Chapter 6 – Manipulating Strings

Manipulating Strings

YouTube Lesson 19 - Raw Strings and Multiline Strings

Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values together with the + operator, but you can do much more than that. You can 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 for copying and pasting text.

In this chapter, you'll learn all this and more. Then you'll work through two different programming projects: a simple password manager and a program to automate the boring chore of formatting pieces of 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

Typing string values in Python code is fairly straightforward: They begin and end with a single quote. But then how can you use a quote inside a string? Typing `'That is Alice's cat.'` won't work, because Python thinks the string ends after `Alice`, and the rest (`s cat.'`) is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do 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."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and 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. An escape character consists of a backslash (`\`) followed by the character you want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) For example, the escape character for a single quote is `\'`. You can use this 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 6-1 lists the escape characters you can use.

Table 6-1. Escape Characters

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

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.
```

Raw Strings

You can place an `r` before the beginning quotation mark of a string to make it a raw string. A *raw string* completely ignores all escape characters and prints any backslash that appears in the string. For example, type the following into the interactive shell:

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character. Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for regular expressions described in the next chapter.

Multiline Strings with Triple Quotes

While you can use the `\n` escape character to put a newline into a string, it is 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 do not apply to lines inside a multiline string.

Open the file editor and write the following:

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

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

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

Notice that the single quote character in `Eve's` does not need to be escaped. Escaping single and double quotes is optional in raw strings. The following `print()` call would print identical text but doesn't use a multiline string:

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

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. The following is perfectly valid Python code:

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

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

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

Indexing and Slicing Strings

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.

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

The space and exclamation point are included in the character count, so `'Hello world!'` is 12 characters long, from `H` at index 0 to `!` at index 11.

Enter the following into the interactive shell:

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

```
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'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 spam is `'Hello world!'`, `spam[0:5]` is `'Hello'`. The substring you get from `spam[0:5]` will include everything from `spam[0]` to `spam[4]`, leaving out the space at index 5.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

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 with Strings

The `in` and `not in` operators can be used with strings just like 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 (the exact string, case sensitive) can be found within the second string.

Useful String Methods


Lesson 20 - String Methods and the pyperclip Module

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

The upper(), lower(), isupper(), and islower() String Methods

The `upper()` and `lower()` string methods return a new string where all the letters in the original string have been converted to uppercase or lower-case, respectively. Nonletter characters in the string remain unchanged. 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 do not 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 where the original was stored. This is why you must use `spam = spam.upper()` to change the string in `spam` instead of simply `spam.upper()`. (This is just like if a variable `eggs` contains the value `10`. Writing `eggs + 3` does not 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. The strings `'great'` and `'GREat'` are not equal to each other. But in the

following small program, it does not matter whether the user types `Great`, `GREAT`, or `grEAT`, because the string is first converted 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, the question is displayed, and entering a variation on `great`, such as `GREat`, will still 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. Expressions that do this will look like a chain of method calls. Enter the following into the interactive shell:

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

The isX String Methods

Along with `islower()` and `isupper()`, there are several string methods that 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 is not blank.
- `isalnum()` returns `True` if the string consists only of letters and numbers and is not blank.
- `isdecimal()` returns `True` if the string consists only of numeric characters and is not blank.
- `isspace()` returns `True` if the string consists only of spaces, tabs, and new-lines and is not 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
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

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, alphanumeric or not. Here, these tests help us reject the input `forty two` and accept `42`, and reject `secr3t!` and accept `secr3t`.

The startswith() and endswith() String Methods

The `startswith()` and `endswith()` methods return `True` if the string value they are called on 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.

The join() and split() String Methods

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. The `join()` method is called on a string, gets passed a list of strings, and returns a string. The returned string is 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 `join()` calls on is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is 'cats, rats, bats'.

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

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

By default, the string `'My name is Simon'` is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not 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. Enter the following into the interactive shell:

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container 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 Text with rjust(), ljust(), and center()

The `rjust()` and `ljust()` string methods return a padded version of the string they are called on, 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     '
```

`'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'` justified right.

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========'
```

These methods are especially useful when you need to print tabular data that has the correct spacing. Open a new file editor window and enter the following code, saving it as *picnicTable.py*:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
```

```
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

In this program, we define a `printPicnic()` method that will take in a dictionary of information and use `center()`, `ljust()`, and `rjust()` to display that information in a neatly aligned table-like format.

The dictionary that we'll pass to `printPicnic()` is `picnicItems`. In `picnicItems`, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

To do this, we decide how wide we want the left and right columns to be. Along with our dictionary, we'll pass these values to `printPicnic()`.

`printPicnic()` takes in a dictionary, a `leftWidth` for the left column of a table, and a `rightWidth` for the right column. It prints a title, `PICNIC ITEMS`, centered above the table. Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

After defining `printPicnic()`, we define the dictionary `picnicItems` and call `printPicnic()` twice, passing it different widths for the left and right table columns.

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.........     4
apples.............    12
```

```
cups..............    4
cookies.............  8000
```

Using `rjust()`, `ljust()`, and `center()` lets you ensure that strings are neatly aligned, even if you aren't sure how many characters long your strings are.

Removing Whitespace with strip(), rstrip(), and lstrip()

Sometimes you may want to strip off whitespace characters (space, tab, and newline) 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. 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 should be stripped. Enter the following into the interactive shell:

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

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

Copying and Pasting Strings with the pyperclip Module

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 to an email, word processor, or some other software.

Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing the `pyperclip` module, enter the following into the interactive shell:

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

Of course, if something outside of your program changes the clipboard contents, the `paste()` function will return it. 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:'
```

Running Python Scripts Outside of IDLE

So far, you've been running your Python scripts using the interactive shell and file editor in IDLE. However, you won't want to go through the inconvenience of opening IDLE and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, OS X, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using IDLE.)

Project: Password Locker

You probably have accounts on many different websites. It's a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts. It's best to use password manager software on your computer that uses one master password to

unlock the password manager. Then you can copy any account password to the clipboard and paste it into the website's Password field.

The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

The Chapter Projects

This is the first "chapter project" of the book. From here on, each chapter will have projects that demonstrate the concepts covered in the chapter. The projects are written in a style that takes you from a blank file editor window to a full, working program. Just like with the interactive shell examples, don't only read the project sections—follow along on your computer!

Step 1: Program Design and Data Structures

You want to be able to run this program with a command line argument that is the account's name—for instance, *email* or *blog*. That account's password will be copied to the clipboard so that the user can paste it into a Password field. This way, the user can have long, complicated passwords without having to memorize them.

Open a new file editor window and save the program as *pw.py*. You need to start the program with a `#!` (*shebang*) line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each account's name with its password, you can store these as strings in a dictionary. The dictionary will be the data structure that organizes your account and password data. Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable `sys.argv`. (See Appendix B for more information on how to use command line arguments in your programs.) The first item in the `sys.argv` list should always be a string containing the program's filename (`'pw.py'`), and the second item should be the first command line argument. For this program, this argument is the name of the account whose password you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the `sys.argv` list has fewer than two values in it). Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]      # first command line arg is the account name
```

Step 3: Copy the Right Password

Now that the account name is stored as a string in the variable `account`, you need to see whether it exists in the `PASSWORDS` dictionary as a key. If so, you want to copy the key's value to the clipboard using `pyperclip.copy()`. (Since you're using the `pyperclip` module, you need to import it.) Note that you don't actually *need* the `account` variable; you could just use `sys.argv[1]` everywhere `account` is used in this program. But a variable named `account` is much more readable than something cryptic like `sys.argv[1]`.

Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.
```

```
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]   # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

This new code looks in the `PASSWORDS` dictionary for the account name. If the account name is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no account with that name.

That's the complete script. Using the instructions in Appendix B for launching command line programs easily, you now have a fast way to copy your account passwords to the clipboard. You will have to modify the `PASSWORDS` dictionary value in the source whenever you want to update the program with a new password.

Of course, you probably don't want to keep all your passwords in one place where anyone could easily copy them. But you can modify this program and use it to quickly copy regular text to the clipboard. Say you are sending out several emails that have many of the same stock paragraphs in common. You could put each paragraph as a value in the `PASSWORDS` dictionary (you'd probably want to rename the dictionary at this point), and then you would have a way to quickly select and copy one of many standard pieces of text to the clipboard.

On Windows, you can create a batch file to run this program with the WIN-R Run window. (For more about batch files, see Appendix B.) Type the following into the file editor and save the file as *pw.bat* in the *C:\Windows* folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

With this batch file created, running the password-safe program on Windows is just a matter of pressing WIN-R and typing `pw <account name>`.

Project: Adding 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. But say you have a really large list that you want to add bullet points to. You could just 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, if 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
```

and then ran the *bulletPointAdder.py* program, the clipboard would then 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:

1. Paste text from the clipboard

2. Do something to it

3. Copy the new text to the clipboard

That second step is a little tricky, but steps 1 and 3 are pretty straightforward: They just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let's just write the part of the program that covers steps 1 and 3. Enter the following, saving the program as *bulletPointAdder.py*:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

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 and Add the Star

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 it is 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.

Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

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 star to each string in "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. Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

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

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

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 will 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 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 the next chapter.

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. 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. So let's learn how to write real programs to do useful automated tasks.

Practice Questions

Q: 1. What are escape characters?

Q: 2. What do the `\n` and `\t` escape characters represent?

Q: 3. How can you put a `\` backslash character in a string?

Q: 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?

Q: 5. If you don't want to put `\n` in your string, how can you write a string with newlines in it?

Q: 6. What do the following expressions evaluate to?

- `'Hello world!'[1]`
- `'Hello world!'[0:5]`
- `'Hello world!'[:5]`
- `'Hello world!'[3:]`

Q: 7. What do the following expressions evaluate to?

- `'Hello'.upper()`
- `'Hello'.upper().isupper()`

- `'Hello'.upper().lower()`

Q: 8. What do the following expressions evaluate to?

- `'Remember, remember, the fifth of November.'.split()`
- `'-'.join('There can be only one.'.split())`

Q: 9. What string methods can you use to right-justify, left-justify, and center a string?

Q: 10. How can you trim whitespace characters from the beginning or end of a string?

Practice Project

For practice, write a program that does the following.

Table Printer

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.

ToC