Strings, Lists, and Dictionaries

4. Strings, Lists, and Dictionaries

This chapter could have had "and Functions" added to its title, but it was already long enough. In this chapter, you will first explore and play with the various ways of representing data and adding some structure to your programs in Python. You will then put everything you learned together into the simple game of Hangman, where you have to guess a word chosen at random by asking whether that word contains a particular letter.

The chapter ends with a reference section that tells you all you need to know about the most useful built-in functions for math, strings, lists, and dictionaries.

4.1. String Theory

No, this is not the Physics kind of String Theory. In programming, a *string* is a sequence of characters you use in your program. In Python, to make a variable that contains a string, you can just use the regular `=` operator to make the assignment, but rather than assigning the variable a number value, you assign it a string value by enclosing that value in single quotes, like this:

```
>>> book_name = 'Programming Raspberry Pi'
```

If you want to see the contents of a variable, you can do so either by entering just the variable name into the Python Shell or by using the `print` command, just as we did with variables that contain a number:

```
>>> book_name

'Programming Raspberry Pi'

>>> print(book_name)

Programming Raspberry Pi
```

There is a subtle difference between the results of each of these methods. If you just enter the variable name, Python puts single quotes around it so that you can tell it is a string. On the other hand, when you use `print`, Python just prints the value.

NOTE *You can also use double quotes to define a string, but the convention is to use single quotes unless you have a reason for using double quotes (for example, if the string you want to create has an apostrophe in it).*

You can find out how many characters a string has in it by doing this:

```
>>> len(book_name)

24
```

You can find the character at a particular place in the string like so:

```
>>> book_name[1]

'r'
```

Two things to notice here: first, the use of square brackets rather than the parentheses that are used for parameters and, second, that the positions start at 0 and not 1. To find the first letter of the string, you need to do the following:

```
>>> book_name[0]
```

```
'P'
```

If you put a number in that is too big for the length of the string, you will see this:

```
>>> book_name[100]

Traceback (most recent call last):

   File "<stdin>", line 1, in <module>

IndexError: string index out of range

>>>
```

This is an error, and it's Python's way of telling us that we have done something wrong. More specifically, the "string index out of range" part of the message tells us that we have tried to access something that we can't. In this case, that's element 100 of a string that is only 24 characters long.

You can chop lumps out of a big string into a smaller string, like this:

```
>>> book_name[0:11]

'Programming'
```

The first number within the brackets is the starting position for the string we want to chop out, and the second number is not, as you might expect, the position of the last character you want, but rather the last character plus 1.

As an experiment, try and chop out the word *raspberry* from the title. If you do not specify the second number, it will default to the end of the string:

```
>>> book_name[12:]

'Raspberry Pi'
```

Similarly, if you do not specify the first number, it defaults to 0.

Finally, you can also join strings together by using + operator. Here's an example:

```
>>> book_name + ' by Simon Monk'

'Programming Raspberry Pi by Simon Monk'
```


4.2. Lists

Earlier in the book when you were experimenting with numbers, a variable could only hold a single number. Sometimes, however, it is useful for a variable to hold a list of numbers or strings, or a mixture of both—or even a list of lists. Figure 4-1 will help you to visualize what is going on when a variable is a list.

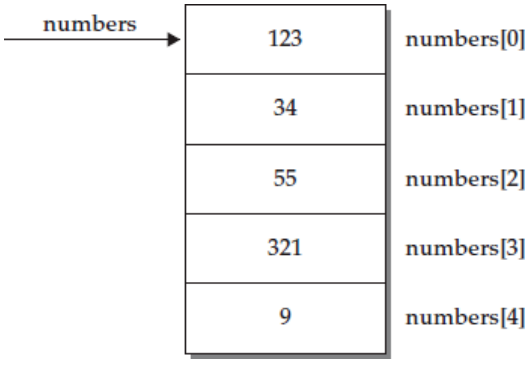

Figure 4-1. An array

Lists behave rather like strings. After all, a string is a list of characters. The following example shows you how to make a list. Notice how `len` works on lists as well as strings:

```
>>> numbers = [123, 34, 55, 321, 9]

>>> len(numbers)

5
```

Square brackets are used to indicate a list, and just like with strings we can use square brackets to find an individual element of a list or to make a shorter list from a bigger one:

```
>>> numbers[0]

123

>>> numbers[1:3]

[34, 55]
```

What's more, you can use = to assign a new value to one of the items in the list, like this:

```
>>> numbers[0] = 1

>>> numbers

[1, 34, 55, 321, 9]
```

This changes the first element of the list (element 0) from 123 to just 1.

As with strings, you can join lists together using the + operator:

```
>>> more_numbers = [5, 66, 44]

>>> numbers + more_numbers

[1, 34, 55, 321, 9, 5, 66, 44]
```

If you want to sort the list, you can do this:

```
>>> numbers.sort()

>>> numbers

[1, 9, 34, 55, 321]
```

To remove an item from a list, you use the command pop, as shown next. If you do not specify an argument to pop, it will just remove the last element of the list and return it.

```
>>> numbers

[1, 9, 34, 55, 321]

>>> numbers.pop()

321

>>> numbers

[1, 9, 34, 55]
```

If you specify a number as the argument to pop, that is the position of the element to be removed. Here's an example:

```
>>> numbers

[1, 9, 34, 55]

>>> numbers.pop(1)
```

```
9

>>> numbers

[1, 34, 55]
```

As well as removing items from a list, you can also insert an item into the list at a particular position. The function `insert` takes two arguments. The first is the position before which to insert, and the second argument is the item to insert.

```
>>> numbers

[1, 34, 55]

>>> numbers.insert(1, 66)

>>> numbers

[1, 66, 34, 55]
```

When you want to find out how long a list is, you use `len(numbers)`, but when you want to sort the list or "pop" an element off the list, you put a dot after the variable containing the list and then issue the command, like this:

```
numbers.sort()
```

These two different styles are a result of something called *object orientation*, which we will discuss in the next chapter.

Lists can be made into quite complex structures that contain other lists and a mixture of different types—numbers, strings, and logical values. Figure 4-2 shows the list structure that results from the following line of code:

```
>>> big_list = [123, 'hello', ['inner list', 2, True]]

>>> big_list

[123, 'hello', ['inner list', 2, True]]
```

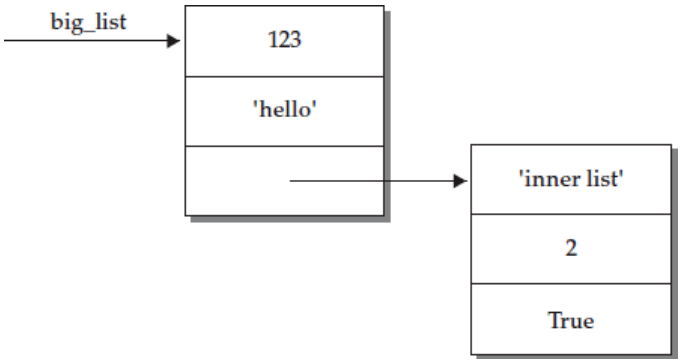

Figure 4-2. A complex list

You can combine what you know about lists with `for` loops and write a short program that creates a list and then prints out each element of the list on a separate line:

```
#4_1_list_and_for

list = [1, 'one', 2, True]

for item in list:

            print(item)
```

Here's the output of this program:

```
1

one
```

```
2

True
```


4.3. Functions

When you are writing small programs like the ones we have been writing so far, they only really perform one function, so there is little need to break them up. It is fairly easy to see what they are trying to achieve. As programs get larger, however, things get more complicated and it becomes necessary to break up your programs into units called *functions*. When we get even further into programming, we will look at better ways still of structuring our programs using classes and modules.

Many of the things I have been referring to as *commands* are actually functions that are built into Python. Examples of this are `range` and `print`.

The biggest problem in software development of any sort is managing complexity. The best programmers write software that is easy to look at and understand and requires very little in the way of extra explanation. Functions are a key tool in creating easy-to-understand programs that can be changed without difficulty or risk of the whole thing falling into a crumpled mess.

A function is a little like a program within a program. We can use it to wrap up a sequence of commands we want to do. A function that we define can be called from anywhere in our program and will contain its own variables and its own list of commands. When the commands have been run, we are returned to just after wherever it was in the code we called the function in the first place.

As an example, let's create a function that simply takes a string as an argument and adds the word *please* to the end of it. Load the following file—or even better, type it in to a new Editor window—and then run it to see what happens:

```
#4_2_polite_function

def make_polite(sentence):

            polite_sentence = sentence + ' please'

            return polite_sentence

print(make_polite('Pass the salt'))
```

The function starts with the keyword `def`. This is followed by the name of the function, which follows the same naming conventions as variables. After that come the parameters inside parentheses and separated by commas if there are more than one. The first line must end with a colon.

Inside the function, we are using a new variable called `polite_sentence` that takes the parameter passed into the function and adds " please" to it (including the leading space). This variable can only be used from inside the function.

The last line of the function is a `return` command. This specifies what value the function should give back to the code that called it. This is just like trigonometric functions such as `sin`, where you pass in an angle and get back a number. In this case, what is returned is the value in the variable `polite_sentence`.

To use the function, we just specify its name and supply it with the appropriate arguments. A return value is not mandatory, and some functions will just do something rather than calculate something. For example, we could write a rather pointless function that prints "Hello" a specified number of times:

```
#4_3_hello_n

def say_hello(n):

            for x in range(0, n):

                        print('Hello')

say_hello(5)
```

This covers the basics of what we will need to do to write our game of Hangman. Although you'll need to learn some other things, we can come

back to these later.

4.4. Hangman

Hangman is a word-guessing game, usually played with pen and paper. One player chooses a word and draws a dash for each letter of the word, and the other player has to guess the word. They guess a letter at a time. If the letter guessed is not in the word, they lose a life and part of the hangman's scaffold is drawn. If the letter is in the word, all occurrences of the letter are shown by replacing the dashes with the letters.

We are going to let Python think of a word and we will have to guess what it is. Rather than draw a scaffold, Python is just going to tell us how many lives we have left.

You are going to start with how to give Python a list of words to chose from. This sounds like a job for a list of strings:

```
words = ['chicken', 'dog', 'cat', 'mouse', 'frog']
```

The next thing the program needs to do is to pick one of those words at random. We can write a function that does that and test it on its own:

```
#4_4_hangman_words

import random

words = ['chicken', 'dog', 'cat', 'mouse', 'frog']

def pick_a_word():

            word_position = random.randint(0, len(words) - 1)

            return words[word_position]

print(pick_a_word())
```

Run this program a few times to check that it is picking different words from the list.

This is a good start, but it needs to fit into the structure of the game. The next thing to do is to define a new variable called `lives_remaining`. This will be an integer that we can start off at 14 and decrease by 1 every time a wrong guess is made. This type of variable is called a *global* variable, because unlike variables defined in functions, we can access it from anywhere in the program.

As well as a new variable, we are also going to write a function called `play` that controls the game. We know what `play` should do, we just don't have all the details yet. Therefore, we can write the function `play` and make up other functions that it will call, such as `get_guess` and `process_guess`, as well as use the function `pick_a_word` we've just written. Here it is:

```
def play():

            word = pick_a_word()

            while True:

                        guess = get_guess(word)

                        if process_guess(guess, word):

                                    print('You win! Well Done!')

                                    break

                        if lives_remaining == 0:

                                    print('You are Hung!')

                                    print('The word was: ' + word)

                                    break
```

A game of Hangman first involves picking a word. There is then a loop that continues until either the word is guessed (`process_guess`

returns `True`) or `lives_remaining` has been reduced to zero. Each time around the loop, we ask the user for another guess.

We cannot run this at the moment because the functions `get_guess` and `process_guess` don't exist yet. However, we can write what are called *stubs* for them that will at least let us try out our `play` function. Stubs are just versions of functions that don't do much; they are stand-ins for when the full versions of the functions are written.

```
def get_guess(word):

            return 'a'

def process_guess(guess, word):

            global lives_remaining

            lives_remaining = lives_remaining - 1

            return False
```

The stub for `get_guess` just simulates the player always guessing the letter *a*, and the stub for `process_guess` always assumes that the player guessed wrong and, thus, decreases `lives_remaining` by 1 and returns `False` to indicate that they didn't win.

The stub for `process_guess` is a bit more complicated. The first line tells Python that the `lives_remaining` variable is the global variable of that name. Without that line, Python assumes that it is a new variable local to the function. The stub then reduces the lives remaining by 1 and returns `False` to indicate that the user has not won yet. Eventually, we will put in checks to see if the player has guessed all the letters or the whole word.

Open the file 4_5_hangman_play.py and run it. You will get a result similar to this:

```
You are Hung!

The word was: dog
```

What happened here is that we have whizzed through all 14 guesses very quickly, and Python has told us what the word was and that we have lost.

All we need to do to complete the program is to replace the stub functions with real functions, starting with `get_guess`, shown here:

```
def get_guess(word):

            print_word_with_blanks(word)

            print('Lives Remaining: ' + str(lives_remaining))

            guess = input(' Guess a letter or whole word?')

            return guess
```

The first thing `get_guess` does is to tell the player the current state of their efforts at guessing (something like "`c--c--n`") using the function `print_word`. This is going to be another stub function for now. The player is then told how many lives they have left. Note that because we want to append a number (`lives_remaining`) after the string `Lives Remaining:`, the number variable must be converted into a string using the built-in `str` function.

The built-in function `input` prints the message in its parameter as a prompt and then returns anything that the user types. Note that in Python 2, the `input` function was called `raw_input`. Therefore, if you decide to use Python 2, change this function to `raw_input`.

Finally, the `get_guess` function returns whatever the user has typed.

The stub function `print_word` just reminds us that we have something else to write later:

```
def print_word_with_blanks(word):

            print('print_word_with_blanks:not done yet')
```

Open the file 4_6_hangman_get_guess.py and run it. You will get a result similar to this:

```
not done yet

Lives Remaining: 14

  Guess a letter or whole word?x

not done yet

Lives Remaining: 13

  Guess a letter or whole word?y

not done yet

Lives Remaining: 12

  Guess a letter or whole word?
```

Enter guesses until all your lives are gone to verify that you get the "losing" message.

Next, we can create the proper version of `print_word`. This function needs to display something like "`c--c--n`," so it needs to know which letters the player has guessed and which they haven't. To do this, it uses a new global variable (this time a string) that contains all the guessed letters. Every time a letter is guessed, it gets added to this string:

```
guessed_letters = ' '
```

Here is the function itself:

```
def print_word_with_blanks(word):

            display_word = ' '

            for letter in word:

                        if guessed_letters.find(letter) > -1:

                               # letter found

                               display_word = display_word + letter

                        else:

                               # letter not found

                               display_word = display_word + '-'

            print display_word
```

This function starts with an empty string and then steps through each letter in the word. If the letter is one of the letters that the player has already guessed, it is added to `display_word`; otherwise, a hyphen (`-`) is added. The built-in function `find` is used to check whether the letter is in the `guessed_letters`. The `find` function returns -1 if the letter is not there; otherwise, it returns the position of the letter. All we really care about is whether or not it is there, so we just check that the result is greater than -1. Finally, the word is printed out.

Currently, every time `process_guess` is called, it doesn't do anything with the guess because it's still a stub. We can make it a bit less of a stub by having it add the guessed letter to `guessed_letters`, like so:

```
def process_guess(guess, word):

            global lives_remaining

            global guessed_letters

            lives_remaining = lives_remaining - 1
```

```
            guessed_letters = guessed_letters + guess

            return False
```

Open the file 4_7_hangman_print_word.py and run it. You will get a result something like this:

```
-------

Lives Remaining: 14

  Guess a letter or whole word?c

c--c---

Lives Remaining: 13

  Guess a letter or whole word?h

ch-c---

Lives Remaining: 12

  Guess a letter or whole word?
```

It's starting to look like the proper game now. However, there is still the stub for `process_guess` to fill out. We will do that next:

```
def process_guess(guess, word):

            if len(guess) > 1:

                        return whole_word_guess(guess, word)

            else:

                        return single_letter_guess(guess, word)
```

When the player enters a guess, they have two choices: They can either enter a single-letter guess or attempt to guess the whole word. In this method, we just decide which type of guess it is and call either `whole_word_guess` or `single_letter_guess`. Because these functions are both pretty straightforward, we will implement them directly rather than as stubs:

```
def single_letter_guess(guess, word):

            global guessed_letters

            global lives_remaining

            if word.find(guess) == -1:

                     # word guess was incorrect

                     lives_remaining = lives_remaining - 1

            guessed_letters = guessed_letters + guess

            if all_letters_guessed(word):

                        return True

def all_letters_guessed(word):

            for letter in word:

                        if guessed_letters.find(letter) == -1:

                                    return False
```

```
            return True
```

The function `whole_word_guess` is actually easier than the `single_letter_guess` function:

```
def whole_word_guess(guess, word):
            global lives_remaining
            if guess.lower() == word.lower():
                        return True
            else:
                        lives_remaining = lives_remaining - 1
                        return False
```

All we have to do is compare the guess and the actual word to see if they are the same when they are both converted to lowercase. If they are not the same, a life is lost. The function returns `True` if the guess was correct; otherwise, it returns `False`.

That's the complete program. Open up 4_8_hangman_full.py in the IDLE Editor and run it. The full listing is shown here for convenience:

```
#04_08_hangman_full
import random
words = ['chicken', 'dog', 'cat', 'mouse', 'frog']
lives_remaining = 14
guessed_letters = ' '
def play():
            word = pick_a_word()
            while True:
                        guess = get_guess(word)
                        if process_guess(guess, word):
                                    print('You win! Well Done!')
                                    break
                        if lives_remaining == 0:
                                    print('You are Hung!')
                                    print('The word was: ' + word)
                                    break
def pick_a_word():
            word_position = random.randint(0, len(words) - 1)
            return words[word_position]
def get_guess(word):
            print_word_with_blanks(word)
            print('Lives Remaining: ' + str(lives_remaining))
```

```
            guess = input(' Guess a letter or whole word?')
            return guess
def print_word_with_blanks(word):
            display_word = ' '
            for letter in word:
                        if guessed_letters.find(letter) > -1:
                                    # letter found
                                    display_word = display_word + letter
                        else:
                                    # letter not found
                                    display_word = display_word + '-'
            print(display_word)
def process_guess(guess, word):
            if len(guess) > 1:
                        return whole_word_guess(guess, word)
            else:
                        return single_letter_guess(guess, word)
def whole_word_guess(guess, word):
            global lives_remaining
            if guess == word:
                        return True
            else:
                        lives_remaining = lives_remaining - 1
                        return False
def single_letter_guess(guess, word):
            global guessed_letters
            global lives_remaining
            if word.find(guess) == -1:
                        # letter guess was incorrect
                        lives_remaining = lives_remaining - 1
            guessed_letters = guessed_letters + guess
            if all_letters_guessed(word):
                        return True
```

```
            return False

def all_letters_guessed(word):

            for letter in word:

                        if guessed_letters.find(letter) == -1:

                                    return False

            return True

play()
```

The game as it stands has a few limitations. First, it is case sensitive, so you have to enter your guesses in lowercase, the same as the words in the `words` array. Second, if you accidentally type **aa** instead of **a** as a guess, it will treat this as a whole-word guess, even though it is too short to be the whole word. The game should probably spot this and only consider guesses the same length as the secret word to be whole-word guesses.

As an exercise, you might like to try and correct these problems. Hint: For the case-sensitivity problem, experiment with the built-in function `lower`. You can look at a corrected version in the file 4_8_hangman_full_solution.py.

4.5. Dictionaries

Lists are great when you want to access your data starting at the beginning and working your way through, but they can be slow and inefficient when they get large and you have a lot of data to trawl through (for example, looking for a particular entry). It's a bit like having a book with no index or table of contents. To find what you want, you have to read through the whole thing.

Dictionaries, as you might guess, provide a more efficient means of accessing a data structure when you want to go straight to an item of interest. When you use a dictionary, you associate a value with a key. Whenever you want that value, you ask for it using the key. It's a little bit like how a variable name has a value associated with it; however, the difference is that with a dictionary, the keys and values are created while the program is running.

Let's look at an example:

```
>>> eggs_per_week = {'Penny': 7, 'Amy': 6, 'Bernadette': 0}

>>> eggs_per_week['Penny']

7

>>> eggs_per_week['Penny'] = 5

>>> eggs_per_week

{'Amy': 6, 'Bernadette': 0, 'Penny': 5}

>>>
```

This example is concerned with recording the number of eggs each of my chickens is currently laying. Associated with each chicken's name is a number of eggs per week. When we want to retrieve the value for one of the hens (let's say Penny), we use that name in square brackets instead of the index number that we would use with a list. We can use the same syntax in assignments to change one of the values.

For example, if Bernadette were to a lay an egg, we could update our records by doing this:

```
eggs_per_week['Bernadette'] = 1
```

You may have noticed that when the dictionary is printed, the items in it are not in the same order as we defined them. The dictionary does not keep track of the order in which items were defined. Also note that although we have used a string as the key and a number as the value, the key could be a string, a number, or a tuple (see the next section), but the value could be anything, including a list or another dictionary.

4.6. Tuples

On the face of it, tuples look just like lists, but without the square brackets. Therefore, we can define and access a tuple like this:

```
>>> tuple = 1, 2, 3

>>> tuple

(1, 2, 3)

>>> tuple[0]

1
```

However, if we try to change an element of a tuple, we get an error message, like this one:

```
>>> tuple[0] = 6

Traceback (most recent call last):

   File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment
```

The reason for this error message is that tuples are *immutable*, meaning that you cannot change them. Strings and numbers are the same. Although you can change a variable to refer to a different string, number, or tuple, you cannot change the number itself. On the other hand, if the variable references a list, you could alter that list by adding, removing, or changing elements in it.

So, if a tuple is just a list that you cannot do much with, you might be wondering why you would want to use one. The answer is, tuples provide a useful way of creating a temporary collection of items. Python lets you do a couple of next tricks using tuples, as described in the next two subsections.

4.6.1. Multiple Assignment

To assign a value to a variable, you just use = operator, like this:

```
a = 1
```

Python also lets you do multiple assignments in a single line, like this:

```
>>> a, b, c = 1, 2, 3

>>> a

1

>>> b

2

>>> c

3
```


4.6.2. Multiple Return Values

Sometimes in a function, you want to return more than one value at a time. As an example, imagine a function that takes a list of numbers and returns the minimum and the maximum. Here is such an example:

```
#04_09_stats

def stats(numbers):

            numbers.sort()

            return (numbers[0], numbers[-1])
```

```
list = [5, 45, 12, 1, 78]

min, max = stats(list)

print(min)

print(max)
```

This method of finding the minimum and maximum is not terribly efficient, but it is a simple example. The list is sorted and then we take the first and last numbers. Note that `numbers[-1]` returns the last number because when you supply a negative index to an array or string, Python counts backward from the end of the list or string. Therefore, the position `-1` indicates the last element, `-2` the second to last, and so on.

4.7. Exceptions

Python uses exceptions to flag that something has gone wrong in your program. Errors can occur in any number of ways while your program is running. A common way we have already discussed is when you try to access an element of a list or string that is outside of the allowed range. Here's an example:

```
>>> list = [1, 2, 3, 4]

>>> list[4]

Traceback (most recent call last):

   File "<stdin>", line 1, in <module>

IndexError: list index out of range
```

If someone gets an error message like this while they are using your program, they will find it confusing to say the least. Therefore, Python provides a mechanism for intercepting such errors and allowing you to handle them in your own way:

```
try:

        list = [1, 2, 3, 4]

           list[4]

except IndexError:

        print('Oops')
```

We cover exceptions again in the next chapter, where you will learn about the hierarchy of the different types of error that can be caught.

4.8. Summary of Functions

This chapter was written to get you up to speed with the most important features of Python as quickly as possible. By necessity, we have glossed over a few things and left a few things out. Therefore, this section provides a reference of some of the key features and functions available for the main types we have discussed. Treat it as a resource you can refer back to as you progress though the book, and be sure to try out some of the functions to see how they work. There is no need to go through everything in this section—just know that it is here when you need it. Remember, the Python Shell is your friend.

For full details of everything in Python, refer to `http://docs.python.org/py3k`.

4.8.1. Numbers

Table 4-1 shows some of the functions you can use with numbers.

Table 4-1. Number Functions

Function	Description	Example

Function	Description	Example
`abs(x)`	Returns the absolute value (removes the - sign).	`>>>abs(-12.3)` `12.3`
`bin(x)`	Used to convert to binary string.	`>>> bin(23)` `'0b10111'`
`complex(r,i)`	Creates a complex number with real and imaginary components. Used in science and engineering.	`>>> complex(2,3)` `(2+3j)`
`hex(x)`	Used to convert to hexadecimal string.	`>>> hex(255)` `'0xff'`
`oct(x)`	Used to convert to octal string.	`>>> oct(9)` `'0o11'`
`round(x, n)`	Round x to n decimal places.	`>>> round(1.111111, 2)` `1.11`
`math.factorial(n)`	Factorial function (as in 4 × 3 × 2 × 1).	`>>> math.factorial` `(4)24`
`math.log(x)`	Natural logarithm.	`>>> math.log(10)` `2.302585092994046`
`math.pow(x, y)`	Raises x to the power of y (alternatively, use x ** y).	`>>> math.pow(2, 8)` `256.0`
`math.sqrt(x)`	Square root.	`>>> math.sqrt(16)` `4.0`

`math.sin, cos, tan, asin, acos, atan`	Trigonometry functions (radians).	`>>> math.sin(math.pi`
		`/ 2)`
		`1.0`

4.8.2. Strings

String constants can be enclosed either with single quotes (most common) or with double quotes. Double quotes are useful if you want to include single quotes in the string, like this:

```
s = "Its 3 o'clock"
```

On some occasions you'll want to include special characters such as end-of-lines and tabs into a string. To do this, you use what are called *escape characters*, which begin with a backslash (\) character. Here are the only ones you are likely to need:

1. **`\t`** Tab character
2. **`\n`** Newline character

Table 4-2 shows some of the functions you can use with strings.

Table 4-2. String Functions

Function	Description	Example
`s.capitalize()`	Capitalizes the first letter and makes the rest lowercase.	`>>> 'aBc'.capitalize()` `'Abc'`

Function	Description	Example
`s.center(width)`	Pads the string with spaces, centering it. An optional extra parameter is used for the fill character.	`>>> 'abc'.center(10, '-')` `'---abc----'`

`s.endswith(str)`	Returns `True` if the end of the string matches.	`>>> 'abcdef'`
		`.endswith('def')`
		`True`

`s.find(str)`	Returns the position of a substring. Optional extra arguments for the start and end positions can be used to limit the search.	`>>> 'abcdef'.find('de')` `3`
`s.format(args)`	Formats a string using template markers using `{}`.	`>>> "Its {0} pm".format('12')` `"Its 12 pm"`
`s.isalnum()`	Returns `True` if all the characters in the string are letters or digits.	`>>> '123abc'.isalnum()` `True`
`s.isalpha()`	Returns `True` if all the characters are alphabetic.	`>>> '123abc'.isalpha()` `False`
`s.isspace()`	Returns `True` if the character is a space, tab, or other whitespace character.	`>>> ' \t'.isspace()` `True`
`s.ljust(width)`	Like `center()`, but left-justified.	`>>> 'abc'.ljust(10, '-')` `'abc-------'`
`s.lower()`	Converts a string into lowercase.	`>>> 'AbCdE'.lower()` `'abcde'`

`s.replace(old, new)`	Replaces all occurrences of `old` with `new`.	`>>> 'hello world'`
		`.replace('world', 'there')`
		`'hello there'`

`s.split()`	Returns a list of all the words in the string, separated by spaces. An optional parameter can be used to indicate a different splitting character. The end of line character (`\n`) is a popular choice.	`>>> 'abc def'.split()` `['abc', 'def']`
`s.splitlines()`	Splits the string on the newline character.	
`s.strip()`	Removes whitespace from both ends of the string.	`>>> '  a b  '.strip()` `'a b'`
`s.upper()`	Refer to `lower()`, earlier in this table.	

4.8.3. Lists

We have already looked at most of the features of lists. Table 4-3 summarizes these features.

Table 4-3. List Functions

Function	Description	Example

`del(a[i:j])`	Deletes elements from the array, from element `i` to element `j-1`.	`>>> a = ['a', 'b', 'c']`
		`>>> del(a[1:2])`
		`>>> a`
		`['a', 'c']`

`a.append(x)`	Appends an element to the end of the list.	`>>> a = ['a', 'b', 'c']`
		`>>> a.append('d')`
		`>>> a`
		`['a', 'b', 'c', 'd']`

`a.count(x)`	Counts the occurrences of a particular element.	`>>> a = ['a', 'b', 'a']`
		`>>> a.count('a')`
		`2`

`a.index(x)`	Returns the index position of the first occurrence of `x` in `a`. Optional parameters can be used for the start and end index.	`>>> a = ['a', 'b', 'c']`
		`>>> a.index('b')`
		`1`

`a.insert(i, x)`	Inserts `x` at position `i` in the list.	`>>> a = ['a', 'c']`
		`>>> a.insert(1, 'b')`
		`>>> a`
		`['a', 'b', 'c']`

`a.pop()`	Returns the last element of the list and removes it. An optional parameter lets you specify another index position for the removal.	`>>> ['a', 'b', 'c']`
		`>>> a.pop(1)`
		`'b'`
		`>>> a`
		`['a', 'c']`

`a.remove(x)`	Removes the element specified.	`>>> a = ['a', 'b', 'c']`
		`>>> a.remove('c')`
		`>>> a`
		`['a', 'b']`

`a.reverse()`	Reverses the list.	`>>> a = ['a', 'b', 'c']`
		`>>> a.reverse()`
		`>>> a`
		`['c', 'b', 'a']`

`a.sort()`	Sorts the list. Advanced options are available when sorting lists of objects. See the next chapter for details.	

4.8.4. Dictionaries

Table 4-4 details a few things about dictionaries that you should know.

Table 4-4. Dictionary Functions

Function	Description	Example

`len(d)`	Returns the number of items in the dictionary.	`>>> d = {'a':1, 'b':2}`
		`>>> len(d)`
		`2`

`del(d[key])`	Deletes an item from the dictionary.	`>>> d = {'a':1, 'b':2}`
		`>>> del(d['a'])`
		`>>> d`
		`{'b': 2}`

`key in d`	Returns `True` if the dictionary (`d`) contains the key.	`>>> d = {'a':1, 'b':2}`
		`>>> 'a' in d`
		`True`

`d.clear()`	Removes all items from the dictionary.	`>>> d = {'a':1, 'b':2}`
		`>>> d.clear()`
		`>>> d`
		`{}`

`get(key, default)`	Returns the value for the key, or default if the key is not there.	`>>> d = {'a':1, 'b':2}`
		`>>> d.get('c', 'c')`
		`'c'`

4.8.5. Type Conversions

We have already discussed the situation where we want to convert a number into a string so that we can append it to another string. Python contains some built-in functions for converting items of one type to another, as detailed in Table 4-5.

Table 4-5. Type Conversions

Function	Description	Example

`float(x)`	Converts x to a floating-point number.	`>>> float('12.34')`
		`12.34`
		`>>> float(12)`
		`12.0`

`int(x)`	Optional argument used to specify the number base.	`>>> int(12.34)`
		`12`
		`>>> int('FF', 16)`
		`255`

`list(x)`	Converts x to a list. This is also a handy way to get a list of dictionaries keys.	`>>> list('abc')`
		`['a', 'b', 'c']`
		`>>> d = {'a':1, 'b':2}`
		`>>> list(d)`
		`['a', 'b']`

4.9. Summary

Many things in Python you will discover gradually. Therefore, do not despair at the thought of learning all these commands. Doing so is really not necessary because you can always search for Python commands or look them up.

In the next chapter, we take the next step and see how Python manages object orientation.