. Addition will only be performed if both objects are numbers. So, to add `"3" + 3` and get `6`, you must first convert the string `"3"` to a number.

Converting Strings to Numbers

The `TypeError` errors you saw in the previous section highlight a common problem encountered when working with user input: type mismatches when trying to use the input in an operation that requires a number and not a string.

Let's look at an example. Save and run the following script.

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

When you enter a number, such as `2`, you expect the output to be `4`, but in this case, you get `22`. Remember, `input()` always returns a string, so if you input `2`, then `num` is assigned the string `"2"`, not the integer `2`. Therefore, the expression `num * 2` returns the string `"2"` concatenated with itself, which is `"22"`.

To perform arithmetic on numbers that are contained in a string, you must first convert them from a string type to a number type. There are two ways to do this: `int()` and `float()`.

`int()` stands for **integer** and converts objects into whole numbers, while `float()` stands for **floating-point number** and converts objects into numbers with decimal points. Here's what using them looks like in the interactive window:

```
>>> int("12")
12

>>> float("12")
12.0
```

Notice how `float()` adds a decimal point to the number. Floating-point numbers always have at least one decimal place of precision. For this reason, you can't change a string that looks like a floating-point number into an integer because you would lose everything after the decimal point:

```
>>> int("12.0")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.0'
```

Even though the extra `0` after the decimal place doesn't add any value to the number, Python won't change `12.0` into `12` because it would result in the loss of precision.

Let's revisit the script from the beginning of this section and see how to fix it. Here's the script again:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

The issue lies in the line `doubled_num = num * 2` because `num` references a string and `2` is an integer. You can fix the problem by wrapping `num` with either `int()` or `float()`. Since the prompts asks the user to input a number, and not specifically an integer, let's convert `num` to a floating-point number:

```
num = input("Enter a number to be doubled: ")
doubled_num = float(num) * 2
print(doubled_num)
```

Now when you run this script and input `2`, you get `4.0` as expected. Try it out!

Converting Numbers to Strings

Sometimes you need to convert a number to a string. You might do this, for example, if you need to build a string from some pre-existing variables that are assigned to numeric values.

As you've already seen, the following produces a `TypeError`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + num_pancakes + " pancakes."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Since `num_pancakes` is a number, Python can't concatenate it with the string `"I'm going to eat"`. To build the string, you need to convert `num_pancakes` to a string using `str()`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + str(num_pancakes) + " pancakes."
'I am going to eat 10 pancakes.'
```

You can also call `str()` on a number literal:

```
>>> "I am going to eat " + str(10) + " pancakes."
'I am going to eat 10 pancakes.'
```

`str()` can even handle arithmetic expressions:

```
>>> total_pancakes = 10
>>> pancakes_eaten = 5
>>> "Only " + str(total_pancakes - pancakes_eaten) + " pancakes left."
'Only 5 pancakes left.'
```

You're not limited to numbers when using `str()`. You can pass it all sorts of objects to get their string representations:

```
>>> str(print)
'<built-in function print>'

>>> str(int)
"<class 'int'>"

>>> str(float)
"<class 'float'>"
```

These examples may not seem very useful, but they illustrate how flexible `str()` is.

In the next section, you'll learn how to format strings neatly to display values in a nice, readable manner. Before you move on, though, check your understanding with the following review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Create a string containing an integer, then convert that string into an actual integer object using `int()`. Test that your new object is a number by multiplying it by another number and displaying the result.
2. Repeat the previous exercise, but use a floating-point number and `float()`.
3. Create a string object and an integer object, then display them side-by-side with a single print statement by using the `str()` function.
4. Write a script that gets two numbers from the user using the `input()` function twice, multiplies the numbers together, and displays the result. If the user enters `2` and `4`, your program should print the following text:

   ```
   The product of 2 and 4 is 8.0.
   ```

Leave feedback on this section »

4.7 Streamline Your Print Statements

Suppose you have a string `name = "Zaphod"` and two integers `heads = 2` and `arms = 3`. You want to display them in the following line: `Zaphod has 2 heads and 3 arms`. This is called **string interpolation**, which is just a fancy way of saying that you want to insert some variables into specific locations in a string.

You've already seen two ways of doing this. The first involves using commas to insert spaces between each part of the string inside of a `print()` function:

```
print(name, "has", str(heads), "heads and", str(arms), "arms")
```

Another way to do this is by concatenating the strings with the `+` operator:

```
print(name + " has " + str(heads) + " heads and " + str(arms) + " arms")
```

Both techniques produce code that can be hard to read. Trying to keep track of what goes inside or outside of the quotes can be tough. Fortunately, there's a third way of combining strings: formatted string literals, more commonly known as f-strings.

The easiest way to understand f-strings is to see them in action. Here's what the above string looks like when written as an f-string:

```
>>> f"{name} has {heads} heads and {arms} arms"
'Zaphod has 2 heads and 3 arms'
```

There are two important things to notice about the above examples:

1. The string literal starts with the letter `f` before the opening quotation mark
2. Variable names surrounded by curly braces (`{` and `}`) are replaced with their corresponding values without using `str()`

You can also insert Python expressions in between the curly braces. The expressions are replaced with their result in the string:

```
>>> n = 3
>>> m = 4
>>> f"{n} times {m} is {n*m}"
'3 times 4 is 12'
```

It is a good idea to keep any expressions used in an f-string as simple as possible. Packing in a bunch of complicated expressions into a string literal can result in code that is difficult to read and difficult to maintain.

f-strings are only available in Python version 3.6 and above. In earlier versions of Python, the `.format()` method can be used to get the same results. Returning to the Zaphod example, you can use `.format()` method to format the string like this:

```
>>> "{} has {} heads and {} arms".format(name, heads, arms)
'Zaphod has 2 heads and 3 arms'
```

f-strings are shorter, and sometimes more readable, than using `.format()`. You will see f-strings used throughout this book.

For an in-depth guide to f-strings and comparisons to other string formatting techniques, check out the Python 3's f-Strings: An Improved String Formatting Syntax (Guide) on realpython.com

Note

There is also another way to print formatted strings: using the `%` operator. You might see this in code that you find elsewhere, and you can read about how it works here if you're curious.

Keep in mind that this style has been phased out entirely in Python 3. Just be aware that it exists and you may see it in legacy Python code bases.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Create a `float` object named `weight` with the value 0.2, and create a string object named `animal` with the value `"newt"`. Then use these objects to print the following string using only string concatenation:
   ```
   0.2 kg is the weight of the newt.
   ```
2. Display the same string by using the `.format()` method and empty `{}` place-holders.
3. Display the same string using an f-string.

Leave feedback on this section »

4.8 Find a String in a String

One of the most useful string methods is `.find()`. As its name implies, you can use this method to find the location of one string in another string—commonly referred to as a **substring**.

To use `.find()`, tack it to the end of a variable or a string literal and pass the string you want to find in between the parentheses:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("surprise")
4
```

The value that `.find()` returns is the index of the first occurrence of the string you pass to it. In this case, `"surprise"` starts at the fifth character of the string `"the surprise is in here somewhere"` which has index `4` because counting starts at `0`.

If `.find()` doesn't find the desired substring, it will return `-1` instead:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("eyjafjallajökull")
-1
```

You can call string methods on a string literal directly, so in this case, you don't need to create a new string:

```
>>> "the surprise is in here somewhere".find("surprise")
4
```

Keep in mind that this matching is done exactly, character by character, and is case-sensitive. For example, if you try to find `"SURPRISE"`, the `.find()` method returns `-1`:

```
>>> "the surprise is in here somewhere".find("SURPRISE")
-1
```

If a substring appears more than once in a string, `.find()` only returns the index of the first appearance, starting from the beginning of the string:

```
>>> "I put a string in your string".find("string")
8
```

There are two instances of the `"string"` in `"I put a string in your string"`. The first starts at index `8`, and the second at index `23`. `.find()` returns `8`, which is the index of the first instance of `"string"`.

The `.find()` method only accepts a string as its input. If you want to find an integer in a string, you need to pass the integer to `.find()` as a string. If you do pass something other than a string to `.find()`, Python raises a `TypeError`:

```
>>> "My number is 555-555-5555".find(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

```
>>> "My number is 555-555-5555".find("5")
13
```

Sometimes you need to find all occurrences of a particular substring and replace them with a different string. Since `.find()` only returns the index of the first occurrence of a substring, you can't easily use it to perform this operation. Fortunately, string objects have a `.replace()` method that replaces each instance of a substring with another string.

Just like `.find()`, you tack `.replace()` on to the end of a variable or string literal. In this case, though, you need to put two strings inside of the parentheses in `.replace()` and separate them with a comma. The first string is the substring to find, and the second string is the string to replace each occurrence of the substring with.

For example, the following code shows how to replace each occurrence of `"the truth"` in the string `"I'm telling you the truth; nothing but the truth"` with the string `"lies"`:

```
>>> my_story = "I'm telling you the truth; nothing but the truth!"
>>> my_story.replace("the truth", "lies")
"I'm telling you lies; nothing but lies!"
```

Since strings are immutable objects, `.replace()` doesn't alter `my_story`. If you immediately type `my_story` into the interactive window after running the above example, you'll see the original string, unaltered:

```
>>> my_story
"I'm telling you the truth; nothing but the truth!"
```

To change the value of `my_story`, you need to reassign to it the new value returned by `.replace()`:

```
>>> my_story = my_story.replace("the truth", "lies")
>>> my_story
"I'm telling you lies; nothing but lies!"
```

`.replace()` can only replace one substring at a time, so if you want to replace multiple substrings in a string you need to use `.replace()` mul-

tiple times:

```
>>> text = "some of the stuff"
>>> new_text = text.replace("some of", "all")
>>> new_text = new_text.replace("stuff", "things")
>>> new_text
'all the things'
```

You'll have some fun with `.replace()` in the challenge in the next section.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. In one line of code, display the result of trying to `.find()` the substring `"a"` in the string `"AAA"`. The result should be `-1`.
2. Replace every occurrence of the character `"s"` with `"x"` in the string `"Somebody said something to Samantha."`.
3. Write and test a script that accepts user input using the `input()` function and displays the result of trying to `.find()` a particular letter in that input.

Leave feedback on this section »

4.9 Challenge: Turn Your User Into a L33t H4x0r

Write a script called `translate.py` that asks the user for some input with the following prompt: `Enter some text:`. Then use the `.replace()` method to convert the text entered by the user into "leetspeak" by making the following changes to lower-case letters:

- The letter `a` becomes `4`
- The letter `b` becomes `8`

- The letter `e` becomes `3`
- The letter `l` becomes `1`
- The letter `o` becomes `0`
- The letter `s` becomes `5`
- The letter `t` becomes `7`

Your program should then display the resulting string as output. Below is a sample run of the program:

```
Enter some text: I like to eat eggs and spam.
I 1ik3 70 347 3gg5 4nd 5p4m.
```

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources.

Leave feedback on this section »

4.10 Summary and Additional Resources

In this chapter, you learned the ins and outs of Python string objects. You learned how to access different characters in a string using subscripts and slices, as well as how to determine the length of a string with `len()`.

Strings come with numerous methods. The `.upper()` and `.lower()` methods convert all characters of a string to upper or lower case, respectively. The `.rstrip()`, `.lstrip()`, and `strip()` methods remove whitespace from strings, and the `.startswith()` and `.endswith()` methods will tell you if a string starts or ends with a given substring.

You also saw how to capture input from a user as a string using the `input()` function, and how to convert that input to a number using `int()` and `float()`. To convert numbers, and other objects, to strings, you use `str()`.