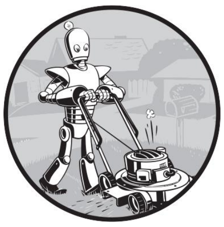
8
STRINGS AND TEXT EDITING



Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values with the + operator, but you can do much more than that, such as extract partial strings from string values, add or remove spacing, convert letters to lowercase or uppercase, and check that strings are formatted correctly. You can even write Python code to access the clipboard used for copying and pasting text.

In this chapter, you'll learn all of this and more. Then, you'll work through a programming project to automate the boring chore of adding bullet points to text.

Working with Strings

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

While string *values* are stored in the program's memory, the string values that literally appear in our code are called *string literals*. Writing string literals in Python code seems straightforward: they begin and end with a single quotation mark, with the text of the string value in between. But how can you use quotes inside a string? Entering `'That is Alice's cat.'` won't work, because Python will think the string ends after `Alice` and will treat the rest (`s cat.'`) as invalid Python code. Fortunately, there are multiple ways to write string literals. The term *string* refers to a string value in the context of a running program and to a string literal when we're talking about entering Python source code.

Double Quotes

String literals can begin and end with double quotes as well as with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Because the string begins with a double quote, Python knows that the single quote is part of the string and is not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string literal. An escape character consists of a backslash (\) followed by the character you want to add to the string. For example, `\'` is the escape character for a single quote and `\n` is the escape character for a newline character. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) You can use this syntax inside a string that begins and ends with single quotes. To see how escape characters work, enter the following into the interactive shell:

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the string value. The escape characters `\'` and `\"` let you put single quotes and double quotes inside your strings, respectively.

Table 8-1 lists the escape characters you can use.

Table 8-1: Escape Characters

Escape character	Prints as ...
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline (line break)
`\\`	Backslash

To practice using these, enter the following into the interactive shell:

```
>>> print("Hello there!\nHow are you?\nI\'m
doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Keep in mind that because the `\` backslash begins an escape character, if you want an actual backslash in your string, you must use the `\\` escape character.

Raw Strings

You can place an `r` before the beginning quotation mark of a string literal to make it a raw string literal. A *raw string* makes it easier to enter string values that have backslashes by ignoring all escape characters. For example, enter the following into the interactive shell:

```
>>> print(r'The file is in C:
\Users\Alice\Desktop')
The file is in C:\Users\Alice\Desktop
```

Because this is a raw string, Python considers the backslash to be part of the string and not the start of an escape character:

```
>>> print('Hello...
\n\n...world!')  # Without a raw string
Hello...

...world!
>>> print(r'Hello...
\n\n...world!')  # With a raw string
Hello...\n\n...world!
```

Raw strings are helpful if your string values contain many backslashes, such as the strings used for Windows filepaths like `r'C:\Users\Al\Desktop'` or regular expression strings, which are described in the next chapter.

Multiline Strings

While you can use the `\n` escape character to insert a newline into a string, it's often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks don't apply to lines inside a multiline string.

For example, open the file editor and enter the following:

```
print('''Dear Alice,

Can you feed Eve's cat this weekend?

Sincerely,
Bob''')
```

Save this program as *feedcat.py* and run it. The output will look like this:

```
Dear Alice,

Can you feed Eve's cat this weekend?

Sincerely,
Bob
```

Notice that the single quote character in `Eve's` doesn't need to be escaped. Escaping single and double quotes is optional in multiline strings:

```
print("Dear Alice,\n\nCan you feed Eve's cat
this weekend?\n\nSincerely,\nBob")
```

This `print()` call prints identical text but doesn't use a multiline string.

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines:

```
"""This is a test Python program.
Written by Al Sweigart
al@inventwithpython.com

This program was designed for Python 3, not
Python 2.
"""

def say_hello():
    """This function prints hello.
    It does not return anything."""
    print('Hello!')
```

The multiline string in this example is perfectly valid Python code.

Indexes and Slices

Strings use indexes and slices the same way lists do. You can think of the string `'Hello, world!'` as a list and each character in the string as an item with a corresponding index and negative index:

```
'   H   e    l    l    o   ,      w   o   r   l
d   !   '
    0   1    2    3    4   5   6   7   8   9 10   11  12
  -13 -12  -11  -10   -9  -8  -7  -6  -5  -4  -3  -2   -1
```

The space and exclamation mark are included in the character count, so `'Hello, world!'` is 13 characters long, from `H` at index 0 to `!` at index 12.

Enter the following into the interactive shell:

```
>>> greeting = 'Hello, world!'
>>> greeting[0]
'H'
>>> greeting [4]
'o'
>>> greeting[-1]
'!'
>>> greeting[0:5]
'Hello'
>>> greeting[:5]
'Hello'
>>> greeting[7:-1]
'world'
>>> greeting[7:]
'world!'
```

If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not. That's why, if `greeting` is `'Hello, world!'`, then `greeting[0:5]` evaluates to `'Hello'`. The substring you get from `greeting[0:5]` will include everything from `greeting[0]` to `greeting[4]`, leaving out the comma at index `5` and the space at index `6`. This is similar to how `range(5)` will cause a `for` loop to iterate up to, but not including, `5`.

Note that slicing a string doesn't modify the original string. You can capture a slice from one variable in a separate variable. Try entering the following into the interactive shell:

```
>>> greeting = 'Hello, world!'
>>> greeting_slice = greeting[0:5]
>>> greeting_slice
'Hello'
>>> greeting
'Hello, world!'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

The in and not in Operators

You can use the `in` and `not in` operators with strings just as you can with list values. An expression with two strings joined using `in` or `not in` will evaluate to a Boolean `True` or `False`. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello, World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello, World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

These expressions test whether the first string (including its capitalization) can be found within the second string.

F-Strings

Putting strings inside other strings is a common operation in programming. So far, we've been using the + operator and string concatenation to do this:

```
>>> name = 'Al'
>>> age = 4000
>>>
'Hello, my name is ' + name + '. I am ' +
str(age) + ' years old.'
'Hello, my name is Al. I am 4000 years old.'
```

```
>>> 'In ten years I will be ' + str(age + 10)
'In ten years I will be 4010'
```

However, this requires a lot of tedious typing. A simpler approach is to use *f-strings*, which let you place variable names or entire expressions within a string. Like the `r` prefix in raw strings, f-strings have an `f` prefix before the starting quotation mark. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> f'My name is {name}. I am {age} years
old.'
'My name is Al. I am 4000 years old.'
>>> f'In ten years I will be {age + 10}'
'In ten years I will be 4010'
```

Everything between the curly brackets (`{}`) is interpreted as if it were passed to `str()` and concatenated with the + operator in the middle of the string. If you need to use literal curly bracket characters in an f-string, use two curly brackets:

```
>>> name = 'Zophie'
>>> f'{name}'
'Zophie'
>>> f'{{name}}'  # Double curly brackets are
literal curly brackets.
'{name}'
```

F-strings are a useful feature in Python, but the language only added them in version 3.6. In older Python code, you may run into alternative techniques.

F-String Alternatives: %s and format()

Versions of Python before 3.6 had other ways to put strings inside other strings. The first is *string interpolation*, in which strings included a `%s` format specifier that Python would replace with another string. For example, enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is %s. I am %s years old.' %
(name, age)
'My name is Al. I am 4000 years old.'
>>> 'In ten years I will be %s' % (age + 10)
'In ten years I will be 4010'
```

Python will replace the first `%s` with the first value in the parentheses after the string, the second `%s` with the second string, and so on. This works just as well as f-strings if you need to insert only one or two strings, but f-strings tend to be more readable when you have several strings to insert.

The next way to put strings inside other strings is with the `format()` string method. You can use a pair of curly brackets to mark where to insert the strings, just like with string interpolation. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is {}. I am {} years
old.'.format(name, age)
'My name is Al. I am 4000 years old.'
```

The `format()` method has a few more features than `%s` string interpolation. You can put the index integer (starting at `0`) inside the curly brackets to note which of the arguments to `format()` should be inserted. This is helpful when inserting strings multiple times or out of order:

```
>>> name = 'Al'
>>> age = 4000
>>> '{1} years ago, {0} was born and named
```

```
{0}.'.format(name, age)
'4000 years ago, Al was born and named Al.'
```

Most programmers prefer f-strings over these two alternatives, but you should learn them anyway, as you may run into them in existing code.

Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

Changing the Case

The `upper()` and `lower()` string methods return a new string with all the letters in the original converted to uppercase or lowercase, respectively. Non-letter characters in the string remain unchanged. For example, enter the following into the interactive shell:

```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam
'HELLO, WORLD!'
>>> spam = spam.lower()
>>> spam
'hello, world!'
```

Note that these methods don't change the string itself, but return new string values. If you want to change the original string, you have to call `upper()` or `lower()` on the string and then assign the new string to the variable that stored the original. This is why you must use `spam = spam.upper()` to change the string in `spam` instead of simply writing `spam.upper()`. (This is the same as if a variable `eggs` contains the value `10`. Writing `eggs + 3` doesn't change the value of `eggs`, but `eggs = eggs + 3` does.)

The `upper()` and `lower()` methods are helpful if you need to make a case-insensitive comparison. For example, the strings `'great'` and `'GREat'` aren't equal to each other, but in the following small program, the user can enter `Great`, `GREAT`, or `grEAT`, because the code converts the string to lowercase:

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is
good.')
```

When you run this program, it displays a question, and entering any variation on `great`, such as `GREat`, will give the output `I feel great too`. Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail:

```
How are you?
GREat
I feel great too.
```

The `isupper()` and `islower()` methods will return a Boolean `True` value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns `False`. Enter the following into the interactive shell, and notice what each method call returns:

```
>>> spam = 'Hello, world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
```

```
>>> '12345'.isupper()
False
```

Since the `upper()` and `lower()` string methods themselves return strings, you can call string methods on *those* returned string values as well:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

Expressions that do this will look like a chain of method calls, as shown here.

Checking String Characteristics

Along with `islower()` and `isupper()`, several other string methods have names beginning with the word *is*. These methods return a Boolean value that describes the nature of the string. Here are some common `is`*`X`*`()` string methods:

`isalpha()` Returns `True` if the string consists only of letters and isn't blank

`isalnum()` Returns `True` if the string consists only of letters and numbers (alphanumerics) and isn't blank

`isdecimal()` Returns `True` if the string consists only of numeric characters and isn't blank

`isspace()` Returns `True` if the string consists only of spaces, tabs, and newlines and isn't blank

`istitle()` Returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '    '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
```

The `isX()` string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input. Open a new file editor window and enter this program, saving it as *validateInput.py*:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your
age.')

while True:
    print('Select a new password (letters and
numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters
and numbers.')
```

In the first `while` loop, we ask the user for their age and store their input in `age`. If `age` is a valid (decimal) value, we break out of this first `while` loop and move on to the second, which asks for a password. Otherwise, we inform the user that they need to enter a number and again ask them to enter their age. In the second `while` loop, we ask for a password, store the user's input in `password`, and break out of the loop if the input was alphanumeric. If it wasn't, we're not satisfied, so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

When run, the program's output looks like this:

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers
only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers
only):
secr3t
```

Calling `isdecimal()` and `isalnum()` on variables, we're able to test whether the values stored in those variables are decimal or not and alphanumeric or not. Here, these tests help us reject the input `forty two` but accept `42`, and reject `secr3t!` but accept `secr3t`.

Checking the Start or End of a String

The `startswith()` and `endswith()` methods return `True` if the string value on which they're called begins or ends (respectively) with the string passed to the method; otherwise, they return `False`. Enter the following into the interactive shell:

```
>>> 'Hello, world!'.startswith('Hello')
True
>>> 'Hello, world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello, world!'.startswith('Hello,
world!')
True
>>> 'Hello, world!'.endswith('Hello, world!')
True
```

These methods are useful alternatives to the equals operator (==) if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

Joining and Splitting Strings

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. We call the `join()` method on a string and pass it a list of strings, and it returns the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string on which `join()` is called is inserted between each string of the list argument. For example, when we call `join(['cats', 'rats', 'bats'])` on the `', '` string, it returns the string `'cats, rats, bats'`.

Remember that we call `join()` on a string value and pass it a list value. (It's easy to accidentally call it the other way around.) The `split()` method works the opposite way: we call it on a string value, and it returns a list of strings. Enter the following into the interactive shell:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the method splits the string `'My name is Simon'` wherever it finds whitespace such as the space, tab, or newline characters. These whitespace characters aren't included in the strings in the returned list. You can pass a delimiter string to the `split()` method to specify a different string to split upon. For example, enter the following into the interactive shell:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters. For example, enter the following into the interactive shell:

```
>>> spam = '''Dear Alice,
... There is a milk bottle in the fridge
... that is labeled "Milk Experiment."
...
... Please do not drink it.
... Sincerely,
... Bob'''
...
>>> spam.split('\n')
['Dear Alice,', 'There is a milk bottle in
the fridge',
'that is labeled "Milk Experiment."', '',
'Please do not drink it.',
'Sincerely,', 'Bob']
```

Passing `split()` the argument `'\n'` lets us split the multiline string stored in `spam` along the newlines and return a list in which each item corresponds to one line of the string.

Justifying and Centering Text

The `rjust()` and `ljust()` string methods return a padded version of the string on which they're called, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello, World'.rjust(20)
'        Hello, World'
>>> 'Hello'.ljust(10)
'Hello     '
```

The code `'Hello'.rjust(10)` says that we want to right-justify `'Hello'` in a string of total length `10`. `'Hello'` is five characters, so five spaces will be added to its left, giving us a string of 10 characters with `'Hello'` right-justified.

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text, rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

Now the printed text is centered.

Removing Whitespace

Sometimes you may want to strip off whitespace characters (spaces, tabs, and newlines) from the left side, right side, or both sides of a string. The `strip()` string method will return a new string without any whitespace characters at the beginning or end, while the `lstrip()` and `rstrip()` methods will remove whitespace characters from the left and right ends, respectively. Enter the following into the interactive shell:

```
>>> spam = '    Hello, World    '
>>> spam.strip()
'Hello, World'
>>> spam.lstrip()
'Hello, World    '
>>> spam.rstrip()
'    Hello, World'
```

Optionally, a string argument will specify which characters on the ends to strip. Enter the following into the interactive shell:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Passing `strip()` the argument `'ampS'` will tell it to strip occurrences of `a`, `m`, `p`, and `S` from the ends of the string stored in `spam`. The order of the characters in the string passed to `strip()` doesn't matter: `strip('ampS')` will do the same thing as `strip('mapS')` or `strip('Spam')`.

Numeric Code Points of Characters

Computers store information as *bytes* (strings of binary numbers), which means we need to be able to convert text to numbers. Because of this requirement, every text character has a corresponding numeric value called a *Unicode code point*. For example, the numeric code point is `65` for `'A'`, `52` for `'4'`, and `33` for `'!'`. You can use the `ord()` function to get the code point of a one-character string, and the `chr()` function to get the one-character string of an integer code point. Enter the following into the interactive shell:

```
>>> ord('A')
65
>>> ord('4')
52
>>> ord('!')
33
>>> chr(65)
'A'
```

These functions are useful when you need to order or perform a mathematical operation on characters:

```
>>> ord('B')
66
>>> ord('A') < ord('B')
True
>>> chr(ord('A'))
'A'
>>> chr(ord('A') + 1)
'B'
```

There is more to Unicode and code points than this, but those details are beyond the scope of this book. If you'd like to know more, I recommend watching or reading Ned Batchelder's 2012 PyCon talk, "Pragmatic Unicode, or How Do I Stop the Pain?" at *https://nedbatchelder.com/text/unipain.html*.

When strings are written to a file or sent over the internet, the conversion from text to bytes is called *encoding*. There are several Unicode encoding standards, but the most popular is UTF-8. If you ever need to choose a Unicode encoding, `'utf-8'` is the correct answer 99 percent of the time. Tom Scott has a Computerphile video titled "Characters, Symbols and the Unicode Miracle" at *https://youtu.be/MijmeoH9LT4* that explains UTF-8 in particular.

Copying and Pasting Strings

The `pyperclip` module has `copy()` and `paste()` functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it into an email, a word processor, or some other software.

The `pyperclip` module doesn't come with Python. To install it, follow the directions for installing third-party packages in Appendix A. After installing `pyperclip`, enter the following into the interactive shell:

```
>>> import pyperclip
>>> pyperclip.copy('Hello, world!')
>>> pyperclip.paste()
'Hello, world!'
```

Of course, if something outside your program changes the clipboard contents, the `paste()` function will return that other value. For example, if I copied this sentence to the clipboard and then called `paste()`, it would look like this:

```
>>> pyperclip.paste()
'For example, if I copied this sentence to
the clipboard and then called
paste(), it would look like this:'
```

The clipboard is an excellent way to enter and receive large amounts of text without having to type it when prompted by an `input()` call. For example, say you want a program to turn text into aLtErNaTiNg uppercase and lowercase letters. You can copy the text you want to alternate to the clipboard, and then run this program, which takes this text and puts the alternating-case text on the clipboard. Enter the following code into a file named *alternatingText.py*:

```
import pyperclip

text = pyperclip.paste()  # Get the text off the clipboard.
alt_text = ''  # This string holds the alternating case.
make_uppercase = False
for character in text:
    # Go through each character and add it to alt_text:
    if make_uppercase:
        alt_text += character.upper()
    else:
        alt_text += character.lower()

    # Set make_uppercase to its opposite value:
    make_uppercase = not make_uppercase
pyperclip.copy(alt_text)  # Put the result on the clipboard.
print(alt_text)  # Print the result on the screen too.
```

If you copy some text to the clipboard (for instance, this sentence) and run this program, the output and clipboard contents become this:

```
iF YoU CoPy sOmE TeXt tO ThE ClIpBoArD (fOr iNsTaNcE, tHiS SeNtEnCe) AnD
```

```
 RuN ThIs pRoGrAm, ThE OuTpUt aNd cLiPbOaRd
cOnTeNtS BeCoMe ThIs:
```

The `pyperclip` module's ability to interact with the clipboard gives you a straightforward way to input and output text to and from your programs.

Project 2: Add Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front of it. But say you have a really large list that you want to add bullet points to. You could type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, say I copied the following text (for the Wikipedia article "List of Lists of Lists") to the clipboard:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

Then, if I ran the *bulletPointAdder.py* program, the clipboard would contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

- Paste text from the clipboard.
- Do something to it.
- Copy the new text to the clipboard.

Manipulating the text is a little tricky, but copying and pasting are pretty straightforward: they just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let's write the part of the program that calls these functions. Enter the following, saving the program as *bulletPointAdder.py*:

```
import pyperclip
text = pyperclip.paste()

# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

The `TODO` comment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text

The call to `pyperclip.paste()` returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in `text` would look like this:

```
'Lists of animals\nLists of aquarium
life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

The `\n` newline characters in this string cause it to be displayed with multiple lines when printed or pasted from the clipboard. There are many "lines" in this one string value. You want to add a star to the start of each of these lines.

You could write code that searches for each `\n` newline character in the string and then adds the star just after that. But it would be easier to use the `split()` method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Edit your program so that it looks like the following:

```
import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):  # Loop through
all indexes in the "lines" list.
    lines[i] = '* ' + lines[i] # Add a star
to each string in the "lines" list.

pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in `lines` and then loop through the items in `lines`. For each line, we add a star and a space to the start of the line. Now each string in `lines` begins with a star.

Step 3: Join the Modified Lines

The `lines` list now contains modified lines that start with stars. But `pyperclip.copy()` is expecting a single string value, not a list of string values. To make this single string value, pass `lines` into the `join()` method to get a single string joined from the list's strings:

```
import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):  # Loop through
all indexes in the "lines" list.
```

```
    lines[i] = '* ' + lines[i]  # Add a star
to each string in the "lines" list.
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don't need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.

A Short Program: Pig Latin

Pig latin is a silly made-up language that alters English words. If a word begins with a vowel, the word *yay* is added to the end of it. If a word begins with a consonant or consonant cluster (like *ch* or *gr*), that consonant or consonant cluster is moved to the end of the word and followed by *ay*.

Let's write a pig latin program that will output something like this:

```
Enter the English message to translate into
pig latin:
My name is AL SWEIGART and I am 4,000 years
old.
Ymay amenay isyay ALYAY EIGARTSWAY andyay
Iyay amyay 4,000 yearsyay oldyay.
```

This program works by altering a string using the methods introduced in this chapter. Enter the following source code into the file editor, and save the file as *pigLat.py*:

```
# English to pig latin
print('Enter the English message to translate
into pig latin:')
message = input()
```

```
VOWELS = ('a', 'e', 'i', 'o', 'u', 'y')

pig_latin = [] # A list of the words in pig
latin
for word in message.split():
    # Separate the non-letters at the start
of this word:
    prefix_non_letters = ''
    while len(word) > 0 and not
word[0].isalpha():
        prefix_non_letters += word[0]
        word = word[1:]
    if len(word) == 0:
        pig_latin.append(prefix_non_letters)
        continue

    # Separate the non-letters at the end of
this word:
    suffix_non_letters = ''
    while not word[-1].isalpha():
        suffix_non_letters = word[-1] +
suffix_non_letters
        word = word[:-1]

    # Remember if the word was in uppercase
or title case:
    was_upper = word.isupper()
    was_title = word.istitle()

    word = word.lower() # Make the word
lowercase for translation.

    # Separate the consonants at the start of
```

```
this word:
    prefix_consonants = ''
    while len(word) > 0 and not word[0] in
VOWELS:
        prefix_consonants += word[0]
        word = word[1:]

    # Add the pig latin ending to the word:
    if prefix_consonants != '':
        word += prefix_consonants + 'ay'
    else:
        word += 'yay'

    # Set the word back to uppercase or title
case:
    if was_upper:
        word = word.upper()
    if was_title:
        word = word.title()

    # Add the non-letters back to the start
or end of the word.
    pig_latin.append(prefix_non_letters +
word + suffix_non_letters)

# Join all the words back together into a
single string:
print(' '.join(pig_latin))
```

Let’s look at this code line by line, starting at the top:

```
# English to pig latin
print('Enter the English message to translate
into pig latin:')
```

```
message = input()

VOWELS = ('a', 'e', 'i', 'o', 'u', 'y')
```

First, we ask the user to enter the English text to translate into pig latin. Also, we create a constant that holds every lowercase vowel (and *y*) as a tuple of strings. We'll use this variable later.

Next, we create the `pigLatin` variable to store the words as we translate them into pig latin:

```
pigLatin = [] # A list of the words in pig
latin
for word in message.split():
    # Separate the non-letters at the start
of this word:
    prefixNonLetters = ''
    while len(word) > 0 and not
word[0].isalpha():
        prefixNonLetters += word[0]
        word = word[1:]
    if len(word) == 0:
        pigLatin.append(prefixNonLetters)
        continue
```

We need each word to be its own string, so we call `message.split()` to get a list of the words as separate strings. The string `'My name is AL SWEIGART and I am 4,000 years old.'` would cause `split()` to return `['My', 'name', 'is', 'AL', 'SWEIGART', 'and', 'I', 'am', '4,000', 'years', 'old.']`.

We also need to remove any non-letters from the start and end of each word so that strings like `'old.'` translate to `'oldyay.'` instead of `'old.yay'`. We save these non-letters to a variable named `prefixNonLetters`.

```
    # Separate the non-letters at the end of
this word:
    suffixNonLetters = ''
    while not word[-1].isalpha():
        suffixNonLetters += word[-1] +
suffixNonLetters
        word = word[:-1]
```

A loop that calls `isalpha()` on the first character in the word determines whether we should remove a character from a word and concatenate it to the end of `prefixNonLetters`. If the entire word is made of non-letter characters, like `'4,000'`, we can simply append it to the `pigLatin` list and continue to the next word to translate. We also need to save the non-letters at the end of the `word` string. This code is similar to the previous loop.

Next, we make sure the program remembers if the word was in uppercase or title case so that we can restore it after translating the word to pig latin:

```
    # Remember if the word was in uppercase
or title case:
    wasUpper = word.isupper()
    wasTitle = word.istitle()

    word = word.lower() # Make the word
lowercase for translation.
```

For the rest of the code in the `for` loop, we'll work on a lowercase version of `word`.

To convert a word like *sweigart* to *eigart-sway*, we need to remove all of the consonants from the beginning of `word`:

```
    # Separate the consonants at the start of
this word:
    prefixConsonants = ''
    while len(word) > 0 and not word[0] in
```

```
VOWELS:
        prefixConsonants += word[0]
        word = word[1:]
```

We use a loop similar to the loop that removed the non-letters from the start of `word`, except now we're pulling off consonants and storing them in a variable named `prefixConsonants`.

If there were any consonants at the start of the word, they're now in `prefixConsonants`, and we should concatenate that variable and the string `'ay'` to the end of `word`. Otherwise, we can assume `word` begins with a vowel and we only need to concatenate `'yay'`:

```
    # Add the pig latin ending to the word:
    if prefixConsonants != '':
        word += prefixConsonants + 'ay'
    else:
        word += 'yay'
```

Recall that we set `word` to its lowercase version with `word = word.lower()`. If `word` was originally in uppercase or title case, this code will convert `word` back to its original case:

```
    # Set the word back to uppercase or title
case:
    if wasUpper:
        word = word.upper()
    if wasTitle:
        word = word.title()
```

At the end of the `for` loop, we append the word, along with any non-letter prefix or suffix it originally had, to the `pigLatin` list:

```
    # Add the non-letters back to the start
or end of the word.
    pigLatin.append(prefixNonLetters + word +
suffixNonLetters)

# Join all the words back together into a
single string:
print(' '.join(pigLatin))
```

After this loop finishes, we combine the list of strings into a single string by calling the `join()` method, then pass this single string to `print()` to display our pig latin on the screen.

Summary

Text is a common form of data, and Python comes with many helpful string methods to process the text stored in string values. You'll make use of indexing, slicing, and string methods in almost every Python program you write.

The programs you are writing now don't seem too sophisticated; they don't have graphical user interfaces (GUIs) with images and colorful text. So far, you're displaying text with `print()` and letting the user enter text with `input()`. However, the user can quickly enter large amounts of text through the clipboard. This ability provides a useful avenue for writing programs that manipulate massive amounts of text. These text-based programs might not have flashy windows or graphics, but they can get a lot of useful work done quickly.

Another way to manipulate large amounts of text is reading and writing files directly off the hard drive. You'll learn how to do this with Python in Chapter 10.

That just about covers all the basic concepts of Python programming! You'll continue to learn new concepts throughout the rest of this book, but you now know enough to start writing some useful programs that can automate tasks. If you'd like to see a collection of short, simple Python programs built from the basic concepts you've learned so far, you can read my other book, *The Big Book of Small Python Projects* (No Starch Press, 2021). Try copying the source code for each program by hand, and then make modifications to see how they affect the behavior of the program. Once you understand how the program works, try re-creating the program yourself from scratch. You don't need to re-create the source code exactly; just focus on what the program does rather than how it does it.

You might not think you have enough Python knowledge to do things such as download web pages, update spreadsheets, or send text messages, but that's where Python modules come in! These modules, written by other programmers, provide functions that make it easy for you to do all these things. In the next chapter, you'll learn how to write real programs to do useful automated tasks.

Practice Questions

1. What are escape characters?
2. What do the `\n` and `\t` escape characters represent?
3. How can you put a \ backslash character in a string?
4. The string value `"Howl's Moving Castle"` is a valid string. Why isn't it a problem that the single quote character in the word `Howl's` isn't escaped?
5. If you don't want to put `\n` in your string, how can you write a string with newlines in it?
6. What do the following expressions evaluate to?
   - `'Hello, world!'[1]`
   - `'Hello, world!'[0:5]`
   - `'Hello, world!'[:5]`
   - `'Hello, world!'[3:]`
7. What do the following expressions evaluate to?
   - `'Hello'.upper()`
   - `'Hello'.upper().isupper()`
   - `'Hello'.upper().lower()`
8. What do the following expressions evaluate to?
   - `'Remember, remember, the fifth of November.'.split()`
   - `'-'.join('There can be only one.'.split())`
9. What string methods can you use to right-justify, left-justify, and center a string?
10. How can you trim whitespace characters from the beginning or end of a string?

Practice Program: Table Printer

For practice, write a function named `printTable()` that takes a list of lists of strings and displays it in a well-organized table with each column right- justified. Assume that all the inner lists will contain the same number of strings. For example, the value could look like this:

```
tableData = [['apples', 'oranges',
'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol',
'David'],
             ['dogs', 'cats', 'moose',
'goose']]
```

Your `printTable()` function would print the following:

```
   apples Alice  dogs
  oranges   Bob  cats
 cherries Carol moose
   banana David goose
```

Hint: Your code will first have to find the longest string in each of the inner lists so that the whole column can be wide enough to fit all the strings. You can store the maximum width of each column as a list of integers. The `printTable()` function can begin with `colWidths = [0] * len(tableData)`, which will create a list containing the same number of `0` values as the number of inner lists in `tableData`. That way, `colWidths[0]` can store the width of the longest string in `tableData[0]`, `colWidths[1]` can store the width of the longest string in `tableData[1]`, and so on. You can then find the largest value in the `colWidths` list to find out what integer width to pass to the `rjust()` string method.

PART II

AUTOMATING TASKS