Chapter 6

Strings

Strings are a data type in Python for dealing with text. Python has a number of powerful features for manipulating strings.

6.1 Basics

Creating a string A string is created by enclosing text in quotes. You can use either single quotes, `'`, or double quotes, `"`. A triple-quote can be used for multi-line strings. Here are some examples:

```
s = 'Hello'
t = "Hello"
m = """This is a long string that is
spread across two lines."""
```

Input Recall from Chapter 1 that when getting numerical input we use an **`eval`** statement with the **`input`** statement, but when getting text, we do not use **`eval`**. The difference is illustrated below:

```
num = eval(input('Enter a number: '))
string = input('Enter a string: ')
```

Empty string The empty string `''` is the string equivalent of the number 0. It is a string with nothing in it. We have seen it before, in the print statement's optional argument, `sep=''`.

Length To get the length of a string (how many characters it has), use the built-in function **`len`**. For example, **`len`**`('Hello')` is 5.

6.2 Concatenation and repetition

The operators + and * can be used on strings. The + operator combines two strings. This operation is called *concatenation*. The * repeats a string a certain number of times. Here are some examples.

Expression	Result
`'AB'+'cd'`	`'ABcd'`
`'A'+'7'+'B'`	`'A7B'`
`'Hi'*4`	`'HiHiHiHi'`

Example 1 If we want to print a long row of dashes, we can do the following

```
print('-'*75)
```

Example 2 The + operator can be used to build up a string, piece by piece, analogously to the way we built up counts and sums in Sections 5.1 and 5.2. Here is an example that repeatedly asks the user to enter a letter and builds up a string consisting of only the vowels that the user entered.

```
s = ''
for i in range(10):
    t = input('Enter a letter: ')
    if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
        s = s + t
print(s)
```

This technique is very useful.

6.3 The `in` operator

The `in` operator is used to tell if a string contains something. For example:

```
if 'a' in string:
    print('Your string contains the letter a.')
```

You can combine `in` with the `not` operator to tell if a string does not contain something:

```
if ';' not in string:
    print('Your string does not contain any semicolons.')
```

Example In the previous section we had the long if condition

```
if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
```

Using the `in` operator, we can replace that statement with the following:

```
if t in 'aeiou':
```

6.4 Indexing

We will often want to pick out individual characters from a string. Python uses square brackets to do this. The table below gives some examples of indexing the string `s='Python'`.

Statement	Result	Description
`s[0]`	`P`	first character of `s`
`s[1]`	`y`	second character of `s`
`s[-1]`	`n`	last character of `s`
`s[-2]`	`o`	second-to-last character of `s`

- The first character of `s` is `s[0]`, not `s[1]`. Remember that in programming, counting usually starts at 0, not 1.
- Negative indices count backwards from the end of the string.

A common error Suppose `s='Python'` and we try to do `s[12]`. There are only six characters in the string and Python will raise the following error message:

```
IndexError: string index out of range
```

You *will* see this message again. Remember that it happens when you try to read past the end of a string.

6.5 Slices

A *slice* is used to pick out part of a string. It behaves like a combination of indexing and the **`range`** function. Below we have some examples with the string `s='abcdefghij'`.

```
index:     0 1 2 3 4 5 6 7 8 9
letters:   a b c d e f g h i j
```

Code	Result	Description
`s[2:5]`	`cde`	characters at indices 2, 3, 4
`s[ :5]`	`abcde`	first five characters
`s[5: ]`	`fghij`	characters from index 5 to the end
`s[-2:]`	`ij`	last two characters
`s[ : ]`	`abcdefghij`	entire string
`s[1:7:2]`	`bdf`	characters from index 1 to 6, by twos
`s[ : :-1]`	`jihgfedcba`	a negative step reverses the string

- The basic structure is

  string name[*starting location* : *ending location+1*]

  Slices have the same quirk as the `range` function in that they does not include the ending location. For instance, in the example above, `s[2:5]` gives the characters in indices 2, 3, and 4, but not the character in index 5.

- We can leave either the starting or ending locations blank. If we leave the starting location blank, it defaults to the start of the string. So `s[:5]` gives the first five characters of `s`. If we leave the ending location blank, it defaults to the end of the string. So `s[5:]` will give all the characters from index 5 to the end. If we use negative indices, we can get the ending characters of the string. For instance, `s[-2:]` gives the last two characters.

- There is an optional third argument, just like in the `range` statement, that can specify the step. For example, `s[1:7:2]` steps through the string by twos, selecting the characters at indices 1, 3, and 5 (but not 7, because of the aforementioned quirk). The most useful step is -1, which steps backwards through the string, reversing the order of the characters.

6.6 Changing individual characters of a string

Suppose we have a string called `s` and we want to change the character at index 4 of `s` to `'X'`. It is tempting to try `s[4]='X'`, but that unfortunately will not work. Python strings are *immutable*, which means we can't modify any part of them. There is more on why this is in Section 19.1. If we want to change a character of `s`, we have to instead build a new string from `s` and reassign it to `s`. Here is code that will change the character at index 4 to `'X'`:

```
s = s[:4] + 'X' + s[5:]
```

The idea of this is we take all the characters up to index 4, then `X`, and then all of the characters after index 4.

6.7 Looping

Very often we will want to scan through a string one character at a time. A for loop like the one below can be used to do that. It loops through a string called `s`, printing the string, character by character, each on a separate line:

```
for i in range(len(s)):
    print (s[i])
```

In the `range` statement we have `len(s)` that returns how long `s` is. So, if `s` were 5 characters long, this would be like having `range(5)` and the loop variable `i` would run from 0 to 4. This means that `s[i]` will run through the characters of `s`. This way of looping is useful if we need to keep track of our location in the string during the loop.

If we don't need to keep track of our location, then there is a simpler type of loop we can use:

```
for c in s:
    print(c)
```

This loop will step through `s`, character by character, with `c` holding the current character. You can almost read this like an English sentence, "For every character `c` in `s`, print that character."

6.8 String methods

Strings come with a ton of *methods*, functions that return information about the string or return a new string that is a modified version of the original. Here are some of the most useful ones:

Method	Description
`lower()`	returns a string with every letter of the original in lowercase
`upper()`	returns a string with every letter of the original in uppercase
`replace(x,y)`	returns a string with every occurrence of `x` replaced by `y`
`count(x)`	counts the number of occurrences of `x` in the string
`index(x)`	returns the location of the first occurrence of `x`
`isalpha()`	returns **`True`** if every character of the string is a letter

Important note One very important note about `lower`, `upper`, and `replace` is that they do not change the original string. If you want to change a string, `s`, to all lowercase, it is not enough to just use `s.lower()`. You need to do the following:

```
s = s.lower()
```

Short examples Here are some examples of string methods in action:

Statement	Description
`print(s.count(' '))`	prints the number of spaces in the string
`s = s.upper()`	changes the string to all caps
`s = s.replace('Hi','Hello')`	replaces each `'Hi'` in `s` with `'Hello'`
`print(s.index('a'))`	prints location of the first `'a'` in `s`

`isalpha` The `isalpha` method is used to tell if a character is a letter or not. It returns **`True`** if the character is a letter and **`False`** otherwise. When used with an entire string, it will only return **`True`** if every character of the string is a letter. The values **`True`** and **`False`** are called booleans and are covered in Section 10.2. For now, though, just remember that you can use `isalpha` in if conditions. Here is a simple example:

```
s = input('Enter a string')
```

```
if s[0].isalpha():
    print('Your string starts with a letter')

if not s.isalpha():
    print('Your string contains a non-letter.')
```

A note about `index` If you try to find the index of something that is not in a string, Python will raise an error. For instance, if `s='abc'` and you try `s.index('z')`, you will get an error. One way around this is to check first, like below:

```
if 'z' in s:
    location = s.index('z')
```

Other string methods There are many more string methods. For instance, there are methods `isdigit` and `isalnum`, which are analogous to `isalpha`. Some other useful methods we will learn about later are `join` and `split`. To see a list of all the string methods, type `dir(str)` into the Python shell. If you do this, you will see a bunch of names that start with ___. You can ignore them. To read Python's documentation for one of the methods, say the `isdigit` method, type `help(str.isdigit)`.

6.9 Escape characters

The backslash, `\`, is used to get certain special characters, called escape characters, into your string. There are a variety of escape characters, and here are the most useful ones:

- `\n` the *newline character*. It is used to advance to the next line. Here is an example:

  ```
  print('Hi\n\nthere!')
  ```

  ```
  Hi

  There!
  ```

- `\'` for inserting apostrophes into strings. Say you have the following string:

  ```
  s = 'I can't go'
  ```

  This will produce an error because the apostrophe will actually end the string. You can use `\'` to get around this:

  ```
  s = 'I can\'t go'
  ```

  Another option is to use double quotes for the string:

  ```
  "s = I can't go"
  ```

- `\"` analogous to `\'`.

- `\\` This is used to get the backslash itself. For example:

```
filename = 'c:\\programs\\file.py'
```

- `\t` the tab character

6.10 Examples

Example 1 An easy way to print a blank line is `print()`. However, if we want to print ten blank lines, a quick way to do that is the following:

```
print('\n'*9)
```

Note that we get one of the ten lines from the `print` function itself.

Example 2 Write a program that asks the user for a string and prints out the location of each `'a'` in the string.

```
s = input('Enter some text: ')
for i in range(len(s)):
    if s[i]=='a':
        print(i)
```

We use a loop to scan through the string one character at a time. The loop variable `i` keeps track of our location in the string, and `s[i]` gives the character at that location. Thus, the third line checks each character to see if it is an `'a'`, and if so, it will print out `i`, the location of that `'a'`.

Example 3 Write a program that asks the user for a string and creates a new string that doubles each character of the original string. For instance, if the user enters `Hello`, the output should be `HHeellllloo`.

```
s = input('Enter some text: ')
doubled_s = ''
for c in s:
    doubled_s = doubled_s + c*2
```

Here we can use the second type of loop from Section 6.7. The variable `c` will run through the characters of `s`. We use the repetition operator, `*`, to double each character. We build up the string `s` in the way described at the end of Section 6.2.

Example 4 Write a program that asks a user for their name and prints it in the following funny pattern:

```
E El Elv Elvi Elvis
```

We will require a loop because we have to repeatedly print sections of the string, and to print the sections of the string, we will use a slice:

```
name = input('Enter your name: ')
for i in range(len(name)):
    print(name[:i+1], end=' ')
```

The one trick is to use the loop variable `i` in the slice. Since the number of characters we need to print is changing, we need a variable amount in the slice. This is reminiscent of the triangle program from Section 2.4. We want to print one character of the name the first time through the loop, two characters the second time, etc. The loop variable, `i`, starts at 0 the first time through the loop, then increases to 1 the second time through the loop, etc. Thus we use `name[:i+1]` to print the first `i+1` characters of the name. Finally, to get all the slices to print on the same line, we use the **`print`** function's optional argument `end=''`.

Example 5 Write a program that removes all capitalization and common punctuation from a string `s`.

```
s = s.lower()
for c in ',.;:-?!()\'"':
    s = s.replace(c, '')
```

The way this works is for every character in the string of punctuation, we replace every occurrence of it in `s` with the empty string, `''`. One technical note here: We need the `'` character in a string. As described in the previous section, we get it into the string by using the escape character `\'`.

Example 6 Write a program that, given a string that contains a decimal number, prints out the decimal part of the number. For instance, if given `3.14159`, the program should print out `.14159`.

```
s = input('Enter your decimal number: ')
print(s[s.index('.')+1:])
```

The key here is the `index` method will find where the decimal point is. The decimal part of the number starts immediately after that and runs to the end of the string, so we use a slice that starts at `s.index('.')+1`.

Here is another, more mathematical way, to do this:

```
from math import floor
num = eval(input('Enter your decimal number: ')
print(num - floor(num))
```

One difference between the two methods is the first produces a string, whereas the second produces a number.

Example 7 A simple and very old method of sending secret messages is the substitution cipher. Basically, each letter of the alphabet gets replaced by another letter of the alphabet, say every `a` gets replaced with an `x`, and every `b` gets replaced by a `z`, etc. Write a program to implement this.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
key = 'xznlwebgjhqdyvtkfuompciasr'

secret_message = input('Enter your message: ')
secret_message = secret_message.lower()

for c in secret_message:
    if c.isalpha():
        print(key[alphabet.index(c)],end='')
    else:
        print(c, end='')
```

The string `key` is a random reordering of the alphabet.

The only tricky part of the program is the for loop. What it does is go through the message one character at a time, and, for every letter it finds, it replaces it with the corresponding letter from the key. This is accomplished by using the `index` method to find the position in the alphabet of the current letter and replacing that letter with the letter from the key at that position. All non-letter characters are copied as is. The program uses the `isalpha` method to tell whether the current character is a letter or not.

The code to decipher a message is nearly the same. Just change `key[alphabet.index(c)]` to `alphabet[key.index(c)]`. Section 19.11 provides a different approach to the substitution cipher.

6.11 Exercises

1. Write a program that asks the user to enter a string. The program should then print the following:
   (a) The total number of characters in the string
   (b) The string repeated 10 times
   (c) The first character of the string (remember that string indices start at 0)
   (d) The first three characters of the string
   (e) The last three characters of the string
   (f) The string backwards
   (g) The seventh character of the string if the string is long enough and a message otherwise
   (h) The string with its first and last characters removed
   (i) The string in all caps
   (j) The string with every *a* replaced with an *e*