Python Programming
Module 1: Getting Started

Learning objectives

1. Installing Python to your system.
2. Installing a text editor to make it easier to write Python code.
3. Executing snippets of Python code in a terminal session.

1. Setting up your programming environment

Here, we'll look at the two major versions of Python that are currently in use and outline the steps to set up Python on your system.

1.1 Python 2 and Python 3

Two versions of Python are available: Python 2 and the newer Python 3. Most changes are incremental and hardly noticeable, but in some cases code written for Python 2 may not work properly on systems with Python 3 installed. If both versions are installed on your system or if you need to install Python, use Python 3.

2. Python on different operating systems

Python is a cross-platform programming language, which means it runs on all the major operating systems. Any Python program you write should run on any modern computer that has Python installed. However, the methods for setting up Python on different operating systems vary slightly.

2.1 Python on Linux

Linux systems are designed for programming, so Python is already installed on most Linux computers.

2.2 Checking your version of Python

Open a terminal window by running the terminal application on your system (in Ubuntu, you can press CTRL-ALT-T). To find out whether Python is installed, enter `python` with a lowercase `p`. You should see output telling you which version of Python is installed and a >>> prompt where you can start entering Python commands, like this:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

This output tells you that Python 2.7.6 is currently the default version of Python installed on this computer. When you've seen this output, press CTRL-D or enter `exit()` to leave the Python prompt and return to a terminal prompt.

To check for Python 3, you might have to specify that version; so even if the output displayed Python 2.7 as the default version, try the command python3:

```
$ python3
Python 3.5.0 (default, Sep 17 2015, 13:05:18)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

This output means you also have Python 3 installed, so you'll be able to use either version. Whenever you see the python command in this book, enter `python3` instead.

2.3 Installing a text editor

Geany is a simple text editor: it's easy to install, will let you run almost all your programs directly from the editor instead of through a terminal

You can install Geany in one line using the terminal on most Linux systems:

```
$ sudo apt-get install geany
```

If this doesn't work, see the instructions at http://geany.org/Download/ThirdPartyPackages/.

2.4 Running the hello world program

To start your first program, open Geany and save an empty Python file (*File* > *Save As*) called `hello_world.py` in your `python_work` folder. The extension `.py` tells Geany your file will contain a Python program.

After you've saved your file, enter the following line:

```
print("Hello Python world!")
```

If multiple versions of Python are installed on your system, you need to make sure Geany is configured to use the correct version. Go to *Build* > *Set Build Commands*. You should see the words `Compile` and `Execute` with a command next to each. Geany assumes the correct command for each is python, but if your system uses the python3 command, you'll need to change this.

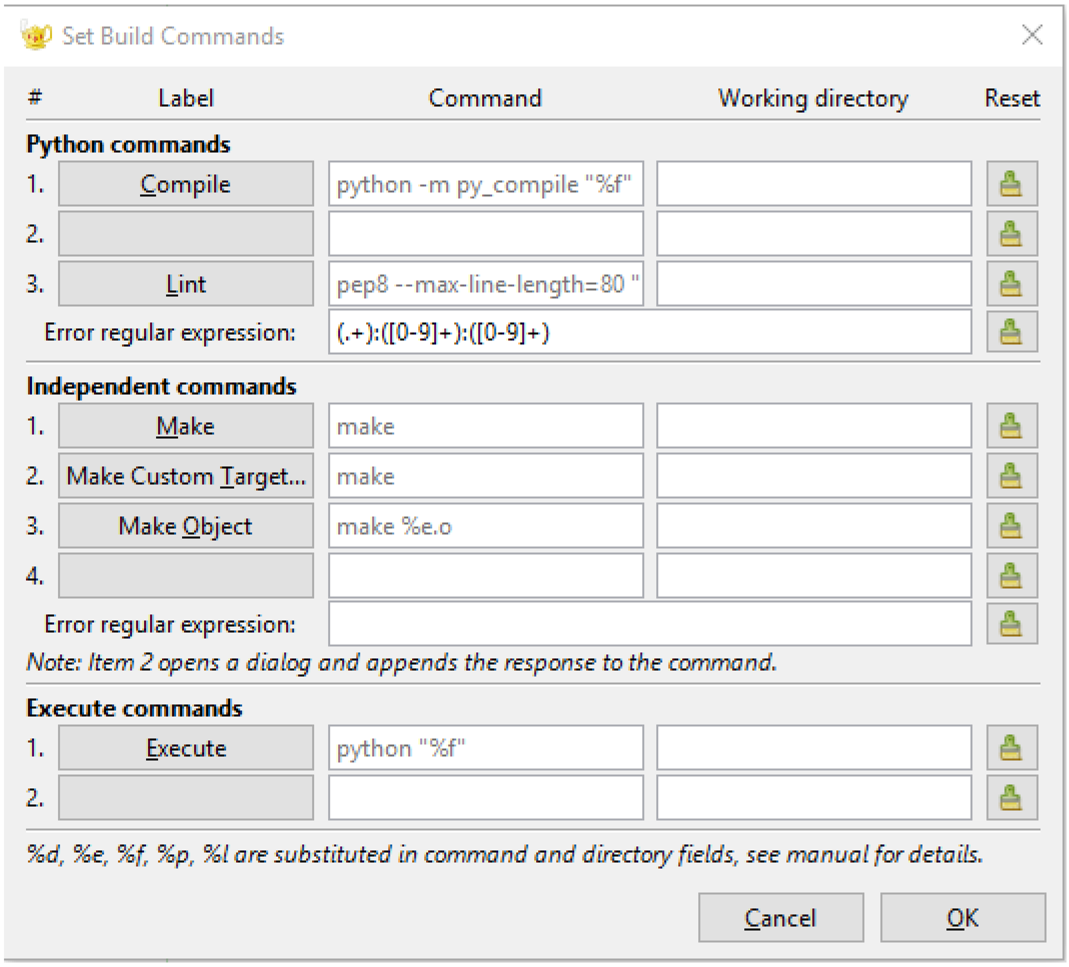
If the command python3 worked in a terminal session, change the Compile and Execute commands so Geany will use the Python 3 interpreter. Your Compile command should look like this:

```
python3 -m py_compile "%f"
```

You need to type this command exactly as it's shown. Make sure the spaces and capitalization match what is shown here. Your Execute command should look like this:

```
python3 "%f"
```

Again, make sure the spacing and capitalization match what is shown here. The figure below shows how these commands should look in Geany's configuration menu.


Set Build Commands

#	Label	Command	Working directory	Reset
Python commands				
1.	Compile	python -m py_compile "%f"		
2.				
3.	Lint	pep8 --max-line-length=80 "		
	Error regular expression:	(.+):([0-9]+):([0-9]+)		
Independent commands				
1.	Make	make		
2.	Make Custom Target...	make		
3.	Make Object	make %e.o		
4.				
	Error regular expression:			

Note: Item 2 opens a dialog and appends the response to the command.

#	Label	Command	Working directory	Reset
Execute commands				
1.	Execute	python "%f"		
2.				

%d, %e, %f, %p, %l are substituted in command and directory fields, see manual for details.

Figure 1-1: Here, Geany is configured to use Python 3 on Linux.

Now run `hello_world.py` by selecting *Build* > *Execute* in the menu, by clicking the Execute icon (which shows a set of gears), or by pressing F5. A terminal window should pop up with the following output:

```
Hello Python world!

------------------
(program exited with code: 0)
Press return to continue.
```

If you don't see this, check every character on the line you entered. Did you accidentally capitalize `print`? Did you forget one or both of the quotation marks or parentheses? Programming languages expect very specific syntax, and if you don't provide that, you'll get errors. If you can't get the program to run, see "Troubleshooting Installation Issues" discussed later in the document.

2.5 Running Python in a terminal session

You can try running snippets of Python code by opening a terminal and typing python or python3, as you did when checking your version. Do this again, but this time enters the following line in the terminal session:

```
>>>print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

You should see your message printed directly in the current terminal window. Remember that you can close the Python interpreter by pressing CTRL-D or by typing the command `exit()`.

3. Python on OS X

Python is already installed on most OS X systems. Once you know Python is installed, you'll need to install a text editor and make sure it's configured correctly.

3.1 Checking whether Python is installed?

Open a terminal window by going to *Applications* > *Utilities* > *Terminal*. You can also press COMMAND-SPACEBAR, then type `terminal`, and then press ENTER. To find out whether Python is installed, enter `python` with a lowercase `p`. You should see output telling you which version of Python is installed on your system and a `>>>` prompt where you can start entering Python commands, like this:

```
$ python
Python 2.7.5 (default, Mar 9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits", or "license" for more
information.
>>>
```

This output tells you that Python 2.7.5 is currently the default version installed on this computer. When you've seen this output, press CTRL-D or enter `exit()` to leave the Python prompt and return to a terminal prompt.

To check for Python 3, try the command `python3`. If `python3` works on your system, whenever you see the python command in this book, make sure you use `python3` instead.

3.2 Running Python in a terminal session

You can try running snippets of Python code by opening a terminal and typing `python` or `python3`. Enter the following line in the terminal session:

```
>>>print("Hello Python interpreter!")
Hello Python interpreter!
```

You can close the Python interpreter by pressing CTRL-D or by typing the command `exit()`.

3.3 Installing a text editor

Sublime Text is a simple text editor: it's easy to install on OS X, will let you run almost all of your programs directly from the editor instead of through a terminal, uses syntax highlighting to color your code, and runs your code in a terminal session embedded in the Sublime Text window to make it easy to see the output.
You can download an installer for Sublime Text from *http://sublimetext .com/3*. Click the download link and look for an installer for OS X. After the installer has been downloaded, open it and then drag the Sublime Text icon into your `Applications` folder.

3.4 Configuring Sublime for Python 3

If you use a command other than python to start a Python terminal session, you'll need to configure Sublime Text so it knows where to find the correct version of Python on your system. Issue the following command to find out the full path to your Python interpreter:

```
$ type -a python3
python3 is /usr/local/bin/python3
```

Now open Sublime Text, and go to *Tools* > *Build System* > *New Build System*, which will open a new configuration file for you. Delete what you see and enter the following:

```
{
     "cmd": ["/usr/local/bin/python3", "-u", "$file"],
}
```

This code tells Sublime Text to use your system's python3 command when running the currently open file. Make sure you use the path you found when issuing the command `type -a python3` in the previous step. Save the file as `Python3.sublime-build` in the default directory that Sublime Text opens when you choose Save.

3.5 Running the hello world program

To start your first program, launch Sublime Text by opening the *Applications* folder and double-clicking the Sublime Text icon. You can also press the command-spacebar and enter `sublime text` in the search bar that pops up.
Make a folder called `python_work` somewhere on your system for your projects. (It's best to use lowercase letters and underscores for spaces in file and folder names because these are Python naming conventions). Save an empty Python file (*File* > *Save As*) called `hello_world.py` in your `python_work` folder. After you've saved your file, enter the following line:

```
print("Hello Python world!")
```

If the command python works on your system, you can run your program by selecting **Tools > Build** on the menu or by pressing COMMAND-B.
A terminal screen should appear at the bottom of the Sublime Text window, showing the following output:

```
Hello Python world!
[Finished in 0.1s]
```

If you can't get the program to run, see "Troubleshooting Installation Issues" discussed later in the document.

4. Python on Windows

Windows don't always come with Python, so you'll probably need to download and install it, and then download and install a text editor.

4.1 Installing Python

Download a Python installer for Windows. Go to [http:// python.org/downloads/](http://python.org/downloads/). You can select one from various available versions to download, which should automatically start downloading the correct installer for your system after the click. After you've downloaded the file, run the installer. Make sure you check the option Add Python to PATH, which will make it easier to configure your system correctly. Figure 1-2 shows this option checked.

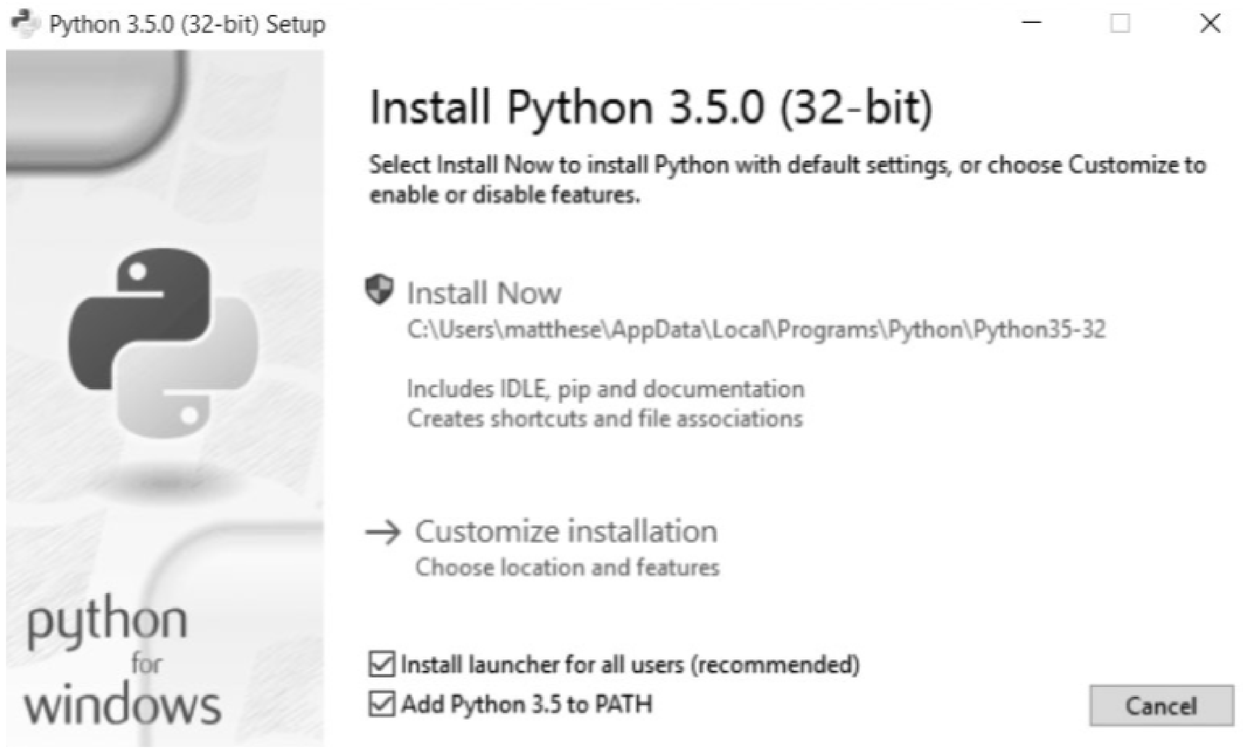

Python 3.5.0 (32-bit) Setup
python for windows
Install Python 3.5.0 (32-bit)
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.
Install Now
C:\Users\matthese\AppData\Local\Programs\Python\Python35-32
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
Install launcher for all users (recommended)
Add Python 3.5 to PATH
Cancel

Figure 1-2: Make sure you check the box labeled Add Python to PATH.

4.2 Starting a Python terminal session

Open a command window and enter `python` in lowercase. If you get a Python prompt (>>>), Windows has found the version of Python you just installed:

```
 C:\>python

Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 22:15:05) [MSC v.1900
32 bit (Intel)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

If this worked, you can move on to the next section, "Running Python in a Terminal Session."
However, you may see output that looks more like this:

```
C:\>python
'python' is not recognized as an internal or external command,
operable program or batch file.
```

In this case, you need to tell Windows how to find the Python version you just installed. Your system's python command is usually saved in your *`C:\`* drive, so open Windows Explorer and open your *`C:\`* drive. Look for a folder starting with the name `Python`, open that folder, and find the `python` file (in lowercase). For example, I have a `Python35` folder with a file named `python` inside it, so the path to the python command on my system is *`C:\Python35\python`*. Otherwise, enter `python` into the search box in Windows Explorer to show you exactly where the python command is stored on your system.
When you think you know the path, test it by entering that path into a terminal window. Open a command window and enter the full path you just found:

```
C:\>C:\Python35\python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 22:15:05) [MSC v.1900
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

If this worked, you know how to access Python on your system.

4.3 Running Python in a terminal Session
Enter the following line in your Python session, and make sure you see the output:

```
Hello Python world!
>>>print("Hello Python world!")
Hello Python world!
>>>
```

Any time you want to run a snippet of Python code, open a command window and start a Python terminal session. To close the terminal session, press CTRL-Z and then press ENTER, or enter the command `exit()`.

4.4 Installing a text editor

Geany is a simple text editor: it's easy to install, will let you run almost all of your programs directly from the editor instead of through a terminal, it uses syntax highlighting to color your code, and runs your code in a terminal window so you'll get used to using terminals.

You can download a Windows installer for Geany from http://geany.org/. Click *Releases* under the Download menu, and look for the ***geany-1.25_setup.exe*** installer or something similar. Run the installer and accept all the defaults.

To start your first program, open Geany. Make a folder called `python_work` somewhere on your system for your projects. (It's best to use lowercase letters and underscores for spaces in file and folder names because these are Python naming conventions.) Go back to Geany and save an empty Python file (*File* > *Save As*) called `hello_world.py` in your `python_work` folder. The extension `.py` tells Geany that your file will contain a Python program. It also tells Geany how to run your program and to highlight the text in a helpful way.

After you've saved your file, type the following line:

```
print("Hello Python world!")
```

If the command python worked on your system, you won't have to configure Geany; skip the next section and move on to "Running the Hello World Program" on page 14. If you needed to enter a path like `C:\Python35\python` to start a Python interpreter, follow the directions in the next section to configure Geany for your system.

4.5 Running the hello world program

Run `hello_world.py` by selecting **Build > Execute** in the menu, by clicking the Execute icon (which shows a set of gears), or by pressing F5. A terminal window should pop up with the following output:

```
Hello Python world!
-------------------
(program exited with code: 0)
Press return to continue
```

If you don't see this, check every character on the line you entered. Did you accidentally capitalize `print`? Did you forget one or both of the quotation marks or parentheses? Programming languages expect very specific syntax, and if you don't provide that, you'll get errors. If you can't get the program to run, see the next section for help.

Till above step you have successfully installed Python to your system if it wasn't already there, and a text editor named Geany. You can also have other text editors of your choice, but at start choosing a simple editor is a good idea and then you can move on to a more complex one as your experience with the language increases.

5. Troubleshooting installation issues

If you've been unable to run `hello_world.py`, here are a few remedies you can try:

1. When a program contains a significant error, Python displays a `traceback`. Python looks through the file and tries to report the problem. The traceback might give you a clue as to what issue is preventing the program from running.
2. Take a step away from your computer, take a short break, and then try again. Remember that syntax is very important in programming, so even a missing colon, a mismatched quotation mark, or mismatched parentheses can prevent a program from running properly.
3. Start over again. You probably don't need to uninstall anything, but it might make sense to delete your `hello_world.py` file and create it again from scratch.

6. Running Python programs from a terminal

Most of the programs you write in your text editor you'll run directly from the editor, but sometimes it's useful to run programs from a terminal instead. For example, you might want to run an existing program without opening it for editing.
You can do this on any system with Python installed if you know how to access the directory where you've stored your program file. To try this, make sure you've saved the `hello_world.py` file in the `python_work` folder on your desktop.

6.1 On Linux and OS X

Running a Python program from a terminal session is the same on Linux and OS X. The terminal command `cd`, for *change directory*, is used to navigate through your file system in a terminal session. The command `ls`, for *list*, shows you all the non-hidden files that exist in the current directory.
Open a new terminal window and issue the following commands to run `hello_world.py`:

```
1 ~$ cd Desktop/python_work/
2 ~/Desktop/python_work$ ls
hello_world.py
3 ~/Desktop/python_work$ python hello_world.py
Hello Python world!
```

At 1, we use the `cd` command to navigate to the `python_work` folder, which is in the `Desktop` folder. Next, we use the `ls` command to make sure `hello_world.py` is in this folder 2. Then, we run the file using the command python hello_world.py 3.
It's that simple. You just use the python (or python3) command to run Python programs.

6.2 On Windows

The terminal command `cd`, for *change directory*, is used to navigate through your file system in a command window. The command `dir`, for a *directory*, shows you all the files that exist in the current directory.
Open a new terminal window and issue the following commands to run `hello_world.py`:

```
1 C:\> cd Desktop\python_work
2 C:\Desktop\python_work> dir
hello_world.py
```

```
3 C:\Desktop\python_work> python hello_world.py
Hello Python world!
```

At 1, we use the cd command to navigate to the `python_work` folder, which is in the *Desktop* folder. Next, we use the dir command to make sure `hello_world.py` is in this folder 2. Then, we run the file using the command `python hello_world.py` 3.
If you haven't configured your system to use the simple command python, you may need to use the longer version of this command:

```
C:\$ cd Desktop\python_work
C:\Desktop\python_work$ dir
hello_world.py
C:\Desktop\python_work$ C:\Python35\python hello_world.py
Hello Python world!
```

Most of your programs will run fine directly from your editor, but as your work becomes more complex, you might write programs that you'll need to run from a terminal.

In a couple of steps above you have executed a snippet of Python code in a terminal session which says Hello python world!.

Python Programming
Module 2: Variables And Simple Data Types

Learning objectives

1. Working with variables.
2. Descriptive variable names and how to resolve name errors and syntax errors when they arise.
3. What strings are and how to display strings using lowercase, uppercase, and title case.
4. Using whitespace to organize output neatly and striping unneeded whitespace from different parts of a string.
5. Start working with integers and floats. Learn about some unexpected behavior to watch out for when working with numerical data.
6. Write explanatory comments to make your code easier for you and others to read.

1. What happens when hello_world.py is executed?

As it turns out, Python does a fair amount of work, even when it runs a simple program:

```
print("Hello Python world!")
```

This line of code produces the following output,

```
Hello Python world!
```

When the file `hello_world.py` executed, the ending `.py` indicates that the file is a Python program. The editor then runs the file through the `Python interpreter`, which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word `print`, it prints to the screen whatever is inside the parentheses.

As you write your programs, your editor highlights different parts of your program in different ways. For example, it recognizes that print is the name of a function and displays that word in blue. It recognizes that `"Hello Python world!"` is not Python code and displays that phrase in orange. This feature is called `syntax highlighting` and is quite useful as you start to write your own programs.

2. Variables

Let's try using a variable in `hello_world.py`. Add a new line at the beginning of the file, and modify the second line:

```
message="Hello Python world!"
print(message)
```

You should see the same output you saw previously:

```
Hello Python world!
```

We've added a `variable` named `message`. Every variable holds a `value`, which is the information associated with that variable. In this case, the value is the text `"Hello Python world!".`
Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the text `"Hello Python world!"` with the variable message. When it reaches the second line, it prints the value associated with a message to the screen.
Let's expand on this program by modifying `hello_world.py` to print a second message. Add a blank line to `hello_world.py`, and then add two new lines of code:

```
message="Hello Python world!"
print(message)

message="Hello Python Crash Course world!"
print(message)
```

Now when you run `hello_world.py`, you should see two lines of output:

```
Hello Python world!
Hello Python Crash Course world!
```

You can change the value of a variable in your program at any time, and Python will always keep track of its current value.

2.1 Naming and using variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable `message_1` but not `1_message`.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, `greeting_message` works, but the `greeting message` will cause errors.
- Do not use words that Python has reserved for a particular programmatic purpose, such as the word print. (See "Python Keywords and Built-in Functions" at the end of the document.)
- Variable names should be short but descriptive. For example, `name` is better than `n`, `student_name` is better than `s_n`, and `name_length` is better than `length_of_persons_name`.
- Be careful when using the lowercase letter l and the upper case letter O because they could be confused with the numbers 1 and 0.

The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but it's a good idea to avoid using them for now.

2.2 Avoiding name errors when using variables

Let's look at an error you're likely to make early on and learn how to fix it. We'll write some code that generates an error on purpose. Enter the following code, including the misspelled word `message` shown in bold:

```
message="Hello Python Crash Course reader!"
print(mesage)
```

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A traceback is a record of where the interpreter ran into trouble when trying to execute your code. Here's an example of the traceback that Python provides after you've accidentally misspelled a variable's name:

```
  Traceback (most recent call last):
1    File "hello_world.py", line 2, in <module>
2          print(mesage)
3 NameError: name 'mesage' is not defined
```

The output at `1`, reports that an error occurs in line 2 of the file `hello_world.py`. The interpreter shows this line to help us spot the error quickly in `2` and `3` tells us what kind of error it found. In this case, it found a `name` error and reports that the variable being printed, `mesage`, has not been defined. Python can't identify the variable name provided. A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name. The best way to understand new programming concepts is to try using them in your programs.

3. Strings

The first data type we'll look at is the string. A string is simply a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favourite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strength is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

3.1 Changing case in a string with methods

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

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

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

```
Ada Lovelace
```

In this example, the lowercase string `"ada lovelace"` is stored in the variable name. The method `title()` appears after the variable in the `print()` statement. A method is an action that Python can perform on a piece of data. The dot `(.)` after name in `name.title()` tells Python to make the `title()` method act on the variable name. Every method is followed by a set of parentheses because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty. `title()` displays each word in title case, where each word begins with a capital letter. You might want your program to recognize the input values `Ada, ADA,` and `ada` as the same name, and display all of them as `Ada`.
Several other useful methods are available for dealing with the case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

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

This will display the following:

```
  ADA LOVELACE
  ada lovelace
```

The `lower()` method is particularly useful for storing data.

3.2 Combining or concatenating strings

It's often useful to combine strings. For example, you might want to store a first name and the last name in separate variables, and then combine them when you want to display someone's full name:

```
  first_name="ada"
  last_name="lovelace"
1 full_name=first_name+" "+last_name

  print(full_name)
```

Python uses the plus symbol (+) to combine strings. In this example, we use + to create a full name by combining a `first_name`, a space, and a `last_name` at `1`, giving this result:

```
  ada lovelace
```

This method of combining strings is called `concatenation`. You can use concatenation to compose complete messages using the information you've stored in a variable. Let's look at an example:

```
  first_name="ada"
  last_name="lovelace"
  full_name=first_name+" "+last_name

1 print("Hello, "+full_name.title()+"!")
```

Here, the full name at `1` is used in a sentence that greets the user, and the `title()` method is used to format the name appropriately. This code returns a simple but nicely formatted greeting:

```
  Hello, Ada Lovelace!
```

You can use concatenation to compose a message and then store the entire message in a variable:

```
  first_name="ada"
  last_name="lovelace"
  full_name=first_name+" "+last_name

1 message="Hello, "+full_name.title()+"!"
```

```
2 print(message)
```

This code displays the message "`Hello, Ada Lovelace!`" as well, but storing the message in a variable at `1` makes the final print statement at `2` much simpler.

3.3 Adding whitespace to strings with tabs or newlines

In programming, whitespace refers to any non-printing character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.
To add a tab to your text, use the character combination `\t` as shown at `1`:

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

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

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

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

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

3.4 Stripping whitespace

To programmers `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.
Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the `rstrip()` method.

```
1 >>> favorite_language='python '
2 >>> favorite_language
  'python '
3 >>> favorite_language.rstrip()
  'python'
```

```
4 >>> favorite_language
  'python '
```

The value stored in `favorite_language` at `1` contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value `2` When the `rstrip()` method acts on the variable `favorite_language` at `3`, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, you can see that the string looks the same as when it was entered, including the extra whitespace `4`. To remove the whitespace from the string permanently, you have to store the stripped value back into the variable:

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

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then store that value back in the original variable, as shown at `1`.
You can also strip whitespace from the left side of a string using the `lstrip()` method or strip whitespace from both sides at once using `strip()`:

```
1 >>> favorite_language=' python '
2 >>> favorite_language.rstrip()
  ' python'
3 >>> favorite_language.lstrip()
  'python '
4 >>> favorite_language.strip()
  'python'
```

In this example, we start with a value that has whitespace at the beginning and the end `1`. We then remove the extra space from the right side at `2`, from the left side at `3`, and from both sides at `4`. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

3.5 Avoiding syntax errors with strings

One kind of error that you might see with some regularity is a syntax error. A `syntax error` occurs when Python doesn't recognize a section of your program as a valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.
Here's how to use single and double quotes correctly. Save this program as `apostrophe.py` and then run it:

```
  message="One of Python's strengths is its diverse community."
```

```
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

```
One of Python's strengths is its diverse community.
```

However, if you use single quotes, Python can't identify where the string should end:

```
message='One of Python's strengths is its diverse community.'
print(message)
```

You'll see the following output:

```
File "apostrophe.py", line 1
     message='One of Python's strengths is its diverse community.'
                                  ^1
SyntaxError: invalid syntax
```

In the output, you can see that the error occurs at `1` right after the second single quote. This `syntax error` indicates that the interpreter doesn't recognize something in the code as a valid Python code. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct.

Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.

4. Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they are being used. Let's first look at how Python manages integers because they are the simplest to work with.

4.1 Integers

You can `add (+), subtract (-), multiply (*), and divide (/)` integers in Python.

```
>>> 2+3
5
>>> 3-2
1
>>> 2*3
6
```

```
>>> 3/2
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3**2
9
>>> 3**3
27
>>> 10**6
1000000
```

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
>>> 2+3*4
14
>>> (2+3)*4
20
```

The spacing in these examples has no effect on how Python evaluates the expressions; it simply helps you more quickly spot the operations that have priority when you're reading through the code.

4.2 Floats

Python calls any number with a decimal point `float`. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number. Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately no matter where the decimal point appears.

For the most part, you can use decimals without worrying about how they behave. Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1+0.1
0.2
>>> 0.2+0.2
0.4
>>> 2*0.1
0.2
>>> 2*0.2
20.4
```

But be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2+0.1
0.30000000000000004
>>> 3*0.1
0.30000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally. Just ignore the extra decimal places for now; you'll learn ways to deal with the extra places when you need to in the projects in Part II.

4.3 Avoiding type errors with the `str()` function

Often, you'll want to use a variable's value within a message. For example, say you want to wish someone a happy birthday. You might write code like this:

```
age=23
message="Happy "+age+"rd Birthday!"

print(message)
```

You might expect this code to print the simple birthday greeting, `Happy 23rd birthday!` But if you run this code, you'll see that it generates an error:

```
  Traceback (most recent call last):
      File "birthday.py", line 2, in <module>
             message="Happy "+age+"rd Birthday!"
1 TypeError: Can't convert 'int' object to str implicitly
```

This is a `type error`. It means Python can't recognize the kind of information you're using. In this example, Python sees at `1` that you're using a variable that has an integer value `(int)`, but it's not sure how to interpret that value. Python knows that the variable could represent either the numerical value 23 or the characters 2 and 3. When you use integers within strings like this, you need to specify explicitly that you want Python to use the integer as a string of characters. You can do this by wrapping the variable in the `str()` function, which tells Python to represent non-string values as strings:

```
age=23
message="Happy "+str(age)+"rd Birthday!"

print(message)
```

Python now knows that you want to convert the numerical value 23 to a string and display the characters 2 and 3 as part of the birthday message. Now you get the message you were expecting, without any errors:

```
Happy 23rd Birthday!
```

Working with numbers in Python is straightforward most of the time. If you're getting unexpected results, check whether Python is interpreting your numbers the way you want it to, either as a numerical value or as a string value.

4.4 Integers in Python 2

Python 2 returns a slightly different result when you divide two integers:

```
>>> python2.7
>>> 3/2
1
```

Instead of `1.5`, Python returns `1`. Division of integers in Python 2 results in an integer with the remainder truncated. Note that the result is not a rounded integer; the remainder is simply omitted.
To avoid this behavior in Python 2, make sure that at least one of the numbers is float. By doing so, the result will be a float as well:

```
>>> 3/2
1
>>> 3.0/2
1.5
>>> 3/2.0
1.5
>>> 3.0/2.0
1.5
```

This division behavior is a common source of confusion when people who are used to Python 3 start using Python 2, or vice versa. If you use or create code that mixes integers and floats, watch out for irregular behavior.

5. Comments

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A `'comment'` allows you to write notes in English within your programs.

5.1 How do you write comments?

In Python, the hash mark (`#`) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter. For example:

```
#Say hello to everyone.
print("Hello Python people!")
```

Python ignores the first line and executes the second line.

```
Hello Python people!
```

5.2 What kind of comments should you write?

The main reason to write comments is to explain what your code is supposed to do and how you are making it work. When you're in the middle of working on a project, you understand how all of the pieces fit together. But when you return to a project after some time away, you'll likely have forgotten some of the details. You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach in clear English.

Python Programming
Module 3: Introducing Lists

Learning objectives

1. What lists are and how to work with individual items in a list.
2. Define a list and how to add and remove elements.
3. Sort lists permanently and temporarily for display purposes.
4. Find the length of a list.
5. How to avoid index errors when you’re working with lists.

LISTS

What is a list?

A list is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0-9, or the names of all the people in your family. A list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names.
In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

```
  bicycles=['trek','cannondale','redline','specialized']
  print(bicycles)
```

If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
  ['trek','cannondale','redline','specialized']
```

Accessing Elements in a List

Lists are ordered collections, so you can access any element in a list by telling Python the position, or index, of the item desired. To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets.
For example, let's pull out the first bicycle in the list bicycles:

```
  bicycles=['trek','cannondale','redline','specialized']
1 print(bicycles[0])
```

The syntax for this is shown at `1`. When we ask for a single item from a list, Python returns just that element without square brackets or quotation marks:

```
  trek
```

This is the result you want your users to see clean, neatly formatted output.
You can also use the string methods on any element in a list. For example, you can format the element `'trek'` more neatly by using the `title()` method:

```
  bicycles=['trek','cannondale','redline','specialized']
  print(bicycles[0].title())
```

This example produces the same output as the preceding example except `'Trek'` is capitalized.

Index Positions Start at 0, Not 1

Python considers the first item in a list to be at position 0, not position 1. This is true of most programming languages, and the reason has to do with how the list operations are implemented at a lower level. If you're receiving unexpected results, determine whether you are making a simple off-by-one error.
The second item in a list has an index of 1. Using this simple counting system, you can get any element you want from a list by subtracting one from its position in the list. For instance, to access the fourth item in a list, you request the item at index-3.
The following asks for the bicycles at index-1 and index-3:

```
  bicycles=['trek','cannondale','redline','specialized']
  print(bicycles[1])
  print(bicycles[3])
```

This code returns the second and fourth bicycles in the list:

```
  cannondale
  specialized
```

Python has a special syntax for accessing the last element in a list. By asking for the item at index-1, Python always returns the last item in the list:

```
  bicycles=['trek','cannondale','redline','specialized']
  print(bicycles[-1])
```

This code returns the value `'specialized'`. This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well. The index-2 returns the second item from the end of the list, the index-3 returns the third item from the end, and so forth.

Using Individual Values from a List

You can use individual values from a list just as you would any other variable. For example, you can use concatenation to create a message based on a value from a list.
Let's try pulling the first bicycle from the list and composing a message using that value.

```
  bicycles=['trek','cannondale','redline','specialized']
1 message="My first bicycle was a "+bicycles[0].title()+"."

  print(message)
```

At `1`, we build a sentence using the value at `bicycles[0]` and store it in the variable message. The output is a simple sentence about the first bicycle in the list:

```
  My first bicycle was a Trek.
```

Changing, adding, and removing elements

Most lists you create will be dynamic, meaning you'll build a list and then add and remove elements from it as your program runs its course. For example, you might create a game in which a player has to shoot aliens out of the sky. You could store the initial set of aliens in a list and then remove an alien from the list each time one is shot down. Each time a new alien appears on the screen, you add it to the list. Your list of aliens will decrease and increase in length throughout the course of the game.

Modifying Elements in a List

To change an element, use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have.
For example, let's say we have a list of motorcycles, and the first item in the list is `'honda'`. How would we change the value of this first item?

```
1 motorcycles=['honda','yamaha','suzuki']
  print(motorcycles)

2 motorcycles[0]='ducati'
  print(motorcycles)
```

The code at `1` defines the original list, with `'honda'` as the first element. The code at `2` changes the value of the first item to `'ducati'`. The output shows that the first item has indeed been changed, and the rest of the list stays the same:

```
['honda','yamaha','suzuki']
['ducati','yamaha','suzuki']
```

You can change the value of any item in a list, not just the first item.

Adding Elements to a List

You might want to add a new element to a list for many reasons. For example, you might want to make new aliens appear in a game, add new data to a visualization, or add new registered users to a website you've built. Python provides several ways to add new data to existing lists.

Appending Elements to the End of a List

When you append an item to a list, the new element is added to the end of the list. Using the same list that we had in the previous example, we'll add the new element `'ducati'` to the end of the list:

```
  motorcycles=['honda','yamaha','suzuki']
  print(motorcycles)

1 motorcycles.append('ducati')
  print(motorcycles)
```

The `append()` method at `1` adds `'ducati'` to the end of the list without affecting any of the other elements in the list:

```
['honda','yamaha','suzuki']
['honda','yamaha','suzuki','ducati']
```

The `append()` method makes it easy to build lists dynamically. For example, you can start with an empty list and then add items to the list using a series of `append()` statements. Using an empty list, let's add the elements `'honda'`, `'yamaha'`, and `'suzuki'` to the list:

```
motorcycles=[]

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

The resulting list looks exactly the same as the lists in the previous examples:

```
['honda','yamaha','suzuki']
```

Inserting Elements into a List

You can add a new element at any position in your list by using the `insert()` method. You do this by specifying the index of the new element and the value of the new item.

```
  motorcycles=['honda','yamaha','suzuki']

1 motorcycles.insert(0, 'ducati')
  print(motorcycles)
```

In this example, the code at `1` inserts the value `'ducati'` at the beginning of the list. The `insert()` method opens a space at position 0 and stores the value `'ducati'` at that location. This operation shifts every other value in the list one position to the right:

```
['ducati','honda','yamaha','suzuki']
```

Removing an Item Using the del Statement

If you know the position of the item you want to remove from a list, you can use the `del` statement.

```
  motorcycles=['honda','yamaha','suzuki']
  print(motorcycles)

1 del motorcycles[0]
  print(motorcycles)
```

The code at `1` uses `del` to remove the first item, `'honda'`, from the list of motorcycles:

```
['honda','yamaha','suzuki']
['yamaha','suzuki']
```

You can remove an item from any position in a list using the `del` statement if you know its index. For example, here's how to remove the second item, `'yamaha'`, in the list:

```
motorcycles=['honda','yamaha','suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

The second motorcycle is deleted from the list:

```
['honda','yamaha','suzuki']
['honda','suzuki']
```

In both examples, you can no longer access the value that was removed from the list after the `del` statement is used.

Removing an Item Using the `pop()` Method

The `pop()` method removes the last item in a list, but it lets you work with that item after removing it. The term `pop` comes from thinking of a list as a stack of items and popping one item off the top of the stack. In this analogy, the top of a stack corresponds to the end of a list.
Let's pop a motorcycle from the list of motorcycles:

```
1 motorcycles=['honda','yamaha','suzuki']
  print(motorcycles)

2 popped_motorcycle=motorcycles.pop()
3 print(motorcycles)
4 print(popped_motorcycle)
```

We start by defining and printing the list motorcycles at `1`. At `2`, we pop a value from the list and store that value in the variable `popped_motorcycle`. We print the list at `3` to show that a value has been removed from the list. Then we print the popped value at `4` to prove that we still have access to the value that was removed.
The output shows that the value `'suzuki'` was removed from the end of the list and is now stored in the variable `popped_motorcycle`:

```
['honda','yamaha','suzuki']
['honda','yamaha']
suzuki
```

How might this `pop()` method be useful? Imagine that the motorcycles in the list are stored in chronological order according to when we owned them. If this is the case, we can use the `pop()` method to print a statement about the last motorcycle we bought:

```
motorcycles=['honda','yamaha','suzuki']

last_owned=motorcycles.pop()
print("The last motorcycle I owned was a "+last_owned.title()+".")
```

The output is a simple sentence about the most recent motorcycle we owned:

```
The last motorcycle I owned was a Suzuki.
```

Popping Items from any Position in a List

You can actually use `pop()` to remove an item in a list at any position by including the index of the item you want to remove in parentheses.

```
motorcycles=['honda','yamaha','suzuki']
```

```
1 first_owned=motorcycles.pop(0)
2 print('The first motorcycle I owned was a'+first_owned.title()+'.')
```

We start by popping the first motorcycle in the list at `1`, and then we print a message about that motorcycle at `2`. The output is a simple sentence describing the first motorcycle I ever owned:

```
The first motorcycle I owned was a Honda.
```

Remember that each time you use `pop()`, the item you work with is no longer stored in the list.
If you're unsure whether to use the del statement or the `pop()` method, here's a simple way to decide: when you want to delete an item from a list and not use that item in any way, use the `del` statement; if you want to use an item as you remove it, use the `pop()` method.

Removing an Item by Value

Sometimes you won't know the position of the value you want to remove from a list. If you only know the value of the item you want to remove, you can use the `remove()` method.
For example, let's say we want to remove the value `'ducati'` from the list of motorcycles.

```
  motorcycles=['honda','yamaha','suzuki','ducati']
  print(motorcycles)

1 motorcycles.remove('ducati')
  print(motorcycles)
```

The code at `1` tells Python to figure out where `'ducati'` appears in the list and remove that element:

```
['honda','yamaha','suzuki','ducati']
['honda','yamaha','suzuki']
```

You can also use the `remove()` method to work with a value that's being removed from a list. Let's remove the value `'ducati'` and print a reason for removing it from the list:

```
1 motorcycles=['honda','yamaha','suzuki','ducati']
  print(motorcycles)

2 too_expensive='ducati'
3 motorcycles.remove(too_expensive)
  print(motorcycles)
4 print("\nA "+too_expensive.title()+" is too expensive for me.")
```

After defining the list at `1`, we store the value `'ducati'` in a variable called `too_expensive` at `2`. We then use this variable to tell Python which value to remove from the list at `3`. At `4`, the value `'ducati'` has been removed from the list but is still stored in the variable `too_expensive`, allowing us to print a statement about why we removed `'ducati'` from the list of motorcycles:

```
['honda','yamaha','suzuki','ducati']
```

```
  ['honda','yamaha','suzuki']

  A Ducati is too expensive for me.
```

The `remove()` *method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to determine if all occurrences of the value have been removed.*

Organizing a list

Sorting a List Permanently with the `sort()` *Method*

Python's `sort()` method makes it relatively easy to sort a list. Imagine we have a list of cars and want to change the order of the list to store them alphabetically. To keep the task simple, let's assume that all the values in the list are lowercase.

```
  cars=['bmw','audi','toyota','subaru']
1 cars.sort()
  print(cars)
```

The `sort()` method, shown at `1`, changes the order of the list permanently. The cars are now in alphabetical order, and we can never revert to the original order:

```
  ['audi','bmw','subaru','toyota']
```

You can also sort this list in reverse alphabetical order by passing the argument reverse = True to the `sort()` method. The following example sorts the list of cars in reverse alphabetical order:

```
  cars=['bmw','audi','toyota','subaru']
  cars.sort(reverse=True)
  print(cars)
```

Again, the order of the list is permanently changed:

```
  ['toyota','subaru','bmw','audi']
```

Sorting a List Temporarily with the `sorted()` *Function*

To maintain the original order of a list but present it in a sorted order, you can use the `sorted()` function. The `sorted()` function lets you display your list in a particular order but doesn't affect the actual order of the list.
Let's try this function on the list of cars.

```
  cars=['bmw','audi','toyota','subaru']

1 print("Here is the original list:")
  print(cars)

2 print("\nHere is the sorted list:")
```

```
  print(sorted(cars))

3 print("\nHere is the original list again:")
  print(cars)
```

We first print the list in its original order at `1` and then in alphabetical order at `2`. After the list is displayed in the new order, we show that the list is still stored in its original order at `3`.

```
  Here is the original list:
  ['bmw','audi','toyota','subaru']

  Here is the sorted list:
  ['audi','bmw','subaru','toyota']

  Here is the original list again:
4 ['bmw','audi','toyota','subaru']
```

Notice that the list still exists in its original order at `4` after the `sorted()` function has been used. The `sorted()` function can also accept a reverse = True argument if you want to display a list in reverse alphabetical order.

Sorting a list alphabetically is a bit more complicated when all the values are not in lowercase. There are several ways to interpret capital letters when you're deciding on a sort order, and specifying the exact order can be more complex than we want to deal with at this time. However, most approaches to sorting will build directly on what you learned in this section.

Printing a List in Reverse Order

To reverse the original order of a list, you can use the `reverse()` method. If we originally stored the list of cars in chronological order according to when we owned them, we could easily rearrange the list into reverse chronological order:

```
cars=['bmw','audi','toyota','subaru']
print(cars)

cars.reverse()
print(cars)
```

Notice that `reverse()` doesn't sort backward alphabetically; it simply reverses the order of the list:

```
['bmw','audi','toyota','subaru']
['subaru','toyota','audi','bmw']
```

The `reverse()` method changes the order of a list permanently, but you can revert to the original order anytime by applying `reverse()` to the same list a second time.

Finding the Length of a List

You can quickly find the length of a list by using the `len()` function. The list in this example has four items, so its length is 4:

```
>>> cars=['bmw','audi','toyota','subaru']
>>> len(cars)
4
```

You'll find `len()` useful when you need to identify the number of aliens that still need to be shot down in a game, determine the amount of data you have to manage in a visualization, or figure out the number of registered users on a website, among other tasks.

Python counts the items in a list starting with one, so you shouldn't run into any off-by-one errors when determining the length of a list.

Avoiding Index errors when working with lists

One type of error is common to see when you're working with lists for the first time. Let's say you have a list with three items, and you ask for the fourth item.

```
motorcycles=['honda','yamaha','suzuki']
print(motorcycles[3])
```

This example results in an `index error`:

```
Traceback (most recent call last):
     File "motorcycles.py", line 3, in <module>
          print(motorcycles[3])
IndexError: list index out of range
```

Python attempts to give you the item at index-3. But when it searches the list, no item in motorcycles has an index of 3. Because of the off-by-one nature of indexing in lists, this error is typical. People think the third item is item number 3, because they start counting at 1. But in Python the third item is number 2, because it starts indexing at 0.
An index error means Python can't figure out the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one. Then run the program again to see if the results are correct.
Keep in mind that whenever you want to access the last item in a list you use the index-1. This will always work, even if your list has changed size since the last time you accessed it:

```
motorcycles=['honda','yamaha','suzuki']
print(motorcycles[-1])
```

The index-1 always returns the last item in a list, in this case the value:

```
'suzuki':'suzuki'
```

The only time this approach will cause an error is when you request the last item from an empty list:

```
motorcycles=[]
print(motorcycles[-1])
```

No items are in motorcycles, so Python returns another index error:

```
Traceback (most recent call last):
     File "motorcyles.py", line 3, in <module>
          print(motorcycles[-1])
IndexError: list index out of range
```

If an index error occurs and you can't figure out how to resolve it, try printing your list or just printing the length of your list. Your list might look much different than you thought it did, especially if it has been managed dynamically by your program. Seeing the actual list, or the exact number of items in your list, can help you sort out such logical errors.

Module 4: Working with Lists

Learning objectives

1. Working efficiently with the elements in a list.
2. Working through a list using a for loop, how Python uses indentation to structure a program.
3. Making simple numerical lists, as well as a few operations you can perform on numerical lists.
4. Slicing a list to work with a subset of items and how to copy lists properly using a slice.
5. Working with tuples, which provide a degree of protection to a set of values that shouldn't change.

Working with Lists

Looping Through an Entire List

Let's say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. A for loop avoids both of these issues by letting Python manage these issues internally.
Let's use a for loop to print out each name in a list of magicians:

```
1 magicians=['alice','david','carolina']
2 for magician in magicians:
3     print(magician)
```

We begin by defining a list at `1`. At `2`, we define a for loop. This line tells Python to pull a name from the list magicians, and store it in the variable magician. At `3`, we tell Python to print the name that was just stored in magician. Python then repeats lines `2` and `3`, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

A Closer Look at Looping

In a simple loop like we used in *magicians.py*, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list magicians and store it in the variable magician. This first value is `'alice'`. Python then reads the next line:

```
print(magician)
```

Python prints the current value of magician, which is still `'alice'`. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, `'david'`, and stores that value in magician. Python then executes the line:

```
print(magician)
```

Python prints the current value of magician again, which is now `'david'`. Python repeats the entire loop once more with the last value in the list,`'carolina'`. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the for loop, so the program simply ends.

Doing More Work Within a for Loop

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
1     print(magician.title()+", that was a great trick!")
```

The only difference in this code is at `1` where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of magician is `'alice'`, so Python starts the first message with the name `'Alice'`. The second time through the message will begin with `'David'`, and the third time through the message will begin with `'Carolina'`.

The output shows a personalized message for each magician in the list:

```
  Alice, that was a great trick!
  David, that was a great trick!
  Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line for `magician` in `magicians` is considered `inside the loop`, and each indented line is executed once for each value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
     print(magician.title() + ", that was a great trick!")
1    print("I can't wait to see your next trick,"+magician.title()+
          ".\n")
```

Because we have indented both print statements, each line will be executed once for every magician in the list. The newline `("\n")` in the second print statement `1` inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
  Alice, that was a great trick!
  I can't wait to see your next trick, Alice.

  David, that was a great trick!
  I can't wait to see your next trick, David.

  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice, you'll often find it useful to do a number of different operations with each item in a list when you use a for loop.

Doing Something After a for Loop

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the for loop without indentation:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
      print(magician.title()+", that was a great trick!")
      print("I can't wait to see your next trick,"+magician.title()+
      ".\n")

1    print("Thank you, everyone. That was a great magic show!")
```

The first two print statements are repeated once for each magician in the list, as you saw earlier. However, because the line at `1` is not indented, it's printed only once:

```
  Alice, that was a great trick!
  I can't wait to see your next trick, Alice.

  David, that was a great trick!
  I can't wait to see your next trick, David.

  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.

  Thank you, everyone. That was a great magic show!
```

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire data set. Python uses indentation to determine when one line of code is connected to the line above it. In the previous examples, the lines that printed messages to individual magicians were part of the for loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure.

Forgetting to Indent

Always indent the line after the for statement in a loop. If you forget, Python will remind you:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
1    print(magician)
```

The print statement at 1 should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with.

```
  File "magicians.py", line 3
      print(magician)
          ^
  Indentation Error: expected an indented block
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the for statement.

Forgetting to Indent Additional Lines
This is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
     print(magician.title()+", that was a great trick!")
1    print("I can't wait to see your next trick, "+magician.title()+
          ".\n")
```

The print statement at 1 is supposed to be indented, but because Python finds at least one indented line after the for statement, it doesn't report an error. As a result, the first print statement is executed once for each name in the list because it is indented. The second print statement is not indented, so it is executed only once after the loop has finished running. Because the final value of magician is `'carolina'`, she is the only one who receives the "`looking forward to the next trick`" message:

```
  Alice, that was a great trick!
  David, that was a great trick!
  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.
```

This is a `logical error`. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

Indenting Unnecessarily
If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

```
  message="Hello Python world!"
1    print(message)
```

We don't need to indent the print statement at 1, because it doesn't belong to the line above it; hence, Python reports that error:

```
File "hello_world.py", line 2
     print(message)
         ^
Indentation Error: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs, you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a for loop.

Indenting Unnecessarily After the Loop

Let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

```
  magicians=['alice', 'david', 'carolina']
  for magician in magicians:
     print(magician.title()+", that was a great trick!")
     print("I can't wait to see your next trick,"+magician.title()+
     ".\n")

1 print("Thank you everyone, that was a great magic show!")
```

Because the line at 1 is indented, it's printed once for each person in the list, as you can see at 2:

```
  Alice, that was a great trick!
  I can't wait to see your next trick, Alice.

2 Thank you everyone, that was a great magic show!
  David, that was a great trick!
  I can't wait to see your next trick, David.

2 Thank you everyone, that was a great magic show!
  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.

2 Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in "Forgetting to Indent Additional Lines". Because Python doesn't know what you're trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, determine whether you just need to unintent the code for that action.

Forgetting the Colon

The colon at the end of a for statement tells Python to interpret the next line as the start of a loop.

```
  magicians=['alice','david','carolina']
1 for magician in magicians
      print(magician)
```

If you accidentally forget the colon, as shown at `1`, you'll get a syntax error because Python doesn't know what you're trying to do.

Using the `range()` Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

```
for value in range(1,5):
    print(value)
```

Although this code looks like it should print the numbers from `1` to `5,` it doesn't print the number `5`:

```
1
2
3
4
```

In this example, `range()` prints only the numbers `1` through `4`. This is another result of the off-by-one behaviour you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been `5` in this case.
To print the numbers from `1` to `5`, you would use `range(1,6)`:

```
for value in range(1,6):
print(value)
```

This time the output starts at 1 and ends at 5:

```
1
2
3
4
5
```

If your output is different than what you expect when you're using `range()`, try adjusting your end value by `1`.

Using `range()` to Make a List of Numbers

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers=list(range(1,6))
print(numbers)
```

And this is the result:

```
[1, 2, 3, 4, 5]
```

We can also use the `range()` function to tell Python to skip numbers in a given range. For example, here's how we would list the even numbers between 1 and 10:

```
even_numbers=list(range(2,11,2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (`**`) represent exponents. Here's how you might put the first 10 square numbers into a list:

```
1 squares=[]
2 for value in range(1,11):
3     square=value**2
4     squares.append(square)

5 print(squares)
```

We start with an empty list called squares at `1`. At `2`, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop, the current value is raised to the second power and stored in the variable square at `3`. At `4`, each new value of square is appended to the list squares. Finally, when the loop has finished running, the list of squares is printed at `5`:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable square and append each new value directly to the list:

```
  squares=[]
  for value in range(1,11):
1     squares.append(value**2)
```

```
print(squares)
```

The code at `1` does the same work as the lines at `3` and `4` in *squares.py*. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

Simple Statistics with a List of Numbers

A few Python functions are specific to lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits=[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.

List Comprehensions

The approach described earlier for generating the list squares consisted of using three or four lines of code. A list comprehension allows you to generate this same list in just one line of code. A list comprehension combines the for loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but we have included them here because you'll most likely see them as soon as you start looking at other people's code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

```
squares=[value**2 for value in range(1,11)]
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as squares. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example, the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is for value in `range(1,11)`, which feeds the values 1 through 10 into the expression `value**2`. Notice that no colon is used at the end of the for statement.

The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices 0 through 3, which would return elements 0, 1, and 2.
The following example involves a list of players on a team:

```
  players=['charles','martina','michael','florence','eli']
1 print(players[0:3])
```

The code at `1` prints a slice of this list, which includes just the first three players. The output retains the structure of the list and includes the first three players in the list:

```
['charles','martina','michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index-1 and end at index-4:

```
players=['charles','martina','michael','florence','eli']
print(players[1:4])
```

This time the slice starts with `'martina'` and ends with `'florence'`:

```
['martina','michael','florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players=['charles','martina','michael','florence','eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles','martina','michael','florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index-2 and omit the second index:

```
players=['charles','martina','michael','florence','eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael','florence','eli']
```

This syntax allows you to output all of the elements from any point in your list to the end regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
  players=['charles','martina','michael','florence','eli']
  print(players[-3:])
```

This prints the names of the last three players and would continue to work as the list of players changes in size.

Looping Through a Slice

You can use a slice in a for loop if you want to loop through a subset of the elements in a list. In the next example we loop through the first three players and print their names as part of a simple roster:

```
  players=['charles','martina','michael','florence','eli']
  print("Here are the first three players on my team:")
1 for player in players[:3]:
      print(player.title())
```

Instead of looping through the entire list of players at `1`, Python loops through only the first three names:

```
  Here are the first three players on my team:
  Charles
  Martina
  Michael
```

Slices are very useful in a number of situations. For instance, when you're working with data, you can use slices to process your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

Copying a List

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list. For example,

```
1 my_foods=['pizza','falafel','carrot cake']
2 friend_foods=my_foods[:]

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

At `1` we make a list of the foods we like called `my_foods`. At `2` we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices and store the copy in `friend_foods`. When we print each list, we see that they both contain the same foods:

```
  My favourite foods are:
  ['pizza','falafel','carrot cake']
  My friend's favourite foods are:
  ['pizza','falafel','carrot cake']
```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favourite foods:

```
  my_foods=['pizza','falafel','carrot cake']
1 friend_foods=my_foods[:]

2 my_foods.append('cannoli')
3 friend_foods.append('ice cream')

  print("My favourite foods are:")
  print(my_foods)
  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

At 1, we copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example. Next, we add a new food to each list: at 2, we add `'cannoli'` to `my_foods`, and at 3, we add `'ice cream'` to `friend_foods`. We then print the two lists to see whether each of these foods is in the appropriate list.

```
  My favourite foods are:
4 ['pizza','falafel','carrot cake','cannoli']

  My friend's favourite foods are:
5 ['pizza','falafel','carrot cake','ice cream']
```

The output at 4 shows that `'cannoli'` now appears in our list of favourite foods but `'ice cream'` doesn't. At 5, we can see that `'ice cream'` now appears in our friend's list but `'cannoli'` doesn't. If we had simply set `friend_foods` equal to `my_foods`, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```
  my_foods=['pizza','falafel','carrot cake']

  #This doesn't work:
1 friend_foods=my_foods

  my_foods.append('cannoli')
  friend_foods.append('ice cream')

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Instead of storing a copy of `my_foods` in `friend_foods` at 1, we set `friend_foods` equal to `my_foods`. This syntax actually tells Python to connect the new variable `friend_foods` to the list that is already contained in `my_foods`, so now both variables point to the same list. As a result, when we add `'cannoli'` to `my_foods`, it will also appear in `friend_foods`. Likewise, 'ice cream' will appear in both lists, even though it appears to be added only to `friend_foods`.
The output shows that both lists are the same now, which is not what we wanted:

```
  My favourite foods are:
  ['pizza','falafel','carrot cake','cannoli','ice cream']
  My friend's favourite foods are:
  ['pizza','falafel','carrot cake','cannoli','ice cream']
```

Don't worry about the details in this example for now. Basically, if you're trying to work with a copy of a list and you see unexpected behaviour, make sure you are copying the list using a slice, as we did in the first example.

Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.
For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

```
1 dimensions=(200,50)
2 print(dimensions[0])
  print(dimensions[1])
```

We define the tuple dimensions at 1, using parentheses instead of square brackets. At 2, we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
  200 50
```

Let's see what happens if we try to change one of the items in the tuple dimensions:

```
  dimensions=(200,50)
1 dimensions[0]=250
```

The code at 1 tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
  Traceback (most recent call last):
     File "dimensions.py", line 3, in <module>
          dimensions[0] = 250
  TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

Looping Through All Values in a Tuple
You can loop over all the values in a tuple using a for loop, just as you did with a list:

```
  dimensions=(200,50)
  for dimension in dimensions:
      print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
  200
  50
```

Writing over a Tuple
Although you can't modify a tuple, you can assign a new value to a variable that holds a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
1 dimensions=(200,50)
  print("Original dimensions:")
  for dimension in dimensions:
      print(dimension)

2 dimensions=(400,100)
3 print("\nModified dimensions:")
  for dimension in dimensions:
      print(dimension)
```

The block at `1` defines the original tuple and prints the initial dimensions. At `2`, we store a new tuple in the variable dimensions. We then print the new dimensions at `3`. Python doesn't raise any errors this time, because overwriting a variable is valid:

```
  Original dimensions:
  200
  50

  Modified dimensions:
  400
  100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

Python Programming
Module 5: `if` Statements

Learning objectives

1. Writing conditional tests, which always evaluate to True or False.
2. Writing simple if statements, if-else chains, and if-elif-else chains.
3. Using if, if-else and if-elif-else structures to identify particular conditions you needed to test, and to know when those conditions have been met in your programs.
4. You learned to handle certain items in a list differently than all other items while continuing to utilize the efficiency of a for loop.
5. Defining a dictionary and how to work with the information stored in a dictionary.
6. Accessing and modify individual elements in a dictionary, and how to loop through all of the information in a dictionary.
7. Loop through a dictionary’s key-value pairs, its keys, and its values.
8. Nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

If statements

A simple example

The following short example shows how if tests let you respond to special situations correctly. Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value `'bmw'` should be printed in all uppercase. The following code loops through a list of car names and looks for the value `'bmw'`. Whenever the value is `'bmw'`, it's printed in uppercase instead of title case:

```
  cars = ['audi', 'bmw', 'subaru', 'toyota']

  for car in cars:
1     if car == 'bmw':
          print(car.upper())
      else:
          print(car.title())
```

The loop in this example first checks if the current value of car is `'bmw'` at `1`. If it is, the value is printed in uppercase. If the value of car is anything other than `'bmw'`, it's printed in title case:

```
Audi
BMW
Subaru
Toyota
```

Conditional tests

At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a conditional test. Python uses the values `True` and `False` to decide whether the code in an if statement should be executed. If a conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the if statement.

Checking for Equality

The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
1 >>> car = 'bmw'
2 >>> car == 'bmw'
  True
```

The line at `1` sets the value of car to `'bmw'` using a single equal sign, as you've seen many times already. The line at `2` checks whether the value of car is `'bmw'` using a double equal sign (= =). This equality operator returns `True` if the values on the left and right side of the operator match, and `False` if they don't match. The values in this example match, so Python returns `True`.
When the value of car is anything other than `'bmw'`, this test returns `False`:

```
1 >>> car = 'audi'
2 >>> car == 'bmw'
```

```
  False
```

A single equal sign is really a statement; you might read the code at `1` as "`Set the value of car equal to 'audi'.`" On the other hand, a double equal sign, like the one at `2`, asks a question: "Is the value of car equal to `'bmw'`?" Most programming languages use equal signs in this way.

Ignoring Case When Checking for Equality

Testing for equality is case sensitive in Python. For example, two values with different capitalization are not considered equal:

```
  >>> car = 'Audi'
  >>> car == 'audi'
  False
```

If case matters, this behaviour is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
  >>> car = 'Audi'
  >>> car.lower() == 'audi'
  True
```

This test would return `True` no matter how the value `'Audi'` is formatted because the test is now case insensitive. The `lower()` function doesn't change the value that was originally stored in car, so you can do this kind of comparison without affecting the original variable:

```
1 >>> car = 'Audi'
2 >>> car.lower() == 'audi'
  True
3 >>> car
  'Audi'
```

At `1`, we store the capitalized string `'Audi'` in the variable car. At `2`, we convert the value of car to lowercase and compare the lowercase value to the string `'audi'`. The two strings match, so Python returns `True`. At `3`, we can see that the value stored in car has not been affected by the conditional test.

Checking for Inequality

When you want to determine whether two values are not equal, you can combine an exclamation point and an equal sign `(!=)`. The exclamation point represents not, as it does in many programming languages.
For example,

```
  requested_topping = 'mushrooms'

1 if requested_topping != 'anchovies':
  print("Hold the anchovies!")
```

The line at 1 compares the value of `requested_topping` to the value `'anchovies'`. If these two values do not match, Python returns `True` and executes the code following the if statement. If the two values match, Python returns `False` and does not run the code following the if statement.
Because the value of `requested_topping` is not `'anchovies'`, the print statement is executed:

```
  Hold the anchovies!
```

Most of the conditional expressions you write will test for equality, but sometimes you'll find it more efficient to test for inequality.

Numerical Comparisons

Testing numerical values is pretty straight forward. For example, you can test to see if two numbers are not equal:

```
  answer = 17

1 if answer != 42:
      print("That is not the correct answer. Please try again!")
```

The conditional test at 1 passes, because the value of answer (`17`) is not equal to `42`. Because the test passes, the indented code block is executed:

```
  That is not the correct answer. Please try again!
```

You can include various mathematical comparisons in your conditional statements as well, such as less than, less than or equal to, greater than, and greater than or equal to:

```
  >>> age = 19
  >>> age < 21
  True
  >>> age <= 21
  True
  >>> age > 21
  False
  >>> age >= 21
  False
```

Checking Multiple Conditions

You may want to check multiple conditions at the same time. For example, sometimes you might need two conditions to be `True` to take an action. Other times you might be satisfied with just one condition being `True`. The keywords `and` and `or` can help you in these situations.

Using and to Check Multiple Conditions

To check whether two conditions are both `True` simultaneously, use the keyword `and` to combine the two conditional tests; if each test passes, the overall expression evaluates to `True`. If either test fails or if both tests fail, the expression evaluates to `False`.
For example, you can check whether two people are both over 21 using the following test:

```
1 >>> age_0 = 22
  >>> age_1 = 18
2 >>> age_0 >= 21 and age_1 >= 21
  False
3 >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

At 1, we define two ages, `age_0` and `age_1`. At 2, we check whether both ages are 21 or older. The test on the left passes, but the test on the right fails, so the overall conditional expression evaluates to `False`. At 3, we change `age_1` to 22. The value of `age_1` is now greater than 21, so both individual tests pass, causing the overall conditional expression to evaluate as `True`.

Using `or` to Check Multiple Conditions

The keyword `or` allows you to check multiple conditions as well, but it passes when either or both of the individual tests pass. An or expression fails only when both individual tests fail.
Let's consider two ages again, but this time we'll look for only one person to be over 21:

```
1 >>> age_0 = 22
  >>> age_1 = 18
2 >>> age_0 >= 21 or age_1 >= 21
  True
3 >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

We start with two age variables again at 1. Because the test for `age_0` at 2 passes, the overall expression evaluates to `True`. We then lower `age_0` to 18. In the test at 3, both tests now fail and the overall expression evaluates to `False`.

Checking Whether a Value Is in a List

To find out whether a particular value is already in a list, use the keyword `in`. Let's consider some code you might write for a pizzeria. We'll make a list of toppings a customer has requested for a pizza and then check whether certain toppings are in the list.

```
  >>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
1 >>> 'mushrooms' in requested_toppings
  True
2 >>> 'pepperoni' in requested_toppings
  False
```

At `1` and `2`, the keyword in tells Python to check for the existence of `'mushrooms'` and `'pepperoni'` in the list `requested_toppings`. This technique is quite powerful because you can create a list of essential values, and then easily check whether the value you're testing matches one of the values in the list.

Checking Whether a Value Is Not in a List

You can use the keyword `not in` this situation. For example, consider a list of users who are banned from commenting in a forum. You can check whether a user has been banned before allowing that person to submit a comment:

```
  banned_users = ['andrew', 'carolina', 'david']
  user = 'marie'

1 if user not in banned_users:
      print(user.title() + ", you can post a response if you wish.")
```

The line at `1` reads quite clearly. If the value of user is not in the list `banned_users`, Python returns `True` and executes the indented line.
The user `'marie'` is not in the list `banned_users`, so she sees a message inviting her to post a response:

```
  Marie, you can post a response if you wish.
```

Boolean Expressions

A Boolean expression is just another name for a conditional test. A Boolean value is either `True` or `False`, just like the value of a conditional expression after it has been evaluated.
Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website:

```
  game_active = True
  can_edit = False
```

Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

If statements

Simple if Statements

The simplest kind of if statement has one test and one action:

```
  if conditional_test:
      do something.
```

You can put any conditional test in the first line and just about any action in the indented block following the test. Let's say we have a variable representing a person's age, and we want to know if that person is old enough to vote. The following code tests whether the person can vote:

```
  age = 19
1 if age >= 18:
2     print("You are old enough to vote!")
```

At `1`, Python checks to see whether the value in age is greater than or equal to 18. It is, so Python executes the intended print statement at `2`:

```
  You are old enough to vote!
```

You can have as many lines of code as you want in the block following the if statement. Let's add another line of output if the person is old enough to vote, asking if the individual has registered to vote yet:

```
  age = 19
  if age >= 18:
      print("You are old enough to vote!")
      print("Have you registered to vote yet?")
```

The conditional test passes, and both print statements are indented, so both lines are printed:

```
  You are old enough to vote!
  Have you registered to vote yet?
```

If the value of age is less than 18, this program would produce no output.

if-else Statements

Often, you'll want to take one action when a conditional test passes and a different action in all other cases. We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
  age = 17
1 if age >= 18:
      print("You are old enough to vote!")
      print("Have you registered to vote yet?")
2 else:
      print("Sorry, you are too young to vote.")
      print("Please register to vote as soon as you turn 18!")
```

If the conditional test at `1` passes, the first block of indented print statements is executed. If the test evaluates to `False`, the else block at `2` is executed. Because age is less than 18 this time, the conditional test fails and the code in the else block is executed:

```
  Sorry, you are too young to vote.
  Please register to vote as soon as you turn 18!
```

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The if-else structure works well in situations in which you want Python to always execute one of two possible actions. In a simple if-else chain like this, one of the two actions will always be executed.

The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if-elif-else` syntax. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order until one passes. When a test passes, the code following that test is executed and Python skips the rest of the tests.

Many real-world situations involve more than two possible conditions. For example, consider an amusement park that charges different rates for different age groups:

```
  Admission for anyone under age 4 is free.
  Admission for anyone between the ages of 4 and 18 is $5.
  Admission for anyone age 18 or older is $10.
```

How can we use an if statement to determine a person's admission rate? The following code tests for the age group of a person and then prints an admission price message:

```
  age = 12

1 if age < 4:
      print("Your admission cost is $0.")
2 elif age < 18:
      print("Your admission cost is $5.")
3 else:
      print("Your admission cost is $10.")
```

The if test at `1` tests whether a person is under 4 years old. If the test passes, an appropriate message is printed and Python skips the rest of the tests. The elif line at `2` is really another if test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is less than 18, an appropriate message is printed and Python skips the else block. If both the if and elif tests fail, Python runs the code in the else block at `3`.

In this example, the test at `1` evaluates to `False`, so its code block is not executed. However, the second test evaluates to `True` (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

```
  Your admission cost is $5.
```

Any age greater than 17 would cause the first two tests to fail. In these situations, the else block would be executed and the admission price would be $10.

Rather than printing the admission price within the if-elif-else block, it would be more concise to set just the price inside the if-elif-else chain and then have a simple print statement that runs after the chain has been evaluated:

```
  age = 12
  if age < 4:
1     price = 0
  elif age < 18:
2     price = 5
  else:
3     price = 10

4 print("Your admission cost is $" + str(price) + ".")
```

The lines at `1`, `2`, and `3` set the value of price according to the person's age, as in the previous example. After the price is set by the if-elif-else chain, a separate unindented print statement `4` uses this value to display a message reporting the person's admission price.
This code produces the same output as the previous example, but the purpose of the if-elif-else chain is narrower. Instead of determining a price and displaying a message, it simply determines the admission price. In addition to being more efficient, this revised code is easier to modify than the original approach. To change the text of the output message, you would need to change only one print statement rather than three separate print statements.

Using Multiple **`elif`** *Blocks*

You can use as many elif blocks in your code as you like. For example, if the amusement park were to implement a discount for seniors, you could add one more conditional test to the code to determine whether someone qualified for the senior discount. Let's say that anyone 65 or older pays half the regular admission, or $5:

```
  age = 12

  if age < 4:
      price = 0
  elif age < 18:
      price = 5
1 elif age < 65:
      price = 10
2 else:
      price = 5

  print("Your admission cost is $" + str(price) + ".")
```

Most of this code is unchanged. The second `elif` block at `1` now checks to make sure a person is less than age 65 before assigning them the full admission rate of $10. Notice that the value assigned in the else block at `2` needs to be changed to $5, because the only ages that make it to this block are people 65 or older.

Omitting the else Block

Python does not require an else block at the end of an if-elif chain. Sometimes an else block is useful; sometimes it is clearer to use an additional elif statement that catches the specific condition of interest:

```
  age = 12

  if age < 4:
      price = 0
  elif age < 18:
      price = 5
  elif age < 65:
      price = 10
1 elif age >= 65:
      price = 5

  print("Your admission cost is $" + str(price) + ".")
```

The extra elif block at 1 assigns a price of $5 when the person is 65 or older, which is a bit clearer than the general else block. With this change, every block of code must pass a specific test in order to be executed.

The else block is a catch all statement. It matches any condition that wasn't matched by a specific if or elif test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you are testing for, consider using a final elif block and omit the else block. As a result, you'll gain extra confidence that your code will run only under the correct conditions.

Testing Multiple Conditions

The if-elif-else chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests. This behaviour is beneficial, because it's efficient and allows you to test for one specific condition.

However, sometimes it's important to check all of the conditions of interest. In this case, you should use a series of simple if statements with no elif or else blocks. This technique makes sense when more than one condition could be `True` and you want to act on every condition that is `True`.

Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

```
1 requested_toppings = ['mushrooms', 'extra cheese']

2 if 'mushrooms' in requested_toppings:
      print("Adding mushrooms.")
3 if 'pepperoni' in requested_toppings:
      print("Adding pepperoni.")
4 if 'extra cheese' in requested_toppings:
      print("Adding extra cheese.")

  print("\nFinished making your pizza!")
```

Because every condition in this example is evaluated, both mushrooms and extra cheese are added to the pizza:

```
Adding mushrooms.
Adding extra cheese.
```

```
Finished making your pizza!
```

Checking for Special Items

Let’s continue with the pizzeria example. The pizzeria displays a message whenever a topping is added to your pizza, as it’s being made. The code for this action can be written very efficiently by making a list of toppings the customer has requested and using a loop to announce each topping as it’s added to the pizza:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```

The output is straightforward because this code is just a simple for loop:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

But what if the pizzeria runs out of green peppers? An `if` statement inside the `for` loop can handle this situation appropriately:

```
  requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

  for requested_topping in requested_toppings:
1 if requested_topping == 'green peppers':
      print("Sorry, we are out of green peppers right now.")
2 else:
      print("Adding " + requested_topping + ".")
  print("\nFinished making your pizza!")
```

This time we check each requested item before adding it to the pizza. The code at `1` checks to see if the person requested green peppers. If so, we display a message informing them why they can’t have green peppers. The else block at `2` ensures that all other toppings will be added to the pizza.
The output shows that each requested topping is handled appropriately:

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Checking That a List Is Not Empty

As an example, let's check whether the list of requested toppings is empty before building the pizza. If the list is empty, we'll prompt the user and make sure they want a plain pizza. If the list is not empty, we'll build the pizza just as we did in the previous examples:

```
1 requested_toppings = []

2 if requested_toppings:
      for requested_topping in requested_toppings:
           print("Adding " + requested_topping + ".")
      print("\nFinished making your pizza!")
3 else:
      print("Are you sure you want a plain pizza?")
```

This time we start out with an empty list of requested toppings at `1`. Instead of jumping right into a for loop, we do a quick check at `2`. When the name of a list is used in an `if` statement, Python returns `True` if the list contains at least one item; an empty list evaluates to `False`. If `requested_toppings` passes the conditional test, we run the same for loop we used in the previous example. If the conditional test fails, we print a message asking the customer if they really want a plain pizza with no toppings `3`.
The list is empty in this case, so the output asks if the user really wants a plain pizza:

```
Are you sure you want a plain pizza?
```

If the list is not empty, the output will show each requested topping being added to the pizza.

Using Multiple Lists

Let's watch out for unusual topping requests before we build a pizza. The following example defines two lists. The first is a list of available toppings at the pizzeria, and the second is the list of toppings that the user has requested. This time, each item in `requested_toppings` is checked against the list of available toppings before it's added to the pizza:

```
1 available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

2 requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

3 for requested_topping in requested_toppings:
4     if requested_topping in available_toppings:
           print("Adding " + requested_topping + ".")
5     else:
           print("Sorry, we don't have " + requested_topping + ".")

  print("\nFinished making your pizza!")
```

At `1`, we define a list of available toppings at this pizzeria. Note that this could be a tuple if the pizzeria has a stable selection of toppings. At `2`, we make a list of toppings that a customer has requested. Note the unusual request, `'french fries'`. At `3`, we loop through the list of requested toppings. Inside the loop, we first check to see if each requested topping is actually in the list of available toppings `4`. If it is,

we add that topping to the pizza. If the requested topping is not in the list of available toppings, the else block will run 5. The else block prints a message telling the user which toppings are unavailable.
This code syntax produces clean, informative output:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

In just a few lines of code, we’ve managed a real-world situation pretty effectively!

Dictionaries

A Simple Dictionary

Consider a game featuring aliens that can have different colours and point values. This simple dictionary stores information about a particular alien:

```
alien_0 = {'colour': 'green', 'points': 5}

print(alien_0['colour'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's colour and point value. The two print statements access and display that information, as shown here:

```
green
5
```

Working with dictionaries

A dictionary in Python is a collection of key-value pairs. Each key is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.
In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'colour': 'green', 'points': 5}
```

A key-value pair is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.
The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'colour': 'green'}
```

This dictionary stores one piece of information about `alien_0`, namely the alien's colour. The string `'colour'` is a key in this dictionary, and its associated value is `'green'.`

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

```
alien_0 = {'colour': 'green'}
print(alien_0['colour'])
```

This returns the value associated with the key `'colour'` from the dictionary `alien_0: green`

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
  alien_0 = {'colour': 'green', 'points': 5}
```

Now you can access either the colour or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
  alien_0 = {'colour': 'green', 'points': 5}

1 new_points = alien_0['points']
2 print("You just earned " + str(new_points) + " points!")
```

Once the dictionary has been defined, the code at `1` pulls the value associated with the key `'points'` from the dictionary. This value is then stored in the variable `new_points`. The line at `2` converts this integer value to a string and prints a statement about how many points the player just earned:

```
  You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

To add two new pieces of information to the `alien_0` dictionary: the alien's x and y-coordinates, which will help us display the alien in a particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

```
  alien_0 = {'colour': 'green', 'points': 5}
  print(alien_0)

1 alien_0['x_position'] = 0
2 alien_0['y_position'] = 25
  print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At `1`, we add a new key-value pair to the dictionary: key `'x_position'` and value 0. We do the same for key `'y_position'` at `2`. When we print the modified dictionary, we see the two-additional key-value pairs:

```
  {'colour': 'green', 'points': 5}
  {'colour': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify colour and point value, and two more specify the alien's position. Notice that the order of the key-value pairs does not match the order in which we added them. Python doesn't care about the order in which you store each key-value pair; it cares only about the connection between each key and its value.

Starting with an Empty Dictionary

To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

```
alien_0 = {}

alien_0['colour'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add colour and point values to it. The result is the dictionary we've been using in previous examples:

```
{'colour': 'green', 'points': 5}3e
```

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

```
alien_0 = {'colour': 'green'}
print("The alien is " + alien_0['colour'] + ".")

alien_0['colour'] = 'yellow'
print("The alien is now " + alien_0['colour'] + ".")
```

We first define a dictionary for `alien_0` that contains only the alien's colour; then we change the value associated with the key `'colour'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.
For example, let's remove the key `'points'` from the `alien_0` dictionary along with its value:

```
alien_0 = {'color': 'green', 'points': 5}
```

```
  print(alien_0)

1 del alien_0['points']
  print(alien_0)
```

The line at `1` tells Python to delete the key `'points'` from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key `'points'` and its value of 5 are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favourite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

```
favourite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
     }
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press enter after the opening brace. Then indent the next line one level (four spaces), and write the first key-value pair, followed by a comma. From this point forward when you press enter, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.
Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.
To use this dictionary, given the name of a person who took the poll, you can easily look up their favourite language:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 print("Sarah's favourite language is " +
2    favourite_languages['sarah'].title() +
3    ".")
```

To see which language Sarah chose, we ask for the value at:

```
favourite_languages['sarah']
```

This syntax is used in the print statement at `2`, and the output shows Sarah's favourite language:

```
Sarah's favorite language is C.
```

Looping through a dictionary

A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs

Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The following dictionary would store one person's username, first name, and last name:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

What if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a for loop:

```
  user_0 = {
      'username': 'efermi',
      'first': 'enrico',
      'last': 'fermi',
      }

1 for key, value in user_0.items():
2     print("\nKey: " + key)
3     print("Value: " + value)
```

As shown at `1`, to write a for loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. This code would work just as well if you had used abbreviations for the variable names, like this:

```
  for k,
2 in user_0.items()
```

The second half of the for statement at `1` includes the name of the dictionary followed by the method `items()`, which returns a list of key-value pairs. The for loop then stores each of these pairs in the two variables provided. In the preceding example, we use the variables to print each key 2, followed by the associated value 3. The "\n" in the first print statement ensures that a blank line is inserted before each key-value pair in the output:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Notice again that the key-value pairs are not returned in the order in which they were stored, even when looping through a dictionary. Python doesn't care about the order in which key-value pairs are stored; it tracks only the connections between individual keys and their values.

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
     }
1 for name, language in favourite_languages.items():
2    print(name.title() + "'s favourite language is " +
          language.title() + ".")
```

The code at 1 tells Python to loop through each key-value pair in the dictionary. As it works through each pair the key is stored in the variable name, and the value is stored in the variable language. These descriptive names make it much easier to see what the print statement at 2 is doing.
Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favourite language is Python.
Sarah's favourite language is C.
Phil's favourite language is Python.
Edward's favourite language is Ruby.
```

This type of looping would work just as well if our dictionary stored the results from polling a thousand or even a million people.

Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don't need to work with all of the values in a dictionary. Let's loop through the `favourite_languages` dictionary and print the names of everyone who took the poll:

```
  favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

1 for name in favourite_languages.keys():
      print(name.title())
```

The line at `1` tells Python to pull all the keys from the dictionary `favourite_languages` and store them one at a time in the variable name. The output shows the names of everyone who took the poll:

```
Jen
Sarah
Phil
Edward
```

Looping through the keys is actually the default behaviour when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favourite_languages:
```

rather than . . .

```
for name in favourite_languages.keys():
```

You can choose to use the `keys()` method explicitly if it makes your code easier to read, or you can omit it if you wish.
You can access the value associated with any key you care about inside the loop by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favourite language:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 friends = ['phil', 'sarah']
  for name in favourite_languages.keys():
     print(name.title())

2    if name in friends:
           print(" Hi " + name.title() +
               ", I see your favorite language is " +
3               favorite_languages[name].title() + "!")
```

At 1, we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then at 2, we check to see whether the name we are working with is in the list friends. If it is, we print a special greeting, including a reference to their language choice. To access the favourite language at 3, we use the name of the dictionary and the current value of name as the key. Everyone's name is printed, but our friends receive a special message:

```
  Edward
  Phil
      Hi Phil, I see your favourite language is Python!
  Sarah
      Hi Sarah, I see your favourite language is C!
  Jen
```

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if `Erin` took the poll:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 if 'erin' not in favourite_languages.keys():
     print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: It actually returns a list of all the keys, and the line at 1 simply checks if `'erin'` is in this list. Because she's not, a message is printed inviting her to take the poll:

```
  Erin, please take our poll!
```

Looping Through a Dictionary's Keys in Order

One way to return items in a certain order is to sort the keys as they're returned in the for loop. You can use the `sorted()` function to get a copy of the keys in order:

```
  favourite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

for name in sorted(favourite_languages.keys()):
    print(name.title() + ", thank you for taking the poll.")
```

This `for` statement is like other for statements except that we've wrapped the `sorted()` function around the `dictionary.keys()` method. This tells Python to list all keys in the dictionary and sort that list before looping through it. The output shows everyone who took the poll with the names displayed in order:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Looping Through All Values in a Dictionary

If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a list of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll without the name of the person who chose each language:

```
favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favourite_languages.values():
    print(language.title())
```

The `for` statement here pulls each value from the dictionary and stores it in the variable language. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a `set`. A set is similar to a list except that each item in the set must be unique:

```
  favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
     }

  print("The following languages have been mentioned:")
1 for language in set(favourite_languages.values()):
     print(language.title())
```

When you wrap `set()` around a list that contains duplicate items, Python identifies the unique items in the list and builds a set from those items. At 1, we use `set()` to pull out the unique languages in `favourite_languages.values()`.
The result is a non-repetitive list of languages that have been mentioned by people taking the poll:
The following languages have been mentioned:

```
  Python
  C
  Ruby
```

Nesting

A List of Dictionaries

The following code builds a list of three aliens:

```
  alien_0 = {'colour': 'green', 'points': 5}
  alien_1 = {'colour': 'yellow', 'points': 10}
  alien_2 = {'colour': 'red', 'points': 15}

1 aliens = [alien_0, alien_1, alien_2]

  for alien in aliens:
     print(alien)
```

We first create three dictionaries, each representing a different alien. At 1 we pack each of these dictionaries into a list called aliens. Finally, we loop through the list and print out each alien:

```
  {'colour': 'green', 'points': 5}
  {'colour': 'yellow', 'points': 10}
  {'colour': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use `range()` to create a fleet of 30 aliens:

```
  # Make an empty list for storing aliens.
  aliens = []

  # Make 30 green aliens.
1 for alien_number in range(30):
2    new_alien = { 'colour': 'green', 'points': 5, 'speed': 'slow' }
3    aliens.append(new_alien)

  # Show the first 5 aliens:
4 for alien in aliens[:5]:
      print(alien)
  print("...")

  # Show how many aliens have been created.
5 print("Total number of aliens: " + str(len(aliens)))
```

This example begins with an empty list to hold all of the aliens that will be created. At `1`, `range()` returns a set of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs we create a new alien `2` and then append each new alien to the list aliens `3`. At `4`, we use a slice to print the first five aliens, and then at `5` we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
...

Total number of aliens: 30
```

These aliens all have the same characteristics, but Python considers each one a separate object, which allows us to modify each alien individually.
How might you work with a set of aliens like this? Imagine that one aspect of a game has some aliens changing colour and moving faster as the game progresses. When it's time to change colours, we can use a for loop and an if statement to change the colour of aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (0,30):
    new_alien = {'colour': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[0:3]:
    if alien['colour'] == 'green':
         alien['colour'] = 'yellow'
         alien['speed'] = 'medium'
         alien['points'] = 10

# Show the first 5 aliens:
for alien in aliens[0:5]:
    print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now but that won't always be the case, so we write an if statement to make sure we're only modifying green aliens. If the alien is green, we change the colour to `'yellow'`, the speed to `'medium'`, and the point value to 10, as shown in the following output:

```
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
...
```

You could expand this loop by adding an elif block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
     if alien['colour'] == 'green':
         alien['colour'] = 'yellow'
         alien['speed'] = 'medium'
         alien['points'] = 10
     elif alien['colour'] == 'yellow':
         alien['colour'] = 'red'
         alien['speed'] = 'fast'
         alien['points'] = 15
```

It's common to store a number of dictionaries in a list when each dictionary contains many kinds of information about one object.

A List in a Dictionary

Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary. For example, consider how you might describe a pizza that someone is ordering. If you were to use only a list, all you could really store is a list of the pizza's toppings. With a dictionary, a list of toppings can be just one aspect of the pizza you're describing.

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key `'toppings'`. To use the items in the list, we give the name of the dictionary and the key `'toppings'`, as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

```
  # Store information about a pizza being ordered.
1 pizza = {
       'crust': 'thick',
       'toppings': ['mushrooms', 'extra cheese'],
       }

  # Summarize the order.
2 print("You ordered a " + pizza['crust'] + "-crust pizza "
      + "with the following toppings:")

3 for topping in pizza['toppings']:
      print("\t" + topping)
```

We begin at 1 with a dictionary that holds information about a pizza that has been ordered. One key in the dictionary is `'crust'`, and the associated value is the string `'thick'`. The next key, `'toppings'`, has a list as its value that stores all requested toppings. At 2, we summarize the order before building the pizza. To print the toppings, we write a for loop 3. To access the list of toppings, we use the key `'toppings'`, and Python grabs the list of toppings from the dictionary.
The following output summarizes the pizza that we plan to build:

```
  You ordered a thick-crust pizza with the following toppings:
      mushrooms
      extra cheese
```

In the earlier example of favourite programming languages, if we were to store each person's responses in a list, people could choose more than one favourite language. When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's for loop, we use another for loop to run through the list of languages associated with each person:

```
1 favourite_languages = {
        'jen': ['python', 'ruby'],
        'sarah': ['c'],
        'edward': ['ruby', 'go'],
        'phil': ['python', 'haskell'],
        }

2 for name, languages in favourite_languages.items():
      print("\n" + name.title() + "'s favourite languages are:")
3     for language in languages:
              print("\t" + language.title())
```

As you can see at `1` the value associated with each name is now a list. When we loop through the dictionary at `2`, we use the variable name languages to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another for loop `3` to run through each person's list of favourite languages:

```
Jen's favourite languages are:
   Python
   Ruby
Sarah's favourite languages are:
   C

Phil's favourite languages are:
   Python
   Haskell

Edward's favourite languages are:
   Ruby
   Go
```

You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples or you're working with someone else's code with significant levels of nesting, most likely a simpler way to solve the problem exists.

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the user names as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

```
users = {
  'aeinstein': {
      'first': 'albert',
      'last': 'einstein',
      'location': 'princeton',
      },
  'mcurie': {
         'first': 'marie',
         'last': 'curie',
         'location': 'paris',
         },

   }
```

```
1 for username, user_info in users.items():
2    print("\nUsername: " + username)
3    full_name = user_info['first'] + " " + user_info['last']
          location = user_info['location']

4    print("\tFull name: " + full_name.title())
     print("\tLocation: " + location.title())
```

We first define a dictionary called users with two keys: one each for the usernames `'aeinstein'` and `'mcurie'`. The value associated with each key is a dictionary that includes each user's first name, last name, and location. At `1` we loop through the users dictionary. Python stores each key in the variable username, and the dictionary associated with each username goes into the variable `user_info`. Once inside the main dictionary loop, we print the username at `2`.

At `3` we start accessing the inner dictionary. The variable `user_info`, which contains the dictionary of user information, has three keys: `'first'`, `'last'`, and `'location'`. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user `4`:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton
Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Notice that the structure of each user's dictionary is identical. Although not required by Python, this structure makes nested dictionaries easier to work with. If each user's dictionary had different keys, the code inside the for loop would be more complicated.

Python Programming
Module 6: Dictionaries

Learning objectives

1. Defining a dictionary and how to work with the information stored in a dictionary.
2. Accessing and modify individual elements in a dictionary, and how to loop through all of the information in a dictionary.
3. Loop through a dictionary’s key-value pairs, its keys, and its values.
4. Nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

Dictionaries

A Simple Dictionary

Consider a game featuring aliens that can have different colours and point values. This simple dictionary stores information about a particular alien:

```
alien_0 = {'colour': 'green', 'points': 5}

print(alien_0['colour'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's colour and point value. The two print statements access and display that information, as shown here:

```
green
5
```

Working with dictionaries

A dictionary in Python is a collection of key-value pairs. Each key is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'colour': 'green', 'points': 5}
```

A key-value pair is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'colour': 'green'}
```

This dictionary stores one piece of information about `alien_0`, namely the alien's colour. The string `'colour'` is a key in this dictionary, and its associated value is `'green'`.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

```
alien_0 = {'colour': 'green'}
print(alien_0['colour'])
```

This returns the value associated with the key `'colour'` from the dictionary `alien_0: green`

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
  alien_0 = {'colour': 'green', 'points': 5}
```

Now you can access either the colour or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
  alien_0 = {'colour': 'green', 'points': 5}

1 new_points = alien_0['points']
2 print("You just earned " + str(new_points) + " points!")
```

Once the dictionary has been defined, the code at `1` pulls the value associated with the key `'points'` from the dictionary. This value is then stored in the variable `new_points`. The line at `2` converts this integer value to a string and prints a statement about how many points the player just earned:

```
  You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

To add two new pieces of information to the `alien_0` dictionary: the alien's x and y-coordinates, which will help us display the alien in a particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

```
  alien_0 = {'colour': 'green', 'points': 5}
  print(alien_0)

1 alien_0['x_position'] = 0
2 alien_0['y_position'] = 25
  print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At `1`, we add a new key-value pair to the dictionary: key `'x_position'` and value 0. We do the same for key `'y_position'` at `2`. When we print the modified dictionary, we see the two-additional key-value pairs:

```
  {'colour': 'green', 'points': 5}
  {'colour': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify colour and point value, and two more specify the alien's position. Notice that the order of the key-value pairs does not match the order in which we added them. Python doesn't care about the order in which you store each key-value pair; it cares only about the connection between each key and its value.

Starting with an Empty Dictionary

To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

```
alien_0 = {}

alien_0['colour'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add colour and point values to it. The result is the dictionary we've been using in previous examples:

```
{'colour': 'green', 'points': 5}3e
```

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

```
alien_0 = {'colour': 'green'}
print("The alien is " + alien_0['colour'] + ".")

alien_0['colour'] = 'yellow'
print("The alien is now " + alien_0['colour'] + ".")
```

We first define a dictionary for `alien_0` that contains only the alien's colour; then we change the value associated with the key `'colour'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.
For example, let's remove the key `'points'` from the `alien_0` dictionary along with its value:

```
alien_0 = {'color': 'green', 'points': 5}
```

```
  print(alien_0)

1 del alien_0['points']
  print(alien_0)
```

The line at 1 tells Python to delete the key `'points'` from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key `'points'` and its value of 5 are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favourite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

```
favourite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
     }
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press enter after the opening brace. Then indent the next line one level (four spaces), and write the first key-value pair, followed by a comma. From this point forward when you press enter, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.
Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.
To use this dictionary, given the name of a person who took the poll, you can easily look up their favourite language:

```
favourite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
     }

print("Sarah's favourite language is " +
   favourite_languages['sarah'].title() +
   ".")
```

To see which language Sarah chose, we ask for the value at:

```
favourite_languages['sarah']
```

This syntax is used in the print statement at `2`, and the output shows Sarah's favourite language:

```
Sarah's favorite language is C.
```

Looping through a dictionary

A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs

Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The following dictionary would store one person's username, first name, and last name:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

What if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a for loop:

```
  user_0 = {
      'username': 'efermi',
      'first': 'enrico',
      'last': 'fermi',
      }

1 for key, value in user_0.items():
2     print("\nKey: " + key)
3     print("Value: " + value)
```

As shown at `1`, to write a for loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. This code would work just as well if you had used abbreviations for the variable names, like this:

```
  for k,
2 in user_0.items()
```

The second half of the for statement at `1` includes the name of the dictionary followed by the method `items()`, which returns a list of key-value pairs. The for loop then stores each of these pairs in the two variables provided. In the preceding example, we use the variables to print each key 2, followed by the associated value 3. The "\n" in the first print statement ensures that a blank line is inserted before each key-value pair in the output:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Notice again that the key-value pairs are not returned in the order in which they were stored, even when looping through a dictionary. Python doesn't care about the order in which key-value pairs are stored; it tracks only the connections between individual keys and their values.

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
     }
1 for name, language in favourite_languages.items():
2    print(name.title() + "'s favourite language is " +
           language.title() + ".")
```

The code at 1 tells Python to loop through each key-value pair in the dictionary. As it works through each pair the key is stored in the variable name, and the value is stored in the variable language. These descriptive names make it much easier to see what the print statement at 2 is doing.
Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favourite language is Python.
Sarah's favourite language is C.
Phil's favourite language is Python.
Edward's favourite language is Ruby.
```

This type of looping would work just as well if our dictionary stored the results from polling a thousand or even a million people.

Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don't need to work with all of the values in a dictionary. Let's loop through the `favourite_languages` dictionary and print the names of everyone who took the poll:

```
  favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

1 for name in favourite_languages.keys():
      print(name.title())
```

The line at `1` tells Python to pull all the keys from the dictionary `favourite_languages` and store them one at a time in the variable name. The output shows the names of everyone who took the poll:

```
Jen
Sarah
Phil
Edward
```

Looping through the keys is actually the default behaviour when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favourite_languages:
```

rather than . . .

```
for name in favourite_languages.keys():
```

You can choose to use the `keys()` method explicitly if it makes your code easier to read, or you can omit it if you wish.
You can access the value associated with any key you care about inside the loop by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favourite language:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 friends = ['phil', 'sarah']
  for name in favourite_languages.keys():
     print(name.title())

2    if name in friends:
          print(" Hi " + name.title() +
             ", I see your favorite language is " +
3             favorite_languages[name].title() + "!")
```

At 1, we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then at 2, we check to see whether the name we are working with is in the list friends. If it is, we print a special greeting, including a reference to their language choice. To access the favourite language at 3, we use the name of the dictionary and the current value of name as the key. Everyone's name is printed, but our friends receive a special message:

```
Edward
Phil
    Hi Phil, I see your favourite language is Python!
Sarah
    Hi Sarah, I see your favourite language is C!
Jen
```

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if `Erin` took the poll:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 if 'erin' not in favourite_languages.keys():
     print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: It actually returns a list of all the keys, and the line at 1 simply checks if `'erin'` is in this list. Because she's not, a message is printed inviting her to take the poll:

```
Erin, please take our poll!
```

Looping Through a Dictionary's Keys in Order

One way to return items in a certain order is to sort the keys as they're returned in the for loop. You can use the `sorted()` function to get a copy of the keys in order:

```
favourite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

for name in sorted(favourite_languages.keys()):
    print(name.title() + ", thank you for taking the poll.")
```

This `for` statement is like other for statements except that we've wrapped the `sorted()` function around the `dictionary.keys()` method. This tells Python to list all keys in the dictionary and sort that list before looping through it. The output shows everyone who took the poll with the names displayed in order:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Looping Through All Values in a Dictionary

If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a list of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll without the name of the person who chose each language:

```
favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favourite_languages.values():
    print(language.title())
```

The `for` statement here pulls each value from the dictionary and stores it in the variable language. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a `set`. A set is similar to a list except that each item in the set must be unique:

```
  favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
     }

  print("The following languages have been mentioned:")
1 for language in set(favourite_languages.values()):
      print(language.title())
```

When you wrap `set()` around a list that contains duplicate items, Python identifies the unique items in the list and builds a set from those items. At 1, we use `set()` to pull out the unique languages in `favourite_languages.values()`.
The result is a non-repetitive list of languages that have been mentioned by people taking the poll:
The following languages have been mentioned:

```
  Python
  C
  Ruby
```

Nesting

A List of Dictionaries

The following code builds a list of three aliens:

```
  alien_0 = {'colour': 'green', 'points': 5}
  alien_1 = {'colour': 'yellow', 'points': 10}
  alien_2 = {'colour': 'red', 'points': 15}

1 aliens = [alien_0, alien_1, alien_2]

  for alien in aliens:
      print(alien)
```

We first create three dictionaries, each representing a different alien. At 1 we pack each of these dictionaries into a list called aliens. Finally, we loop through the list and print out each alien:

```
  {'colour': 'green', 'points': 5}
  {'colour': 'yellow', 'points': 10}
  {'colour': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use `range()` to create a fleet of 30 aliens:

```
  # Make an empty list for storing aliens.
  aliens = []

  # Make 30 green aliens.
1 for alien_number in range(30):
2    new_alien = { 'colour': 'green', 'points': 5, 'speed': 'slow' }
3    aliens.append(new_alien)

  # Show the first 5 aliens:
4 for alien in aliens[:5]:
      print(alien)
  print("...")

  # Show how many aliens have been created.
5 print("Total number of aliens: " + str(len(aliens)))
```

This example begins with an empty list to hold all of the aliens that will be created. At `1`, `range()` returns a set of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs we create a new alien `2` and then append each new alien to the list aliens `3`. At `4`, we use a slice to print the first five aliens, and then at `5` we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
...

Total number of aliens: 30
```

These aliens all have the same characteristics, but Python considers each one a separate object, which allows us to modify each alien individually.
How might you work with a set of aliens like this? Imagine that one aspect of a game has some aliens changing colour and moving faster as the game progresses. When it's time to change colours, we can use a for loop and an if statement to change the colour of aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (0,30):
    new_alien = {'colour': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[0:3]:
    if alien['colour'] == 'green':
         alien['colour'] = 'yellow'
         alien['speed'] = 'medium'
         alien['points'] = 10

# Show the first 5 aliens:
for alien in aliens[0:5]:
    print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now but that won't always be the case, so we write an if statement to make sure we're only modifying green aliens. If the alien is green, we change the colour to `'yellow'`, the speed to `'medium'`, and the point value to 10, as shown in the following output:

```
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
...
```

You could expand this loop by adding an elif block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
     if alien['colour'] == 'green':
         alien['colour'] = 'yellow'
         alien['speed'] = 'medium'
         alien['points'] = 10
     elif alien['colour'] == 'yellow':
         alien['colour'] = 'red'
         alien['speed'] = 'fast'
         alien['points'] = 15
```

It's common to store a number of dictionaries in a list when each dictionary contains many kinds of information about one object.

A List in a Dictionary

Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary. For example, consider how you might describe a pizza that someone is ordering. If you were to use only a list, all you could really store is a list of the pizza's toppings. With a dictionary, a list of toppings can be just one aspect of the pizza you're describing.

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key `'toppings'`. To use the items in the list, we give the name of the dictionary and the key `'toppings'`, as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

```
  # Store information about a pizza being ordered.
1 pizza = {
       'crust': 'thick',
       'toppings': ['mushrooms', 'extra cheese'],
       }

  # Summarize the order.
2 print("You ordered a " + pizza['crust'] + "-crust pizza "
      + "with the following toppings:")

3 for topping in pizza['toppings']:
      print("\t" + topping)
```

We begin at 1 with a dictionary that holds information about a pizza that has been ordered. One key in the dictionary is `'crust'`, and the associated value is the string `'thick'`. The next key, `'toppings'`, has a list as its value that stores all requested toppings. At 2, we summarize the order before building the pizza. To print the toppings, we write a for loop 3. To access the list of toppings, we use the key `'toppings'`, and Python grabs the list of toppings from the dictionary.
The following output summarizes the pizza that we plan to build:

```
  You ordered a thick-crust pizza with the following toppings:
      mushrooms
      extra cheese
```

In the earlier example of favourite programming languages, if we were to store each person's responses in a list, people could choose more than one favourite language. When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's for loop, we use another for loop to run through the list of languages associated with each person:

```
1 favourite_languages = {
        'jen': ['python', 'ruby'],
        'sarah': ['c'],
        'edward': ['ruby', 'go'],
        'phil': ['python', 'haskell'],
        }

2 for name, languages in favourite_languages.items():
      print("\n" + name.title() + "'s favourite languages are:")
3     for language in languages:
          print("\t" + language.title())
```

As you can see at 1 the value associated with each name is now a list. When we loop through the dictionary at 2, we use the variable name languages to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another for loop 3 to run through each person's list of favourite languages:

```
Jen's favourite languages are:
   Python
   Ruby
Sarah's favourite languages are:
   C

Phil's favourite languages are:
   Python
   Haskell

Edward's favourite languages are:
   Ruby
   Go
```

You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples or you're working with someone else's code with significant levels of nesting, most likely a simpler way to solve the problem exists.

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the user names as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

```
users = {
  'aeinstein': {
      'first': 'albert',
      'last': 'einstein',
      'location': 'princeton',
      },
  'mcurie': {
         'first': 'marie',
         'last': 'curie',
         'location': 'paris',
         },

   }
```

```
1 for username, user_info in users.items():
2    print("\nUsername: " + username)
3    full_name = user_info['first'] + " " + user_info['last']
          location = user_info['location']

4    print("\tFull name: " + full_name.title())
      print("\tLocation: " + location.title())
```

We first define a dictionary called users with two keys: one each for the usernames `'aeinstein'` and `'mcurie'`. The value associated with each key is a dictionary that includes each user's first name, last name, and location. At `1` we loop through the users dictionary. Python stores each key in the variable username, and the dictionary associated with each username goes into the variable `user_info`. Once inside the main dictionary loop, we print the username at `2`.

At `3` we start accessing the inner dictionary. The variable `user_info`, which contains the dictionary of user information, has three keys: `'first'`, `'last'`, and `'location'`. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user `4`:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton
Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Notice that the structure of each user's dictionary is identical. Although not required by Python, this structure makes nested dictionaries easier to work with. If each user's dictionary had different keys, the code inside the for loop would be more complicated.

Python Programming
Module 7: User Input and while loops

Learning objectives

1. How to use `input()` to allow users to provide their own information in your programs.
2. Work with both text and numerical input and how to use while loops to make your programs run as long as your users want them to.
3. controlling the flow of a while loop by setting an active flag, using the break statement, and using the continue statement.
4. Using a `while` loop to move items from one list to another and how to remove all instances of a value from a list.
5. How while loops can be used with dictionaries.

User Input and while loops

How the `input()` function works

The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it stores it in a variable to make it convenient for you to work with.
For example, the following program asks the user to enter some text, then displays that message back to the user:

```
message = input("Tell me something, and I will repeat it back to
                you:")
print(message)
```

The `input()` function takes one argument: the *prompt*, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt Tell me something, and I will repeat it back to you:. The program waits while the user enters their response and continues after the user presses enter. The response is stored in the variable message, then `print(message)` displays the input back to the user:

```
Tell me something, and I will repeat it back to you: Hello
    everyone!
Hello everyone!
```

Writing Clear Prompts

Any statement that tells the user what to enter should work. For example:

```
name = input("Please enter your name: ")
print("Hello, " + name + "!")
```

Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text. For example:

```
Please enter your name: Eric
Hello, Eric!
```

You can store your prompt in a variable and pass that variable to the `input()` function. This allows you to build your prompt over several lines, then write a clean `input()` statement.

```
prompt = "If you tell us who you are, we can personalize the
          messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print("\nHello, " + name + "!")
```

This example shows one way to build a multi-line string. The first line stores the first part of the message in the variable prompt. In the second line, the operator `+=` takes the string that was stored in prompt and adds the new string onto the end.
The prompt now spans two lines, again with space after the question mark for clarity:

```
  If you tell us who you are, we can personalize the messages you
  see.
  What is your first name? Eric

  Hello, Eric!
```

Using `int()` to Accept Numeric Input
When you use the `input()` function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
  >>> age = input("How old are you? ")
  How old are you? 21
  >>> age
  '21'
```

The user enters the number `21`, but when we ask Python for the value of age, it returns `'21'`, the string representation of the numerical value entered. But if you try to use the input as a number, you'll get an error:

```
  >>> age = input("How old are you? ")
  How old are you? 21
1 >>> age >= 18
  Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
2 TypeError: unorderable types: str() >= int()
```

We can resolve this issue by using the `int()` function, which tells Python to treat the input as a numerical value. The `int()` function converts a string representation of a number to a numerical representation, as shown here:

```
  >>> age = input("How old are you? ")
  How old are you? 21
1 >>> age = int(age)
  >>> age >= 18 True
```

In this example, when we enter 21 at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by `int()` at `1`. Now Python can run the conditional test: it compares age (which now contains the numerical value 21) and 18 to see if age is greater than or equal to 18. This test evaluates to `True`.

How do you use the `int()` function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

```
height = input("How tall are you, in inches? ")
height = int(height)

 if height >= 36:
     print("\nYou're tall enough to ride!")
else:
     print("\nYou'll be able to ride when you're a little older.")
```

The program can compare height to 36 because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 36, we tell the user that they're tall enough:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

The Modulo Operator

A useful tool for working with numerical information is the *modulo operator* (`%`), which divides one number by another number and returns the remainder:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

The modulo operator doesn't tell you how many times one number fits into another; it just tells you what the remainder is.

When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd:

```
number = input("Enter a number, and I'll tell you if it's even or
odd: ")
number = int(number)

if number % 2 == 0:
  print("\nThe number " + str(number) + " is even.")
else:
  print("\nThe number " + str(number) + " is odd.")
```

Even numbers are always divisible by two, so if the modulo of a number and two is zero (here, `if number % 2 == 0`) the number is even. Otherwise, it's odd.

```
Enter a number, and I'll tell you if it's even or odd: 42
The number 42 is even.
```

Introducing `while` loops

The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or `while`, a certain condition is true.

The `while` Loop in Action

You can use a while loop to count up through a series of numbers. For example, the following while loop counts from 1 to 5:

```
current_number = 1
 while current_number <= 5:
    print(current_number)
    current_number += 1
```

In the first line, we start counting from 1 by setting the value of `current_number` to 1. The while loop is then set to keep running as long as the value of `current_number` is less than or equal to 5. The code inside the loop prints the value of `current_number` and then adds 1 to that value with `current_number += 1`. (The `+=` operator is shorthand for `current_number = current_number + 1`.)

Python repeats the loop as long as the condition `current_number <= 5` is true. Because 1 is less than 5, Python prints 1 and then adds 1, making the current number 2. Because 2 is less than 5, Python prints 2 and adds 1 again, making the current number 3, and so on. Once the value of `current_number` is greater than 5, the loop stops running and the program ends:

```
1
2
3
4
5
```

Letting the User Choose When to Quit

We can make the *parrot.py* program run as long as the user wants by putting most of the program inside a while loop. We'll define a *quit value* and then keep the program running as long as the user has not entered the quit value:

```
1 prompt = "\nTell me something, and I will repeat it back to you:"
  prompt += "\nEnter 'quit' to end the program. "
2 message = ""
3 while message != 'quit':
       message = input(prompt)
```

```
        print(message)
```

At `1`, we define a prompt that tells the user their two options: entering a message or entering the quit value (in this case, `'quit'`). Then we set up a variable message `2` to store whatever value the user enters. We define message as an empty string, `""`, so Python has something to check the first time it reaches the while line. The first time the program runs and Python reaches the while statement, it needs to compare the value of message to `'quit'`, but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this problem, we make sure to give message an initial value. Although it's just an empty string, it will make sense to Python and allow it to perform the comparison that makes the while loop work. This while loop `3` runs as long as the value of message is not `'quit'`.

The first time through the loop, message is just an empty string, so Python enters the loop. At `message = input(prompt)`, Python displays the prompt and waits for the user to enter their input. Whatever they enter is stored in message and printed; then, Python re-evaluates the condition in the while statement. As long as the user has not entered the word `'quit'`, the prompt is displayed again and Python waits for more input. When the user finally enters `'quit'`, Python stops executing the while loop and the program ends:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

This program works well, except that it prints the word `'quit'` as if it were an actual message. A simple if test fixes this:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message!= 'quit':
    message = input(prompt)

    if message!= 'quit':
        print(message)
```

Now the program makes a quick check before displaying the message and only prints the message if it does not match the quit value:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Using a Flag

In the previous example, we had the program perform certain tasks while a given condition was true. But what about more complicated programs in which many different events could cause the program to stop running?

For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a *flag*, acts as a signal to the program. We can write our programs so they run while the flag is set to `True` and stop running when any of several events sets the value of the flag to False. As a result, our overall while statement needs to check only one condition: whether or not the flag is currently `True`. Then, all our other tests (to see if an event has occurred that should set the flag to False) can be neatly organized in the rest of the program.

Let's add a flag to *parrot.py* from the previous section. This flag, which we'll call active (though you can call it anything), will monitor whether or not the program should continue running:

```
  prompt = "\nTell me something, and I will repeat it back to you:"
  prompt += "\nEnter 'quit' to end the program. "

1 active = True
2 while active:
      message = input(prompt)

3 if message == 'quit':
     active = False
4 else:
     print(message)
```

We set the variable active to True `1` so the program starts in an active state. Doing so makes the while statement simpler because no comparison is made in the while statement itself; the logic is taken care of in other parts of the program. As long as the active variable remains True, the loop will continue running `2`.

In the if statement inside the while loop, we check the value of message once the user enters their input. If the user enters `'quit'` at `3`, we set active to False, and the while loop stops. If the user enters anything other than `'quit'` at `4`, we print their input as a message.

This program has the same output as the previous example where we placed the conditional test directly in the while statement. But now that we have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as elif statements) for events that should cause active to become False. This is useful in complicated programs like games in which there may be many events that should each make the program stop running. When any of these events causes the active flag to become False, the main game loop will exit, a *Game Over* message can be displayed, and the player can be given the option to play again.

Using break to Exit a Loop

To exit a while loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the break statement. The break statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

For example, consider a program that asks the user about places they've visited. We can stop the while loop in this program by calling break as soon as the user enters the `'quit'` value:

```
  prompt = "\nPlease enter the name of a city you have visited:"
  prompt += "\n(Enter 'quit' when you are finished.) "

1 while True:
          city = input(prompt)

       if city == 'quit':
           break
       else:
           print("I'd love to go to " + city.title() + "!")
```

A loop that starts with while True `1` will run forever unless it reaches a break statement. The loop in this program continues asking the user to enter the names of cities they've been to until they enter `'quit'`. When they enter `'quit'`, the break statement runs, causing Python to exit the loop:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

Using continue in a Loop

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the continue statement to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

```
current_number = 0
while current_number < 10:
```

```
1       current_number += 1
        if current_number % 2 == 0:
            continue

        print(current_number)
```

First, we set `current_number` to 0. Because it's less than 10, Python enters the while loop. Once inside the loop, we increment the count by 1 at `1`, so `current_number` is 1. The if statement then checks the modulo of `current_number` and 2. If the modulo is 0 (which means `current_number` is divisible by 2), the continue statement tells Python to ignore the rest of the loop and return to the beginning. If the current number is not divisible by 2, the rest of the loop is executed and Python prints the current number:

```
1
3
5
7
9
```

Avoiding Infinite Loops

Every while loop needs a way to stop running so it won't continue to run for- ever. For example, this counting loop should count from 1 to 5:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

But if you accidentally omit the line `x += 1` (as shown next), the loop will run forever:

```
# This loop runs forever!
X = 1
while x <= 5:
    print(x)
```

Now the value of x will start at 1 but never change. As a result, the conditional test `x <= 5` will always evaluate to True and the while loop will run forever, printing a series of 1s, like this:

```
1
1
1
1
--snip--
```

If your program gets stuck in an infinite loop, press CTRL-C or just close the terminal window displaying your program's output. To avoid writing infinite loops, test every while loop and make sure the loop stops when you expect it to. If you want your program to end when the user enters a certain input value, run the program and enter that value. If the program doesn't end, scrutinize the way your program handles the value that should cause the loop to exit.

Using a while loop with lists and dictionaries

So far, we've worked with only one piece of user information at a time. We received the user's input and then printed the input or a response to it. The next time through the while loop, we'd receive another input value and respond to that. But to keep track of many users and pieces of information, we'll need to use lists and dictionaries with our while loops.

A for loop is effective for looping through a list, but you shouldn't modify a list inside a for loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a while loop. Using while loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

Moving Items from One List to Another

Consider a list of newly registered but unverified users of a website. After we verify these users; how can we move them to a separate list of confirmed users? One way would be to use a while loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users. Here's what that code might look like:

```
  # Start with users that need to be verified,
  # and an empty list to hold confirmed users.
1 unconfirmed_users = ['alice', 'brian', 'candace']
  confirmed_users = []

  # Verify each user until there are no more unconfirmed users.
  # Move each verified user into the list of confirmed users.
2 while unconfirmed_users:
3     current_user = unconfirmed_users.pop()

      print("Verifying user: " + current_user.title())
4     confirmed_users.append(current_user)
  # Display all confirmed users.
  print("\nThe following users have been confirmed:")
  for confirmed_user in confirmed_users:
     print(confirmed_user.title())
```

We begin with a list of unconfirmed users at `1` (Alice, Brian, and Candace) and an empty list to hold confirmed users. The while loop at `2` runs as long as the list `unconfirmed_users` is not empty. Within this loop, the `pop()` function at `3` removes unverified users one at a time from the end of `unconfirmed_users`. Here, because Candace is last in the `unconfirmed_users` list, her name will be the first to be removed, stored in `current_user`, and added to the `confirmed_users` list at `4`. Next is Brian, then Alice.

We simulate confirming each user by printing a verification message and then adding them to the list of confirmed users. As the list of unconfirmed users shrinks, the list of confirmed users grows. When the list of unconfirmed users is empty, the loop stops and the list of confirmed users is printed:

```
Verifying user: Candace
      Verifying user: Brian
      Verifying user: Alice

      The following users have been confirmed:
      Candace
      Brian
      Alice
```

Removing All Instances of Specific Values from a List

We used `remove()` to remove a specific value from a list. The `remove()` function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

Say you have a list of pets with the value `'cat'` repeated several times. To remove all instances of that value, you can run a while loop until 'cat' is no longer in the list, as shown here:

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
            pets.remove('cat')

print(pets)
```

We start with a list containing multiple instances of `'cat'`. After printing the list, Python enters the while loop because it finds the value `'cat'` in the list at least once. Once inside the loop, Python removes the first instance of `'cat'`, returns to the while line, and then re-enters the loop when it finds that `'cat'` is still in the list. It removes each instance of `'cat'` until the value is no longer in the list, at which point Python exits the loop and prints the list again:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Filling a Dictionary with User Input

You can prompt for as much input as you need in each pass-through a while loop. Let's make a polling program in which each pass through the loop prompts for the participant's name and response. We'll store the data we gather in a dictionary, because we want to connect each response with a particular user:

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True
```

```
  while polling_active:
      # Prompt for the person's name and response.
1      name = input("\nWhat is your name? ")
       response = input("Which mountain would you like to climb
     someday? ")

#      Store the response in the dictionary:
2      responses[name] = response

#      Find out if anyone else is going to take the poll.
3      repeat = input("Would you like to let another person respond?
      (yes/ no) ")
       if repeat == 'no':
             polling_active = False

# Polling is complete. Show the results.
   print("\n--- Poll Results ---")
4  for name, response in responses.items():
       print(name + " would like to climb " + response + ".")
```

The program first defines an empty dictionary (`responses`) and sets a flag (`polling_active`) to indicate that polling is active. As long as `polling_active` is True, Python will run the code in the while loop.

Within the loop, the user is prompted to enter their username and a mountain they'd like to climb 1. That information is stored in the responses dictionary 2, and the user is asked whether or not to keep the poll running 3. If they enter yes, the program enters the while loop again. If they enter no, the `polling_active` flag is set to False, the while loop stops running, and the final code block at 4 displays the results of the poll.
If you run this program and enter sample responses, you should see output like this:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

Python Programming
Module 8: Functions

Learning objectives

1. How to write functions and to pass arguments so that your functions have access to the information they need to do their work.
2. How to use positional and keyword arguments, and how to accept an arbitrary number of arguments.
3. You saw functions that display output and functions that return values.
4. How to use functions with lists, dictionaries, if statements, and while loops.
5. You saw how to store your functions in separate files called modules, so your program files will be simpler and easier to understand.
6. Learned to style your functions so your programs will continue to be well-structured and as easy as possible for you and others to read.

Functions

Defining a Function

Here's a simple function named greet_user() that prints a greeting:

```
1 def greet_user():
 2   """Display a simple greeting."""
 3   print("Hello!")
 4   greet_user()
```

This example shows the simplest structure of a function. The line at 1 uses the keyword `def` to inform Python that you're defining a function. This is the *function definition*, which tells Python the name of the function and, if applicable, what kind of information the function needs to do its job. The parentheses hold that information. In this case, the name of the function is `greet_user()`, and it needs no information to do its job, so its parentheses are empty. (Even so, the parentheses are required.) Finally, the definition ends in a colon.

Any indented lines that follow `def greet_user():` make up the body of the function. The text at 2 is a comment called a docstring, which describes what the function does. Docstrings are enclosed in triple quotes, which Python looks for when it generates documentation for the functions in your programs.

The line `print("Hello!")` is the only line of actual code in the body of this function, so `greet_user()` has just one job: `print("Hello!")`.

When you want to use this function, you call it. A function call tells Python to execute the code in the function. To call a function, you write the name of the function, followed by any necessary information in parentheses, as shown at 4. Because no information is needed here, calling our function is as simple as entering `greet_user()`. As expected, it prints `Hello!`.

Passing Information to a Function

The function `greet_user()` can not only tell the user `Hello!` but also greet them by name. For the function to do this, you enter username in the parentheses of the function's definition at `def greet_user()`. By add-ing `username` here you allow the function to accept any value of username you specify. The function now expects you to provide a value for username each time you call it. When you call `greet_user()`, you can pass it a name, such as `'jesse'`, inside the parentheses:

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username.title() + "!")

greet_user('jesse')
```

Arguments and Parameters

In the preceding `greet_user()` function, we defined `greet_user()` to require a value for the variable `username`. Once we called the function and gave it the information (a person's name), it printed the right greeting.

The variable username in the definition of `greet_user()` is an example of a parameter, a piece of information the function needs to do its job. The value `'jesse'` in `greet_user('jesse')` is an example of an argument. An argument is a piece of information that is passed from a function call to a function.

Passing Arguments

Because a function definition can have multiple parameters, a function call may need multiple arguments. You can pass arguments to your functions in a number of ways. You can use `positional arguments`, which need to be in the same order the parameters were written; `keyword arguments`, where each argument consists of a variable name and a value; and lists and dictionaries of values. Let's look at each of these in turn.

Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called positional arguments.

To see how this works, consider a function that displays information about pets. The function tells us what kind of animal each pet is and the pet's name, as shown here:

```
1 def describe_pet(animal_type, pet_name):
      """Display information about a pet.""" print("\nI have a "
      + animal_type + ".")

      print("My " + animal_type + "'s name is " +
      pet_name.title() + ".")

2 describe_pet('hamster', 'harry')
```

The definition shows that this function needs a type of animal and the animal's name `1`. When we call `describe_pet()`, we need to provide an animal type and a name, in that order. For example, in the function call, the argument `'hamster'` is stored in the parameter `animal_type` and the argument `'harry'` is stored in the parameter `pet_name 2`. In the function body, these two parameters are used to display information about the pet being described.

The output describes a hamster named Harry:

```
I have a hamster.

My hamster's name is Harry.
```

Multiple Function Calls

You can call a function as many times as needed. Describing a second, dif-ferent pet requires just one more call to `describe_pet()`:

```
describe_pet('dog', 'willie')
```

In this second function call, we pass `describe_pet()` the arguments `'dog'` and `'willie'`. As with the previous set of arguments we used, Python matches `'dog'` with the parameter `animal_type` and `'willie'` with the parameter `pet_name`.

As before, the function does its job, but this time it prints values for a dog named Willie. Now we have a hamster named Harry and a dog named Willie:

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Calling a function multiple times is a very efficient way to work. The code describing a pet is written once in the function. Then, anytime you want to describe a new pet, you call the function with the new pet's information. Even if the code for describing a pet were to expand to ten lines, you could still describe a new pet in just one line by calling the function again.

You can use as many positional arguments as you need in your functions. Python works through the arguments you provide when calling the function and matches each one with the corresponding parameter in the function's definition.

Order Matters in Positional Arguments

You can get unexpected results if you mix up the order of the arguments in a function call when using positional arguments.

```
describe_pet('harry', 'hamster')
```

In this function call we list the name first and the type of animal second. Because the argument `'harry'` is listed first this time, that value is stored in the parameter `animal_type`. Likewise, `'hamster'` is stored in `pet_name`. Now we have a `'harry'` named `'Hamster'`:

```
I have a harry.
My harry's name is Hamster.
```

Keyword Arguments

A `keyword argument` is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion (you won't end up with a harry named Hamster). Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

Let's rewrite `pets.py` using keyword arguments to call `describe_pet()`:

```
describe_pet(animal_type='hamster', pet_name='harry')
```

The function `describe_pet()` hasn't changed. But when we call the function, we explicitly tell Python which parameter each argument should be matched with. When Python reads the function call, it knows to store the argument `'hamster'` in the parameter `animal_type` and the argument `'harry'` in `pet_name`. The output correctly shows that we have a hamster named Harry.

The order of keyword arguments doesn't matter because Python knows where each value should go. The following two function calls are equivalent.

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

Default Values

When writing a function, you can define a `default` value for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. So when you define a default value for a parameter, you can exclude the corresponding argument you'd usually write in the function call. Using default values can simplify your function calls and clarify the ways in which your functions are typically used.

For example, if you notice that most of the calls to `describe_pet()` are being used to describe dogs, you can set the default value of `animal_type` to `'dog'`. Now anyone calling `describe_pet()` for a dog can omit that information:

```
def describe_pet(pet_name, animal_type='dog'):
     """Display information about a pet."""
     print("\nI have a " + animal_type + ".")
     print("My " + animal_type + "'s name is " +
     pet_name.title() + ".")

describe_pet(pet_name='willie')
```

We changed the definition of `describe_pet()` to include a default value, `'dog'`, for `animal_type`. Now when the function is called with no `animal_type` specified, Python knows to use the value `'dog'` for this parameter:

```
I have a dog.
My dog's name is Willie.
```

Note that the order of the parameters in the function definition had to be changed. Because the default value makes it unnecessary to specify a type of animal as an argument, the only argument left in the function call is the pet's name. Python still interprets this as a positional argument, so if the function is called with just a pet's name, that argument will match up with the first parameter listed in the function's definition. This is the reason the first parameter needs to be `pet_name`.

The simplest way to use this function now is to provide just a dog's name in the function call:

```
describe_pet('willie')
```

This function call would have the same output as the previous example. The only argument provided is `'willie'`, so it is matched up with the first parameter in the definition, `pet_name`. Because no argument is provided for `animal_type`, Python uses the default value `'dog'`.

To describe an animal other than a dog, you could use a function call like this:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Because an explicit argument for `animal_type` is provided, Python will ignore the parameter's default value.

When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.

Equivalent Function Calls

Because positional arguments, keyword arguments, and default values can all be used together, often you'll have several equivalent ways to call a function. Consider the following definition for `describe_pets()` with one default value provided:

```
def describe_pet(pet_name, animal_type='dog'):
```

With this definition, an argument always needs to be provided for pet_name, and this value can be provided using the positional or keyword format. If the animal being described is not a dog, an argument for `animal_type` must be included in the call, and this argument can also be specified using the positional or keyword format.

All of the following calls would work for this function:

```
# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# A hamster named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Each of these function calls would have the same output as the previous examples.

It doesn't really matter which calling style you use. As long as your function calls produce the output you want, just use the style you find easiest to understand.

Avoiding Argument Errors

When you start to use functions, don't be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work. For example, here's what happens if we try to call `describe_pet()` with no arguments, Python recognizes that some information is missing from the function call, and the traceback tells us that:

```
Traceback (most recent call last):
1 File "pets.py", line 6, in <module>
2 describe_pet()
3 TypeError: describe_pet() missing 2 required positional
arguments: 'animal_
```

```
    type' and 'pet_name'
```

At 1 the traceback tells us the location of the problem, allowing us to look back and see that something went wrong in our function call. At 2 the offending function call is written out for us to see. At 3 the traceback tells us the call is missing two arguments and reports the names of the missing arguments. If this function were in a separate file, we could probably rewrite the call correctly without having to open that file and read the function code.

If you provide too many arguments, you should get a similar traceback that can help you correctly match your function call to the function definition.

Return Values

A function doesn't always have to display its output directly. Instead, it can process some data and then return a value or set of values. The value the function returns is called a *return value*. The `return` statement takes a value from inside a function and sends it back to the line that called the function. Return values allow you to move much of your program's grunt work into functions, which can simplify the body of your program.

Returning a Simple Value

Let's look at a function that takes a first and last name, and returns a neatly formatted full name:

```
 def get_formatted_name(first_name, last_name):
     """Return a full name, neatly formatted."""
     1 full_name = first_name + ' ' + last_name
     2 return full_name.title()

3 musician = get_formatted_name('jimi', 'hendrix')
4 print(musician)
```

The definition of `get_formatted_name()` takes as parameters a first and last name 1. The function combines these two names, adds a space between them, and stores the result in `full_name` 2. The value of `full_name` is converted to title case, and then returned to the calling line at 3

When you call a function that returns a value, you need to provide a variable where the return value can be stored. In this case, the returned value is stored in the variable `musician` at 4. The output shows a neatly formatted name made up of the parts of a person's name:

```
Jimi Hendrix
```

This might seem like a lot of work to get a neatly formatted name when we could have just written:

```
print("Jimi Hendrix")
```

But when you consider working with a large program that needs to store many first and last names separately, functions like `get_formatted_name()` become very useful. You store first and last names separately and then call this function whenever you want to display a full name.

Making an Argument Optional

Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

For example, say we want to expand `get_formatted_name()` to handle middle names as well. A first attempt to include middle names might look like this:

```
def get_formatted_name(first_name, middle_name, last_name):
     """Return a full name, neatly formatted."""
     full_name = first_name + ' ' + middle_name + ' ' +
last_name
     return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

This function works when given a first, middle, and last name. The function takes in all three parts of a name and then builds a string out of them. The function adds spaces where appropriate and converts the full name to title case:

```
John Lee Hooker
```

But middle names aren't always needed, and this function as written would not work if you tried to call it with only a first name and a last name. To make the middle name optional, we can give the `middle_name` argument an empty default value and ignore the argument unless the user provides a value. To make `get_formatted_name()` work without a middle name, we set the default value of `middle_name` to an empty string and move it to the end of the list of parameters:

```
def get_formatted_name(first_name, last_name, middle_name=''):
     """Return a full name, neatly formatted."""
     1 if middle_name:
     full_name = first_name + ' ' + middle_name + ' ' +
last_name
     2 else:
     full_name = first_name + ' ' + last_name
     return full_name.title()

4 musician = get_formatted_name('jimi', 'hendrix')
print(musician)
3 musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

In this example, the name is built from three possible parts. Because there's always a first and last name, these parameters are listed first in the function's definition. The middle name is optional, so it's listed last in the definition, and its default value is an empty string **1**.

In the body of the function, we check to see if a middle name has been provided. Python interprets non-empty strings as True, so if `middle_name` evaluates to `True` if a middle name argument is in the function call **2**. If a middle name is provided, the first, middle, and last names are combined to form a full name. This name is then changed to title case and returned to the function call line where it's stored in the variable musician and printed. If no middle name is provided, the empty string fails the if test and the else block runs **3**. The full name is made with just a first and last name, and the formatted name is returned to the calling line where it's stored in `musician` and printed.

Calling this function with a first and last name is straightforward. If we're using a middle name, however, we have to make sure the middle name is the last argument passed so Python will match up the positional arguments correctly **4**.

This modified version of our function works for people with just a first and last name, and it works for people who have a middle name as well:

```
Jimi Hendrix
John Lee Hooker
```

Optional values allow functions to handle a wide range of use cases while letting function calls remain as simple as possible.

Returning a Dictionary

A function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries. For example, the following function takes in parts of a name and returns a dictionary representing a person:

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
    1 person = {'first': first_name, 'last': last_name}
    2 return person

musician = build_person('jimi', 'hendrix')
3 print(musician)
```

The function `build_person()` takes in a first and last name, and packs these values into a dictionary at **1**. The value of `first_name` is stored with the key `'first'`, and the value of `last_name` is stored with the key `'last'`. The entire dictionary representing the person is returned at **2**. The return value is printed at **3** with the original two pieces of textual information now stored in a dictionary:

```
{'first': 'jimi', 'last': 'hendrix'}
```

This function takes in simple textual information and puts it into a more meaningful data structure that lets you work with the information beyond just printing it. The strings 'jimi' and 'hendrix' are now labeled as a first name and last name.

Using a Function with a while Loop

You can use functions with all the Python structures you've learned about so far. For example, let's use the `get_formatted_name()` function with a while loop to greet users more formally. Here's a first attempt at greeting people using their first and last names:

```
# This is an infinite loop!
while True:
```

```
1 print("\nPlease tell me your name:")
  f_name = input("First name: ")
  l_name = input("Last name: ")

  formatted_name = get_formatted_name(f_name, l_name)
  print("\nHello, " + formatted_name + "!")
```

For this example, we use a simple version of `get_formatted_name()` that doesn't involve middle names. The while loop asks the user to enter their name, and we prompt for their first and last name separately **1**.

But there's one problem with this `while` loop: We haven't defined a quit condition. Where do you put a quit condition when you ask for a series of inputs? We want the user to be able to quit as easily as possible, so each prompt should offer a way to quit. The `break` statement offers a straightforward way to exit the loop at either prompt:

```
def get_formatted_name(first_name, last_name):
     """Return a full name, neatly formatted."""
     full_name = first_name + ' ' + last_name
     return full_name.title()
     while True:
          print("\nPlease tell me your name:")
          print("(enter 'q' at any time to quit)")

          f_name = input("First name: ")
          if f_name == 'q':
               break

          l_name = input("Last name: ")
          if l_name == 'q':
               break

 formatted_name = get_formatted_name(f_name, l_name)
 print("\nHello, " + formatted_name + "!")
```

We add a message that informs the user how to quit, and then we break out of the loop if the user enters the quit value at either prompt. Now the program will continue greeting people until someone enters 'q' for either `name`:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes

Hello, Eric Matthes!

Please tell me your name:
```

```
(enter 'q' at any time to quit)
First name: q
```

Passing a List

You'll often find it useful to pass a list to a function, whether it's a list of names, numbers, or more complex objects, such as dictionaries. When you pass a list to a function, the function gets direct access to the contents of the list. Let's use functions to make working with lists more efficient.

Say we have a list of users and want to print a greeting to each. The following example sends a list of names to a function called `greet_users()`, which greets each person in the list individually:

```
def greet_users(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = "Hello, " + name.title() + "!"
        print(msg)
1 usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

We define `greet_users()` so it expects a list of names, which it stores in the parameter names. The function loops through the list it receives and prints a greeting to each user. At **1** we define a list of users and then pass the list usernames to `greet_users()` in our function call:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

This is the output we wanted. Every user sees a personalized greeting, and you can call the function any time you want to greet a specific set of `users`.

Modifying a List in a Function

When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work efficiently even when you're dealing with large amounts of data.

Consider a company that creates 3D printed models of designs that users submit. Designs that need to be printed are stored in a list, and after being printed they're moved to a separate list. The following code does this without using functions:

```
# Start with some designs that need to be printed.
unprinted_designs = ['iphone case', 'robot pendant',
'dodecahedron']
completed_models = []
# Simulate printing each design, until none are left.
# Move each design to completed_models after printing.
```

```
while unprinted_designs:
     current_design = unprinted_designs.pop()

# Simulate creating a 3D print from the design.
print("Printing model: " + current_design)
completed_models.append(current_design)

# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
     print(completed_model)
```

This program starts with a list of designs that need to be printed and an empty list called `completed_models` that each design will be moved to after it has been printed. As long as designs remain in `unprinted_designs`, the `while` loop simulates printing each design by removing a design from the end of the list, storing it in `current_design`, and displaying a message that the current design is being printed. It then adds the design to the list of completed models. When the loop is finished running, a list of the designs that have been printed is displayed:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case
The following models have been printed:
dodecahedron
robot pendant
iphone case
```

We can reorganize this code by writing two functions, each of which does one specific job. Most of the code won't change; we're just making it more efficient. The first function will handle printing the designs, and the second will summarize the prints that have been made:

```
 1 def print_models(unprinted_designs, completed_models):
     """
      Simulate printing each design, until none are left.
      Move each design to completed_models after printing.
      """
      while unprinted_designs:
          current_design = unprinted_designs.pop()
          #Simulate creating a 3D print from the design.
          print("Printing model: " + current_design)
          completed_models.append(current_design)

2 def show_completed_models(completed_models):
     """Show all the models that were printed."""
     print("\nThe following models have been printed:")
     for completed_model in completed_models:
            print(completed_model)

unprinted_designs = ['iphone case', 'robot pendant',
'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

At **1** we define the function `print_models()` with two parameters: a `list` of designs that need to be printed and a `list` of completed models. Given these two lists, the function simulates printing each design by emptying the list of unprinted designs and filling up the list of completed models. At **2** we define the function `show_completed_models()` with one parameter: the list of completed models. Given this list, `show_completed_models()` displays the name of each model that was printed.

This program has the same output as the version without functions, but the code is much more organized. The code that does most of the work has been moved to two separate functions, which makes the main part of the program easier to understand. Look at the body of the program to see how much easier it is to understand what this program is doing:

```
unprinted_designs = ['iphone case', 'robot pendant',
'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

We set up a list of unprinted designs and an empty list that will hold the completed models. Then, because we've already defined our two functions, all we have to do is call them and pass them the right arguments. We call `print_models()` and pass it the two lists it needs; as expected, `print_models()` simulates printing the designs. Then we call `show_completed_models()` and pass it the list of completed models so it can report the models that have been printed.

Preventing a Function from Modifying a List

Sometimes you'll want to prevent a function from modifying a list. For example, say that you start with a list of unprinted designs and write a 150 Chapter 8 function to move them to a list of completed models, as in the previous example. You may decide that even though you've printed all the designs, you want to keep the original list of unprinted designs for your records. But because you moved all the design names out of `unprinted_designs`, the list is now empty, and the empty list is the only version you have; the original is gone. In this case, you can address this issue by passing the function a copy of the list, not the original. Any changes the function makes to the list will affect only the copy, leaving the original list intact.

You can send a copy of a list to a function like this:

```
function_name(list_name[:])
```

The slice notation `[:]` makes a copy of the list to send to the function. If we didn't want to empty the list of unprinted designs in `print_models.py`, we could call `print_models()` like this:

```
print_models(unprinted_designs[:], completed_models)
```

The function `print_models()` can do its work because it still receives the names of all unprinted designs. But this time it uses a copy of the original unprinted designs list, not the actual `unprinted_designs` list. The list `completed_models` will fill up with the names of printed models like it did before, but the original list of unprinted designs will be unaffected by the function.

Passing an Arbitrary Number of Arguments

Sometimes you won't know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, `*toppings`, but this parameter collects as many arguments as the calling line provides:

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the parameter name `*toppings` tells Python to make an empty tuple called `toppings` and pack whatever values it receives into this tuple. The `print` statement in the function body produces output showing that Python can handle a function call with one value and a call with three values. It treats the different calls similarly. Note that Python packs the arguments into a tuple, even if the function receives only one value:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Now we can replace the print statement with a loop that runs through the list of toppings and describes the pizza being ordered:

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The function responds appropriately, whether it receives one value or three values:

```
Making a pizza with the following toppings:
- pepperoni
Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Mixing Positional and Arbitrary Arguments

If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

For example, if the function needs to take in a size for the pizza, that parameter must come before the parameter `*toppings`:

```
def make_pizza(size, *toppings):
     """Summarize the pizza we are about to make."""
     print("\nMaking a " + str(size) +
     "-inch pizza with the following toppings:")
     for topping in toppings:
          print("- " + topping)
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

In the function definition, Python stores the first value it receives in the parameter `size`. All other values that come after are stored in the tuple `toppings`. The function calls include an argument for the size first, followed by as many toppings as needed.

Now each pizza has a size and a number of toppings, and each piece of information is printed in the proper place, showing size first and toppings after:

```
Making a 16-inch pizza with the following toppings:
- pepperoni
Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Using Arbitrary Keyword Arguments

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive. The function `build_profile()` in the following example always takes in a first and last name, but it accepts an arbitrary number of keyword arguments as well:

```
def build_profile(first, last, **user_info):
        """Build a dictionary containing everything we know about
        a user."""
        profile = {}
        1 profile['first_name'] = first
        profile['last_name'] = last
        2 for key, value in user_info.items():
             profile[key] = value
        return profile
user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want. The double asterisks before the parameter `**user_info` cause Python to create an empty dictionary called `user_info` and pack whatever name-value pairs it receives into this dictionary. Within the function, you can access the namevalue pairs in `user_info` just as you would for any dictionary.

In the body of `build_profile()`, we make an empty dictionary called profile to hold the user's `profile`. At **1** we add the first and last names to this dictionary because we'll always receive these two pieces of information from the user. At **2** we loop through the additional key-value pairs in the dictionary `user_info` and add each pair to the `profile` dictionary. Finally, we return the profile dictionary to the function call line.

We call `build_profile()`, passing it the first name `'albert'`, the last name `'einstein'`, and the two key-value pairs `location='princeton'` and `field='physics'`. We store the returned profile in `user_profile` and print `user_profile`:

```
{'first_name': 'albert', 'last_name': 'einstein',
'location': 'princeton', 'field': 'physics'}
```

Storing Your Functions in Modules

One advantage of functions is the way they separate blocks of code from your main program. By using descriptive names for your functions, your main program will be much easier to follow. You can go a step further by storing your functions in a separate file called a *module* and then *importing* that *module* into your main program. An *import* statement tells Python to make the code in a module available in the currently running program file.

Storing your functions in a separate file allows you to hide the details of your program's code and focus on its higher-level logic. It also allows you to reuse functions in many different programs. When you store your functions in separate files, you can share those files with other programmers without having to share your entire program. Knowing how to import functions also allows you to use libraries of functions that other programmers have written.

There are several ways to import a module, and I'll show you each of these briefly.

Importing an Entire Module

To start importing functions, we first need to create a module. A *module* is a file ending in *.py* that contains the code you want to import into your program. Let's make a module that contains the function `make_pizza()`. To make this module, we'll remove everything from the file `pizza.py` except the function `make_pizza()`:

```
def make_pizza(size, *toppings):
     """Summarize the pizza we are about to make."""
     print("\nMaking a " + str(size) +
     "-inch pizza with the following toppings:")
     for topping in toppings:
          print("- " + topping)
```

Now we'll make a separate file called `making_pizzas.py` in the same directory as `pizza.py`. This file imports the module we just created and then makes two calls to `make_pizza()`:

```
import pizza
1 pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra
cheese')
```

When Python reads this file, the line `import pizza` tells Python to open the file *pizza.py* and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes as the program runs. All you need to know is that any function defined in *pizza.py* will now be available in *making_pizzas.py*.

To call a function from an imported module, enter the name of the module you imported, `pizza`, followed by the name of the function, `make_pizza()`, separated by a dot **1**. This code produces the same output as the original program that didn't import a module:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This first approach to importing, in which you simply write `import` followed by the name of the module, makes every function from the module available in your program. If you use this kind of `import` statement to import an entire module named *module_name.py*, each function in the module is available through the following syntax:

```
module_name.function_name()
```

Importing Specific Functions

You can also import a specific function from a module. Here's the general syntax for this approach:

```
from module_name import function_name
```

You can import as many functions as you want from a module by separating each function's name with a comma:

```
from module_name import function_0, function_1, function_2
```

The *making_pizzas.py* example would look like this if we want to import just the function we're going to use:

```
from pizza import make_pizza
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

With this syntax, you don't need to use the dot notation when you call a function. Because we've explicitly imported the function `make_pizza()` in the `import` statement, we can call it by name when we use the function.

Using as to Give a Function an Alias

If the name of a function you're importing might conflict with an existing name in your program or if the function name is long, you can use a short, unique alias—an alternate name similar to a nickname for the function. You'll give the function this special nickname when you import the function.

Here we give the function `make_pizza()` an alias, `mp()`, by importing `make_pizza` as `mp`. The `as` keyword renames a function using the alias you provide:

```
from pizza import make_pizza as mp
mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The import statement shown here renames the function `make_pizza()` to `mp()` in this program. Any time we want to call `make_pizza()` we can simply write `mp()` instead, and Python will run the code in `make_pizza()` while avoiding any confusion with another `make_pizza()` function you might have written in this program file.

The general syntax for providing an alias is:

```
from module_name import function_name as fn
```

Using as to Give a Module an Alias

You can also provide an alias for a module name. Giving a module a short alias, like `p` for `pizza`, allows you to call the module's functions more quickly. Calling `p.make_pizza()` is more concise than calling `pizza.make_pizza()`:

```
import pizza as p
p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The module `pizza` is given the alias `p` in the import statement, but all of the module's functions retain their original names. Calling the functions by writing `p.make_pizza()` is not only more concise than writing `pizza.make_pizza()`, but also redirects your attention from the module name and allows you to focus on the descriptive names of its functions. These function names, which clearly tell you what each function does, are more important to the readability of your code than using the full module name.

The general syntax for this approach is:

```
import module_name as mn
```

Importing All Functions in a Module

You can tell Python to import every function in a module by using the asterisk `(*)` operator:

```
from pizza import *
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the `import` statement tells Python to copy every function from the module `pizza` into this program file. Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get some unexpected results. Python may see several functions or variables with the same name, and instead of importing all the functions separately, it will overwrite the functions.

The best approach is to import the function or functions you want, or import the entire module and use the dot notation. This leads to clear code that's easy to read and understand. I include this section so you'll recognize import statements like the following when you see them in other people's code:

```
from module_name import *
```

Styling Functions

You need to keep a few details in mind when you're styling functions. Functions should have descriptive names, and these names should use lowercase letters and underscores. Descriptive names help you and others understand what your code is trying to do. Module names should use these conventions as well.

Every function should have a comment that explains concisely what the function does. This comment should appear immediately after the function definition and use the docstring format. In a well-documented function, other programmers can use the function by reading only the description in the docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

If you specify a default value for a parameter, no spaces should be used on either side of the equal sign:

```
def function_name(parameter_0, parameter_1='default value')
```

The same convention should be used for keyword arguments in function calls:

```
function_name(value_0, parameter_1='value')
```

PEP 8 (`https://www.python.org/dev/peps/pep-0008/`) recommends that you limit lines of code to 79 characters so every line is visible in a reasonably sized editor window. If a set of parameters causes a function's definition to be longer than 79 characters, press enter after the opening parenthesis on the definition line. On the next line, press tab twice to separate the list of arguments from the body of the function, which will only be indented one level.

Most editors automatically line up any additional lines of parameters to match the indentation you have established on the first line:

```
def function_name(
 parameter_0, parameter_1, parameter_2,
 parameter_3, parameter_4, parameter_5):
 function body…
```

If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.

All `import` statements should be written at the beginning of a file. The only exception is if you use comments at the beginning of your file to describe the overall program.

Python Programming
Module 9: Classes

Learning objectives

1. How to write your own classes.
2. How to store information in a class using attributes and how to write methods that give your classes the behavior they need.
3. How to modify the attributes of an instance directly and through methods.
4. How inheritance can simplify the creation of classes that are related to each other.
5. Using instances of one class as attributes in another class to keep each class simple.

Classes

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. here is small introduction of Object-Oriented Programming (OOP) to bring you at speed.

Overview of OOP Terminology

Class − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

Data member − A class variable or instance variable that holds data associated with a class and its objects.

Function overloading − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

Instance variable − A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance − The transfer of the characteristics of a class to other classes that are derived from it.

Instance − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation − The creation of an instance of a class.

Method − A special kind of function that is defined in a class definition.

Object − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading − The assignment of more than one function to a particular operator.

Creating Classes

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows −

```
class ClassName:
   '''Optional class documentation string'''
   Class_suite
```

- The class has a documentation string, which can be accessed via `ClassName.__doc__`.

- The `class_suite` consists of all the component statements defining class members, data attributes and functions.

Following is an example of a simple Python class –

```
class Employee:
   '''Common base class for all employees'''
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable `empCount` is a class variable whose value is shared among all instances of this class. This can be accessed as `Employee.empCount` from inside the class or outside the class.

- The first method `__init__()` is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

- You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects
To create instances of a class, you call the class using class name and pass in whatever arguments its `__init__` method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Now, putting all the concepts together –

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result −

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time −

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions −

- The `getattr(obj, name[, default])` − to access the attribute of object.
- The `hasattr(obj, name)` − to check if an attribute exists or not.
- The `setattr(obj, name, value)` − to set an attribute. If attribute does not exist, then it would be created.
- The `delattr(obj, name)` − to delete an attribute.

```
hasattr(emp1,  'age')       #  Returns  true  if  'age'  attribute
exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

`__dict__`: Dictionary containing the class's namespace.

`__doc__`: Class documentation string or none, if undefined.

`__name__`: Class name.

`__module__`: Module name in which the class is defined. This attribute is "__main__" in interactive mode.

`__bases__`: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes −

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result −

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection. Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes. An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method `__del__()`, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

This `__del__()` destructor prints the class name of an instance that is about to be destroyed −

```
class Point:
   def __init__( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

When the above code is executed, it produces following result −

```
3083401324 3083401324 3083401324
Point destroyed
```

Ideally, you should define your classes in separate file, then you should import them in your main program file using import statement.

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name. The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   Class_suite
```

For example consider the following example:

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result −

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....

class B:         # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```

You can use `issubclass()` or `isinstance()` functions to check a relationships of two classes and instances.

- The `issubclass(sub, sup)` boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.
- The `isinstance(obj, Class)` boolean function returns true if `obj` is an instance of class `Class` or is an instance of a subclass of `Class`

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result −

```
Calling child method
```

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you. You could, however, define the `__add__` method in your class to perform vector addition and then the plus operator would behave as per expectation –

```
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not directly visible to outsiders.

```
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

When the above code is executed, it produces the following result –

```
1
2
Traceback (most recent call last):
   File "test.py", line 12, in <module>
```

```
      print counter.__secretCount
AttributeError:   JustCounter   instance   has   no   attribute
'__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as `object._className__attrName`. If you would replace your last line as following, then it works for you −

```
.........................
print counter._JustCounter__secretCount
```

When the above code is executed, it produces the following result −

```
1
2
2
```