

## Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

```
name.py name = "ada lovelace"  
print(name.title())
```

---

Save this file as *name.py*, and then run it. You should see this output:

---

```
Ada Lovelace
```

---

In this example, the variable *name* refers to the lowercase string "ada lovelace". The method `title()` appears after the variable in the `print()` call. A *method* is an action that Python can perform on a piece of data. The dot (`.`) after *name* in `name.title()` tells Python to make the `title()` method act on the variable *name*. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty.

The `title()` method changes each word to title case, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values *Ada*, *ADA*, and *ada* as the same name, and display all of them as *Ada*.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

---

```
name = "Ada Lovelace"  
print(name.upper())  
print(name.lower())
```

---

This will display the following:

---

```
ADA LOVELACE  
ada lovelace
```

---

The `lower()` method is particularly useful for storing data. Many times you won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

## Using Variables in Strings

In some situations, you'll want to use a variable's value inside a string. For example, you might want two variables to represent a first name and a last name respectively, and then want to combine those values to display someone's full name:

```
full_name.py first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

To insert a variable's value into a string, place the letter `f` immediately before the opening quotation mark ❶. Put braces around the name or names of any variable you want to use inside the string. Python will replace each variable with its value when the string is displayed.

These strings are called *f-strings*. The *f* is for *format*, because Python formats the string by replacing the name of any variable in braces with its value. The output from the previous code is:

```
ada lovelace
```

You can do a lot with f-strings. For example, you can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

The full name is used in a sentence that greets the user ❶, and the `title()` method changes the name to title case. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

You can also use f-strings to compose a message, and then assign the entire message to a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

This code displays the message `Hello, Ada Lovelace!` as well, but by assigning the message to a variable ❶ we make the final `print()` call much simpler ❷.

**NOTE**

*F-strings were first introduced in Python 3.6. If you're using Python 3.5 or earlier, you'll need to use the `format()` method rather than this `f` syntax. To use `format()`, list the variables you want to use in the string inside the parentheses following `format`. Each variable is referred to by a set of braces; the braces will be filled by the values listed in parentheses in the order provided:*

---

```
full_name = "{} {}".format(first_name, last_name)
```

---

### **Adding Whitespace to Strings with Tabs or Newlines**

In programming, *whitespace* refers to any nonprinting character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.

To add a tab to your text, use the character combination `\t` as shown at ❶:

---

```
>>> print("Python")
Python
❶ >>> print("\tPython")
    Python
```

---

To add a newline in a string, use the character combination `\n`:

---

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

---

You can also combine tabs and newlines in a single string. The string `"\n\t"` tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

---

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

---

Newlines and tabs will be very useful in the next two chapters when you start to produce many lines of output from just a few lines of code.

### **Stripping Whitespace**

Extra whitespace can be confusing in your programs. To programmers `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.

It's important to think about whitespace, because often you'll want to compare two strings to determine whether they are the same. For example, one important instance might involve checking people's usernames when they log in to a website. Extra whitespace can be confusing in much simpler situations as well. Fortunately, Python makes it easy to eliminate extraneous whitespace from data that people enter.

Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the `rstrip()` method.

---

```
❶ >>> favorite_language = 'python '  
❷ >>> favorite_language  
'python '  
❸ >>> favorite_language.rstrip()  
'python'  
❹ >>> favorite_language  
'python '
```

---

The value associated with `favorite_language` at ❶ contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value ❷. When the `rstrip()` method acts on the variable `favorite_language` at ❸, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, you can see that the string looks the same as when it was entered, including the extra whitespace ❹.

To remove the whitespace from the string permanently, you have to associate the stripped value with the variable name:

---

```
>>> favorite_language = 'python '  
❶ >>> favorite_language = favorite_language.rstrip()  
>>> favorite_language  
'python'
```

---

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then associate this new value with the original variable, as shown at ❶. Changing a variable's value is done often in programming. This is how a variable's value can be updated as a program is executed or in response to user input.

You can also strip whitespace from the left side of a string using the `lstrip()` method, or from both sides at once using `strip()`:

---

```
❶ >>> favorite_language = ' python '  
❷ >>> favorite_language.rstrip()  
' python '  
❸ >>> favorite_language.lstrip()  
'python '  
❹ >>> favorite_language.strip()  
'python'
```

---