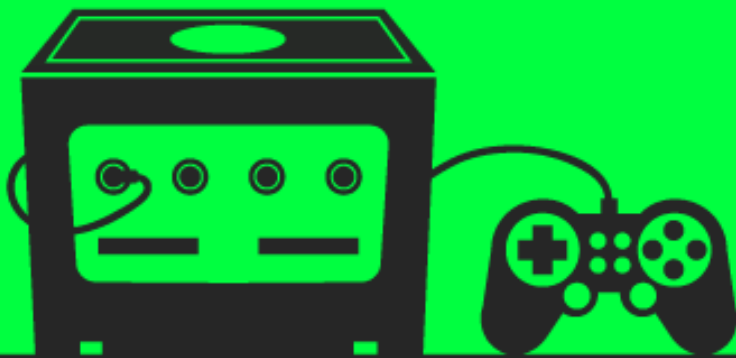




Python

Succinctly

by Jason Cannon

Python Succinctly

By

Jason Cannon

Foreword by Daniel Jebaraj



Copyright © 2016 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

I mportant licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

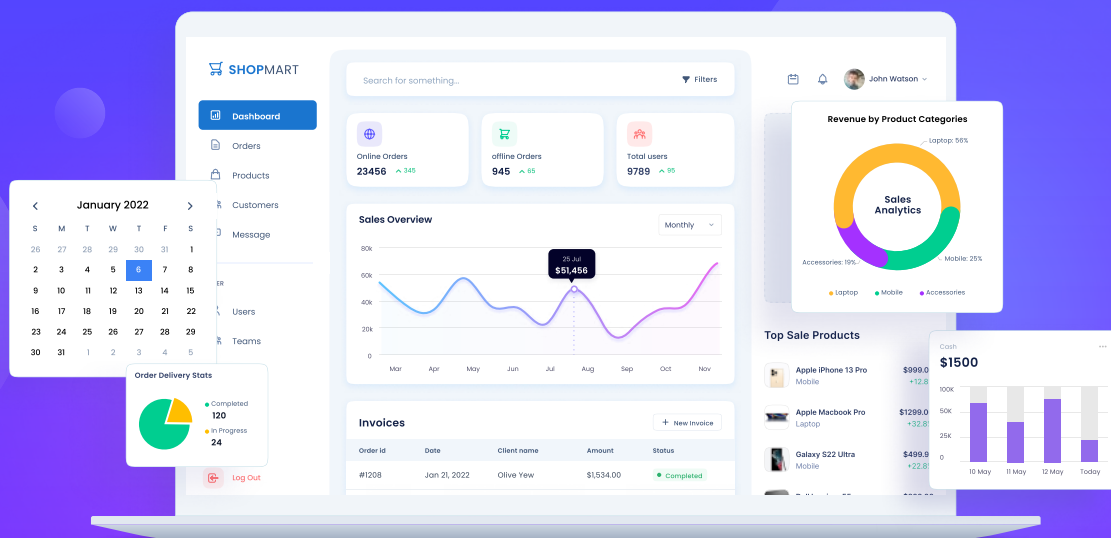
Technical Reviewer: James McCaffrey

Copy Editor: Graham High, content producer, Syncfusion, Inc.

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS



GET YOUR **FREE** .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense



1,700+ components for
mobile, web, and
desktop platforms



Support within 24 hours
on all business days



Uncompromising
quality



Hassle-free licensing



28000+ customers



20+ years in
business

Trusted by the world's leading companies



Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	9
About the Author	11
Introduction.....	12
A Note on the Text.....	12
Getting Started	12
Configuring Your Environment for Python	13
Installing Python	13
Choosing Python 2 or Python 3.....	13
Windows Installation Instructions	13
Mac Installation Instructions	16
Linux Installation Instructions	20
Preparing Your Computer for Python	24
Running Python Programs	24
Creating and Editing Python Source Code	27
Downloading the Source Code Examples.....	29
Review	29
Resources.....	29
Chapter 1 Variables and Strings	31
Variables.....	31
Strings.....	32
Using Quotes within Strings	32
Indexing	33
Built-in Functions	34
The print() Function	34

The len() Function	34
String Methods	36
The lower() String Method	36
The upper() String Method	36
String Concatenation	37
Repeating Strings	38
The str() Function	39
Formatting Strings	40
Getting User Input	43
Review	44
Exercises	45
Animal, Vegetable, Mineral	45
Copy Cat.....	46
Pig Speak	47
Resources.....	48
Chapter 2 Numbers, Math, and Comments	49
Numeric Operations.....	49
Strings and Numbers.....	52
The int() Function	53
The float() Function	53
Comments	54
Review	56
Exercises	56
Calculate the Cost of Cloud Hosting	56
Calculate the Cost of Cloud Hosting, Continued	57
Chapter 3 Booleans and Conditionals.....	60
Comparators	60
Boolean Operators	62

Conditionals.....	65
Review.....	68
Exercises.....	69
Walk, Drive, or Fly.....	69
Resources.....	70
Chapter 4 Functions.....	71
Review.....	78
Exercises.....	79
Fill in the Blank Word Game.....	79
Resources.....	81
Chapter 5 Lists.....	82
Adding Items to a List.....	83
Slices.....	85
String Slices.....	86
Finding an Item in a List.....	87
Exceptions.....	87
Looping through a List.....	89
Sorting a List.....	90
List Concatenation.....	91
Ranges.....	92
Review.....	94
Exercises.....	95
Grocery List.....	95
Resources.....	96
Chapter 6 Dictionaries.....	98
Adding Items to a Dictionary.....	99
Removing Items from a Dictionary.....	99

Finding a Key in a Dictionary.....	101
Finding a Value in a Dictionary.....	102
Looping through a Dictionary.....	103
Nesting Dictionaries.....	104
Review	106
Exercises	106
Interesting Facts	106
Resources.....	107
Chapter 7 Tuples.....	108
Switching between Tuples and Lists	110
Looping through a Tuple.....	111
Tuple Assignment.....	112
Review	114
Exercises	115
ZIP Codes.....	115
Resources.....	116
Chapter 8 File I/O	117
File Position	118
Closing a File	119
Automatically Closing a File	121
Reading a File One Line at a Time.....	121
File Modes	123
Writing to a File.....	124
Binary Files	126
Exceptions	126
Review	127
Exercises	128
Line Numbers	128

Alphabetize.....	128
Resources.....	130
Chapter 9 Modules.....	131
Modules	131
Peeking Inside a Module	133
The Module Search Path.....	133
The Python Standard Library.....	136
Creating Your Own Modules	137
Using main.....	139
Review	140
Exercises	140
Pig Speak, Redux.....	140
Resources.....	143
Conclusion	144
Appendix.....	146
Appendix A: Trademarks.....	146

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Jason Cannon started his career as a Unix and Linux System Engineer in 1999. Since that time he has utilized his Linux skills at companies such as Xerox, UPS, Hewlett-Packard, and Amazon.com. Additionally, he has acted as a technical consultant and independent contractor for small businesses as well as Fortune 500 companies.

Jason has professional experience with CentOS, RedHat Enterprise Linux, SUSE Linux Enterprise Server, and Ubuntu. He has used several Linux distributions on personal projects including Debian, Slackware, CrunchBang, and others. In addition to Linux, Jason has experience supporting proprietary Unix operating systems including AIX, HP-UX, and Solaris.

He enjoys teaching others how to use and exploit the power of open source software. Jason is the author of [Command Line Kung Fu](#), [Shell Scripting](#), and [Linux for Beginners](#). He is also the founder of the [Linux Training Academy](#) where he blogs and teaches online video training courses.

Introduction

A Note on the Text

This e-book is an update to an existing book, *Python Programming for Beginners: An Introduction to the Python Computer Language and Computer Programming*. This e-book includes revised text and is the most up-to-date version available at the time of publication.

Getting Started

Choosing a place to begin when learning a new skill can often be difficult, especially when you are dealing with a broad or complex topic. In many cases, there is so much information available that it can be a real challenge to decide exactly where to start. Even worse, you may finally take the first steps toward learning, only to quickly discover far too many concepts, programming examples, and nuances that aren't fully and thoughtfully explained. This type of experience can be incredibly frustrating, ultimately leaving you with more questions than answers.

Python Succinctly will help you sidestep this frustration. In this book we make no assumptions about your technical background, your knowledge of computer programming, or your general understanding of the Python language. You need no prior knowledge to benefit from reading this book. In these pages you will be guided step-by-step using a logical and systematic approach. While there will be new concepts, code, and jargon introduced, they will be explained in plain language, making it easy for absolutely anyone to understand.

Throughout the book you will be presented with many examples, as well as various Python programs. You can download all of these examples, as well as additional resources, at <https://bitbucket.org/syncfusiontech/python-succinctly>.

Let's get started.

Configuring Your Environment for Python

Installing Python

Choosing Python 2 or Python 3

If you are a Python beginner, or are just getting started on a new project, I highly recommend using Python 3. Released in 2008, Python 3 is the most current incarnation of the program, with the Python 2.x series now considered legacy. Keep in mind however that there are still many Python 2 programs in use today, and you may encounter them from time to time. For the most part though this isn't an issue, as the Python 2.7 release effectively bridges the gap between Python 2 and Python 3. Much of the code written for Python 3 will work on Python 2.7. Unfortunately, it is important to note that the same code will most likely not run unmodified on Python versions 2.6 and lower.

That's why, when it is at all possible, you should try to use the latest version of Python available. If you absolutely must use Python 2, definitely use Python 2.7, as it is compatible with all Python 2 code, and much of Python 3 code. The primary reason that you might opt to choose Python 2 over Python 3 is if your project requires third-party software that is not yet compatible with Python 3.

Windows Installation Instructions

It is important to note that Python does not come installed on the Windows operating system. In order to access the program you will need to download the Python installer from the Python downloads page at <https://www.python.org/downloads>. Click the **Download** link next to the desired version of Python to download the installer. Double-clicking on the file will begin the installation process. From here, simply keep clicking **Next** to accept all of the defaults. If you are asked if you want to install software on this computer, click **Yes**. To exit the installer and complete your Python installation, simply click **Finish**.

Depending on which version of Python you install, the images shown in Figures 1–5 may be somewhat different. In version 3.4, the installation process does not automatically edit the system Path environment variable. Starting with version 3.5, the installation process allows you to automatically edit the Path variable. I strongly suggest you ensure the **Add Python 3.5 to PATH** check box is selected.

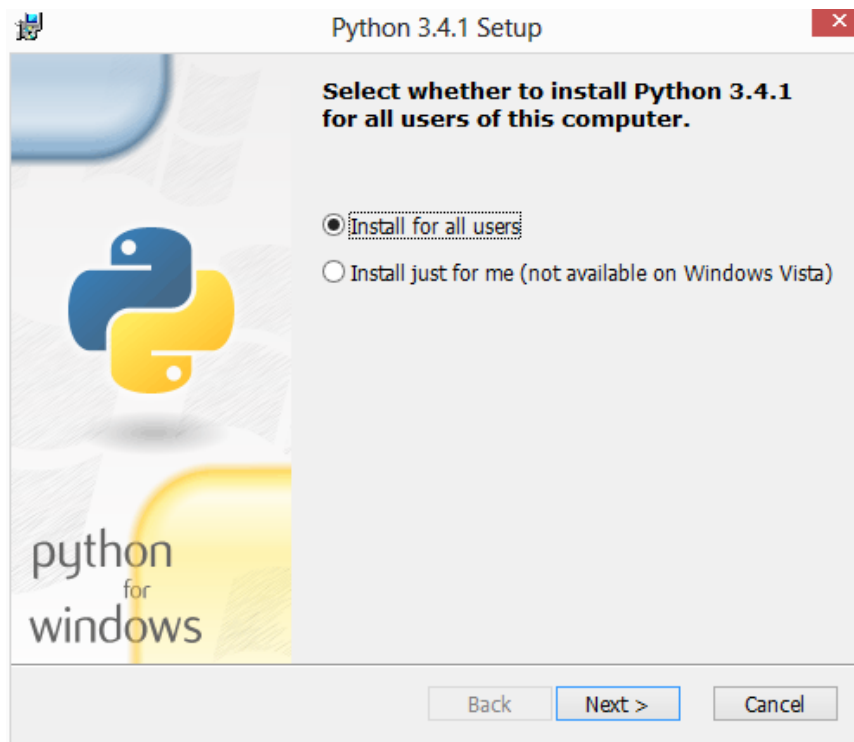


Figure 1: Installing Python

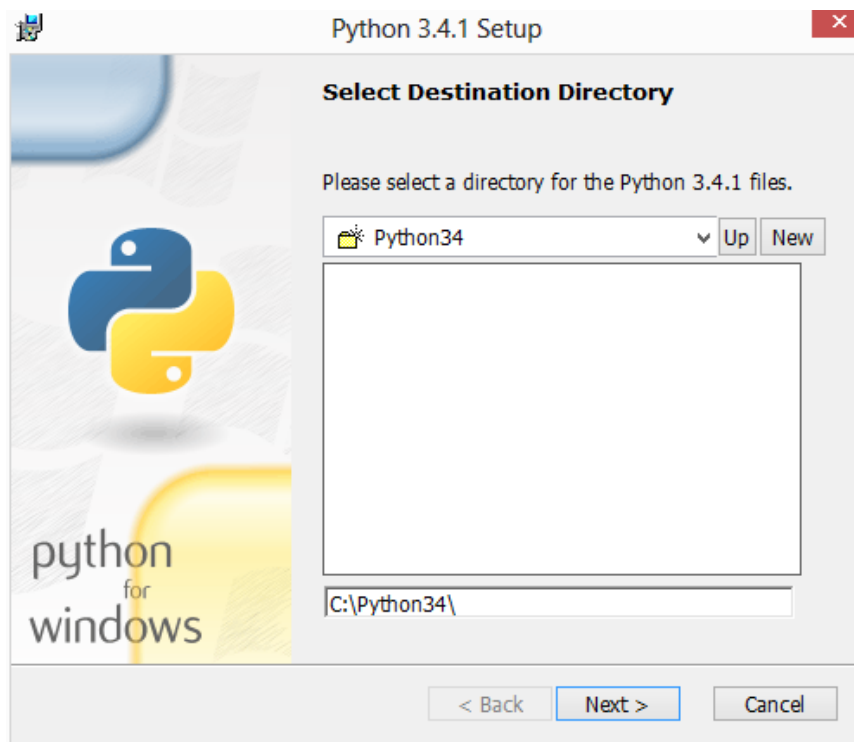


Figure 2: Installing Python

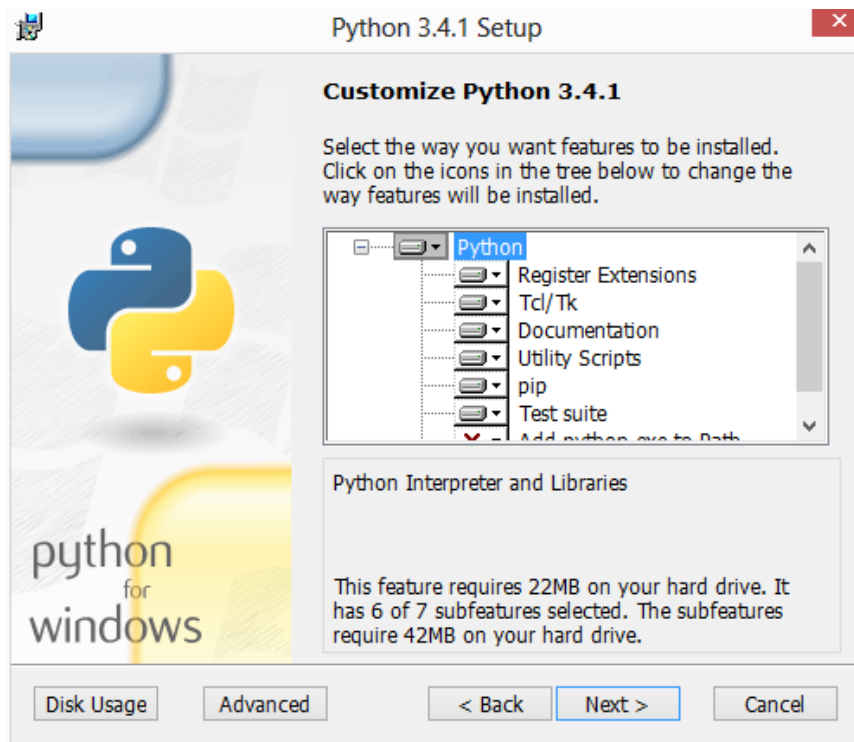


Figure 3: Installing Python

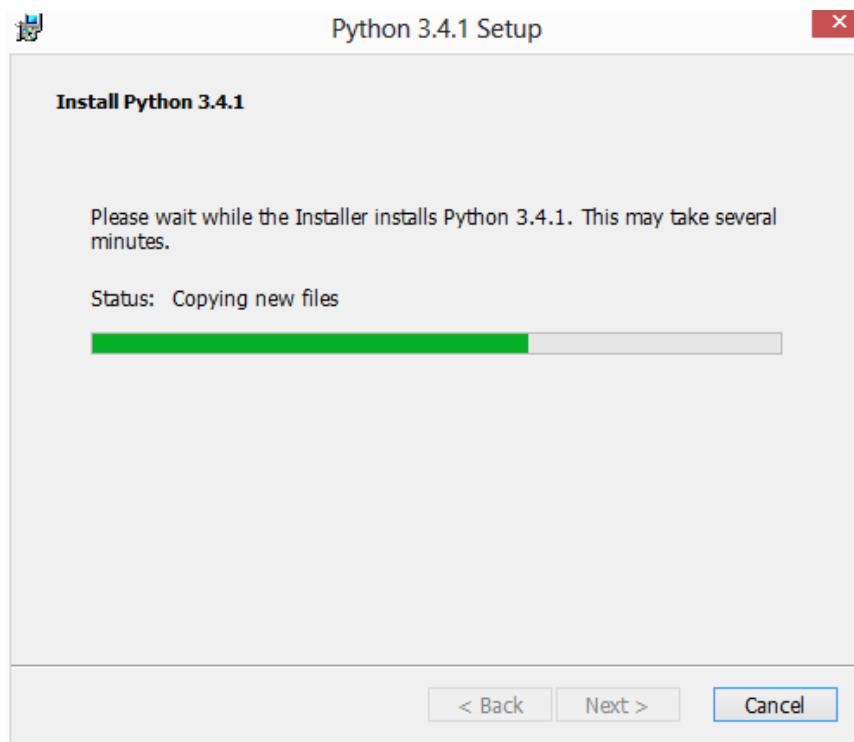


Figure 4: Installing Python



Figure 5: Installing Python

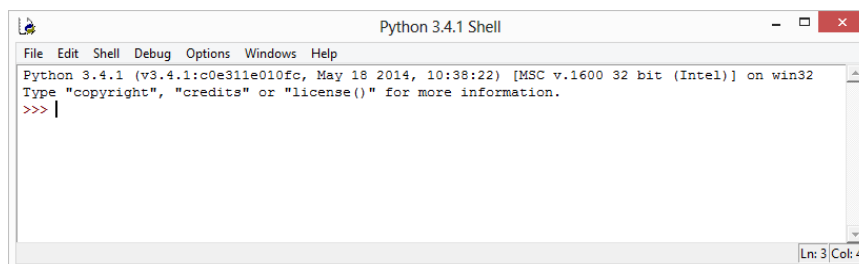


Figure 6: Python Installed

Mac Installation Instructions

At the time of this writing the current Mac operating system ships with Python 2. You will most likely want to upgrade to the most current version of Python, and to do this you will need to download and install it yourself. Visit the Python downloads page at <https://www.python.org/downloads> and click **Download Python 3.x.x**. Once downloaded, double-click the file to access the contents of the disk image. Double-click the **Python.mpkg** file to run the installer. You may encounter a message stating that "Python.mpkg can't be opened because it is from an unidentified developer." If this occurs you will need to hold control and click the Python.mpkg file. From there, select **Open with**, and finally, click **Installer**. Once asked if you are sure you want to open it, click **Open**. If you are asked to enter an administrator's username and password, do so.

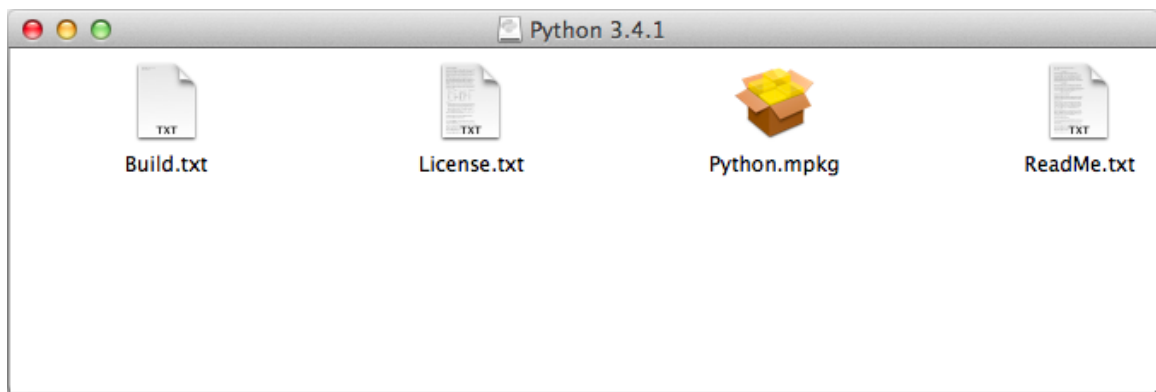


Figure 7: Installing Python



Figure 8: Installing Python

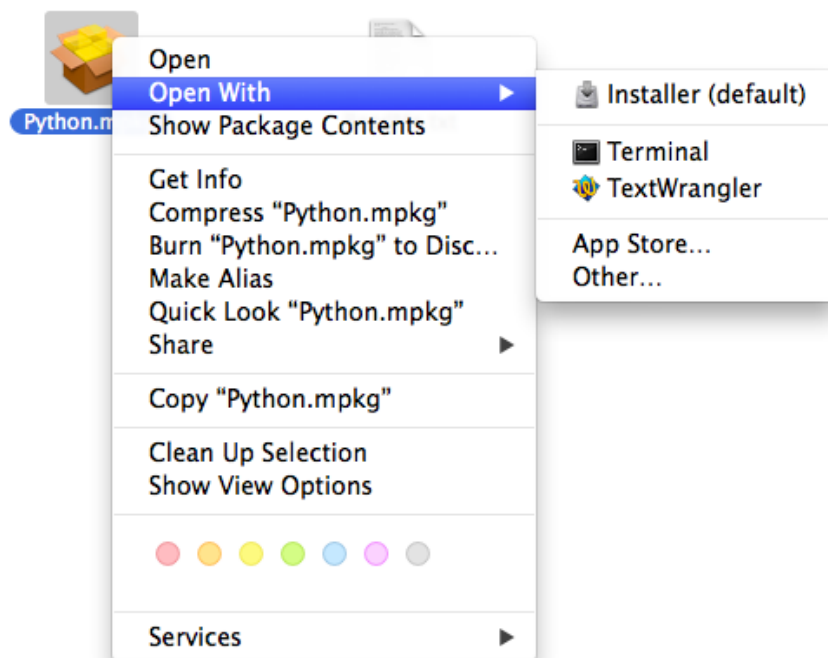


Figure 9: Installing Python



Figure 10: Installing Python

Accept all of the defaults as you click through the installer.

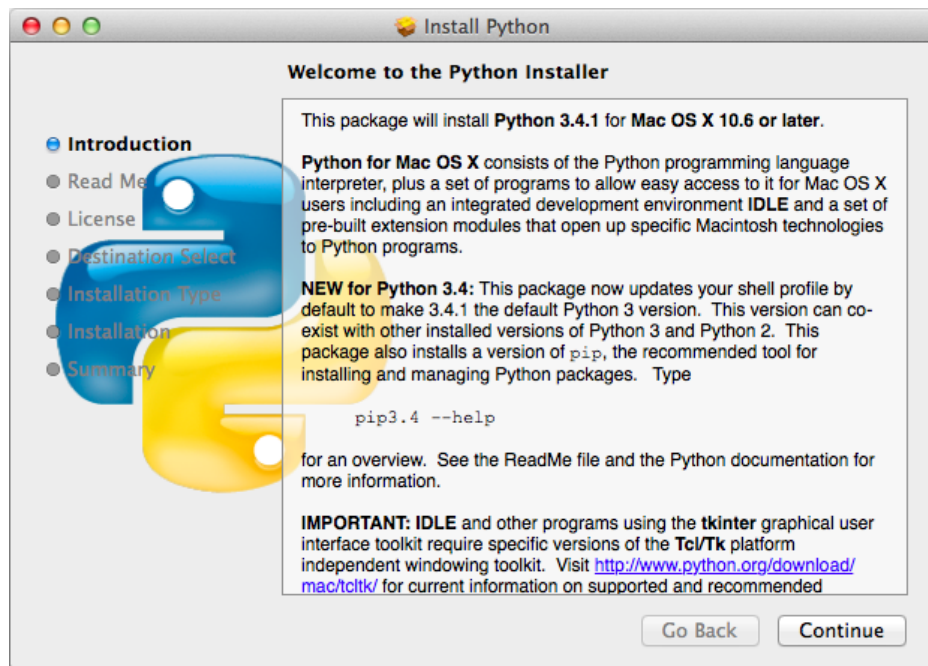


Figure 11: Installing Python

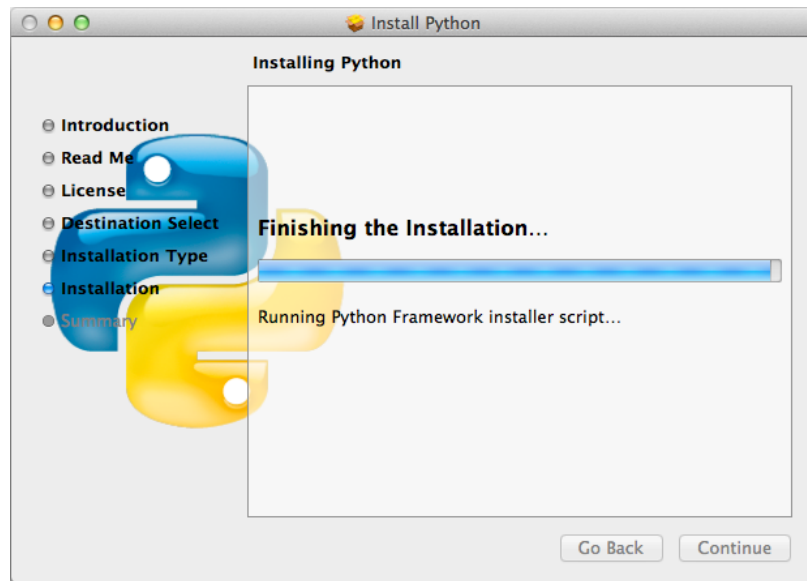


Figure 12: Installing Python

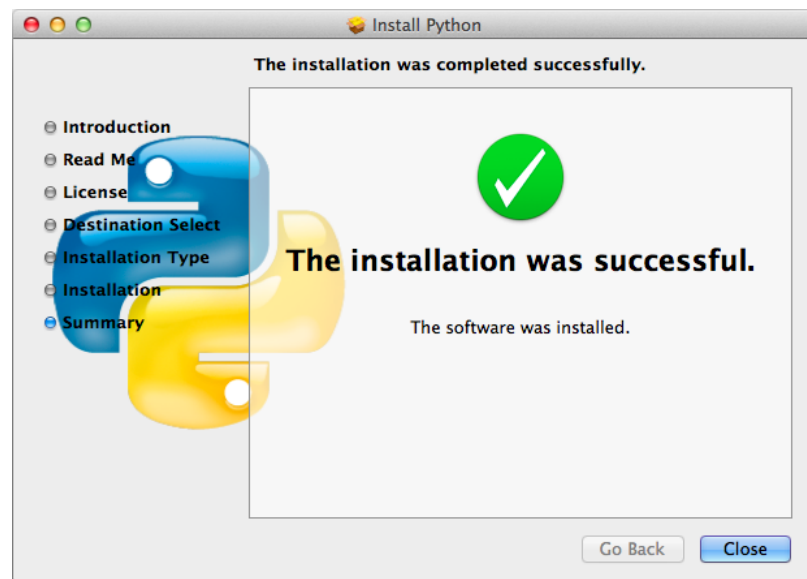


Figure 13: Python Installed

Once the installation is complete, you will find your Python folder inside the Applications folder on your computer. Within the Python folder you will discover a link to IDLE, the Integrated DeveLopment Environment, as well as a link to some Python documentation. In addition to accessing Python from IDLE, you will also be able to open the Terminal application, located at /Application/Utilities/Terminal, and run **python3**. Later in this chapter we will discuss in more detail how to run Python programs using IDLE and the command line. To check whether the installation was successful, run the following commands. The first command shows where Python was installed. The second command displays the version of Python that was installed.

Code Listing 1

```
[jason@mac ~]$ which python3
```

```
/Library/Frameworks/Python.framework/Versions/3.4/bin/python3
```

```
[jason@mac ~]$ python3 --version
```

```
Python 3.4.1
```

Linux Installation Instructions

In the case of Linux distributions, there will be some that ship with only Python 2 installed. However, it is becoming increasingly common to see Python 2 and Python 3 installed by default. To determine which version of Python you have installed, try opening a terminal emulator application such as xterm or console and type **python --version** and **python3 --version** at the command prompt. Many times the **python** command will actually be Python 2, and there will be a separate **python3** command used for running Python 3.

Code Listing 2

```
[jason@linuxsvr ~]$ python --version
```

```
Python 2.7.9
```

```
[jason@linuxsvr ~]$ python3 --version
```

```
Python 3.4.1
```

In the specific case where **python** or **python3** is not installed on your Linux system, you will receive an error message stating "command not found." In the following example, Python 2 is installed, but Python 3 is not.

Code Listing 3

```
[jason@linuxsvr ~]$ python --version
```

```
Python 2.7.9
```

```
[jason@linuxsvr ~]$ python3 --version
```

```
python3: command not found
```

Installing Python on Debian-Based Linux Distributions

In order to install Python 3 on Debian-based distributions such as Debian, Ubuntu, and Linux Mint, try running **apt-get install -y python3 idle3**. In any instance where you are attempting to install software, you will need to use superuser privileges. To do this you will need to execute the **apt** command as the root user, or precede the command with **sudo**. Keep in mind that **sudo** will work only if it has been previously configured, either by you, the distribution, or the system administrator. The following listing is an example of using **sudo** to install Python 3 on an Ubuntu Linux system.

Code Listing 4

```
[jason@ubuntu ~]$ sudo apt-get install -y python3 idle3
...
Setting up python3
[jason@ubuntu ~]$ python3 --version
3.4.1
```

In order to perform the installation as root, you will need to either switch to the root user using the **su -** command, or log into the Linux system as root.

Code Listing 5

```
[jason@ubuntu ~]$ su -
Password:
[root@ubuntu ~]# apt-get install -y python3 idle3
...
Setting up python3
[root@ubuntu ~]# python3 --version
3.4.1
[root@ubuntu ~]# exit
[jason@ubuntu ~]$
```

Installing Python on RPM Based Linux Distributions

For RPM-based Linux distributions such as RedHat, CentOS, Fedora, and Scientific Linux, use the command **yum install -y python3 python3-tools** when you are attempting to install Python 3. Installing software requires root privileges, so ensure that you run the command as root, or precede it with **sudo**. Note that **sudo** will only work in the instance where it has been previously configured, either by you, the distribution, or the system administrator. Here is an example of installing Python 3 on a Fedora Linux system using **sudo**.

Code Listing 6

```
[jason@fedora ~]$ sudo yum install -y python3 python3-tools
...
Complete!
[jason@fedora ~]$ python3 --version
3.4.1
```

If during installation you receive an error message such as "No package python3 available," or "Error: Nothing to do," then it will be necessary for you to install Python 3 directly from source code. Begin this process by installing the tools required in order to build and install Python. You can do this by running **yum groupinstall -y 'development tools'** with root privileges. From there, install the remaining dependencies by running **yum install -y zlib-dev openssl-devel sqlite-devel bzip2-devel tk-devel**.

Code Listing 7

```
[jason@centos ~]$ sudo yum groupinstall -y 'development tools'
...
Complete!
[jason@centos ~]$ sudo yum install -y zlib-dev openssl-devel sqlite-devel bzip2-
devel tk-devel
...
Complete!
```

Your next step will be to visit the Python downloads page at <https://www.python.org/downloads>, and click **Download Python 3.x.x**. Using a terminal emulator application, navigate to the directory where you just saved the Python download. Extract the contents of the file using **tar xf Python*z**. Change into the directory that was created by performing the extraction with the **cd Python-*** command. Run **./configure**, followed by **make**, and finally, as root, run **make install**. If **sudo** is configured on your system you can run **sudo make install**. This process will install Python 3 into the `/usr/local/bin` directory.

Code Listing 8

```
[jason@centos ~]$ cd ~/Downloads
[jason@centos ~/Downloads]$ tar xf Python*z
[jason@centos ~/Downloads/Python-3.4.1]$ cd Python-*
[jason@centos ~/Downloads/Python-3.4.1]$ ./configure
...
creating Makefile
[jason@centos ~/Downloads/Python-3.4.1]$ make
...
[jason@centos ~/Downloads/Python-3.4.1]$ sudo make install
...
[jason@centos ~/Downloads/Python-3.4.1]$ which python3
/usr/local/bin/python3
[jason@centos ~/Downloads/Python-3.4.1]$ python3 --version
Python 3.4.1
```

To acquire a greater depth of knowledge regarding Linux operating systems, I encourage you to read *Linux for Beginners*. You can get your copy by visiting <http://www.LinuxTrainingAcademy.com/linux>.

Preparing Your Computer for Python

It is important to be able to run the Python interpreter interactively, as well as execute existing Python programs. We refer to Python as an interpreter because it translates the Python language into a format that is understood by the underlying operating system and hardware. When you use the Python interpreter interactively, you are able to type Python commands and receive immediate feedback. It's an excellent way to experiment with Python, as well as answer the age old question, "I wonder what happens when I do this?"

There are two ways to start the Python interpreter. The first way is through launching the IDLE application, otherwise known as the Integrated DeveLopment Environment. The other way to start the Python interpreter is by using the command line. When using Windows, start the command prompt and type **python**. This process will be explained in more detail shortly. On Mac and Linux systems, execute **python3** directly from the command line. To exit the Python interpreter type **exit()** or **quit()**. You can also press Ctrl+D on Mac and Linux, or Ctrl+Z on Windows, to exit the interpreter. The following is an example of running the Python interpreter on a Mac system using the command line.

Code Listing 9

```
[jason@mac ~]$ python3
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello')
Hello
>>> exit()
[jason@mac ~]$
```

Don't worry about the **print('Hello')** line at this point. In the following chapters you will learn the details of that and other important commands. For now, just know that you can either start the IDLE application or execute the Python command in order to interact directly with the Python interpreter.

Running Python Programs

Once you've understood how to use the Python interpreter interactively, you will need to find a way to create, save, and execute Python programs. Keep in mind that Python programs are simply text files that include a series of Python commands. Typically Python programs end with a **.py** extension.

Running Python Programs on Windows

If you are going to run a Python program on Windows, one way is to navigate to the location of the Python file using Windows Explorer. Once you've found the file, double-click it. The disadvantage of using this method is that once the program exits, the program's window will close. In this instance you may not be able to view the output that was generated by the program, especially if no user interaction was involved. A far better way to run Python programs is by using the command line, sometimes called the command prompt in Windows.

To do this we first need to make sure that the Python interpreter is actually in our path. Using Windows Explorer, navigate to the folder where you have Python installed. If you accepted the defaults during the installation process, then the path should be C:\PythonNN, where NN is the version number. For example, if you installed Python 3.4 it would be C:\Python34. From there, navigate to the Tools folder, and then to the Scripts folder. Double-click the win_add2path file. The full path to this file is C:\Python34\Tools\Scripts\win_add2path.py. You will briefly see a window pop up and then disappear. This script adds the location of Python to your PATH so you can launch Python from the command prompt. If you are using version 3.5, you can select the **Add Python 3.5 to PATH** check box during installation to have this step performed for you. Also note that the default install location is different in version 3.5.

Locate the Command Prompt application and open it. There are a number of ways you can do this, depending on the version of Windows you are currently using. The following procedure will work on most, if not all, versions of Windows. Press the Windows logo key+R. The **Run** prompt will open. Type cmd and press **Enter**.

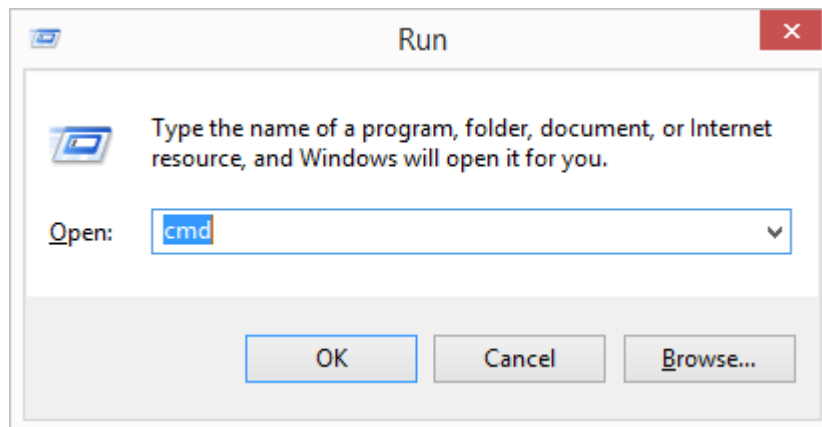
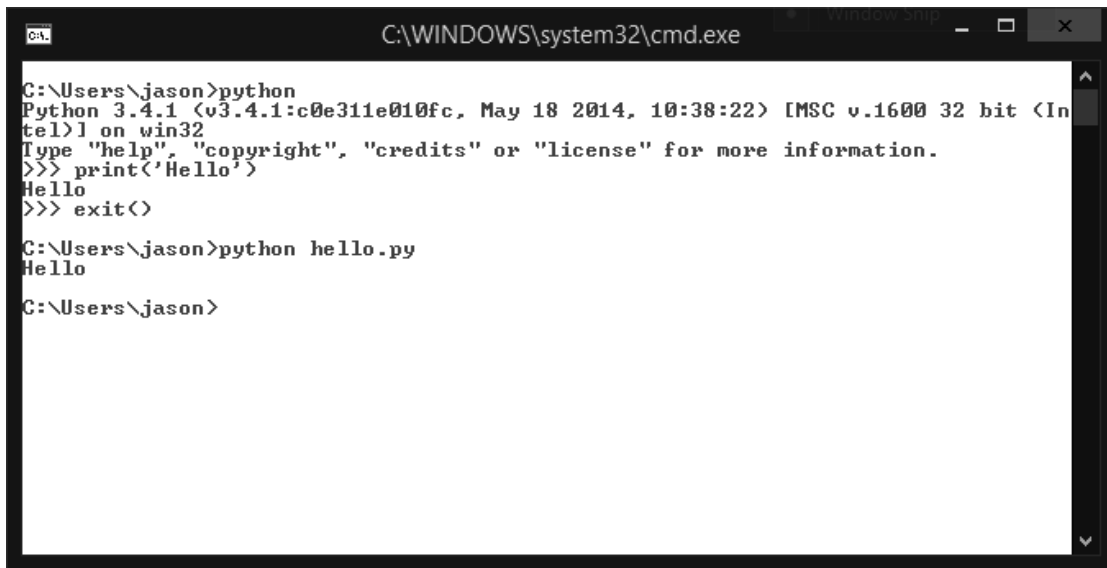


Figure 14: Locating the Command Prompt on Windows

Searching for the command prompt is also an option. For Windows Vista and Windows 7, just click the **Start** button, type **cmd** in the search box, and press **Enter**. When using Windows 8, click the **Search** icon, type **cmd** in the Search box, and press **Enter**.

Once the command prompt has been opened, you can run Python interactively by typing **python**, or run a Python application by typing **python program_name.py**. At this point you may receive an error message such as, "python is not recognized as an internal or external command, operable program or batch file." If this is the case, try rebooting your computer and then repeat the process.

The following figure shows the option of running Python interactively from the command line, and then from there running the **hello.py** program.



```
C:\WINDOWS\system32\cmd.exe

C:\Users\jason>python
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('Hello')
Hello
>>> exit()

C:\Users\jason>python hello.py
Hello

C:\Users\jason>
```

Figure 15: Running Python Interactively

Running Python Programs on Mac and Linux

When using Mac and Linux you can execute a Python program by running **python3 program_name.py** directly from the command line. From here the Python interpreter will read, interpret, and execute the code in the file that follows the **python** command.

The following code listing is the body of the **hello.py** file.

Code Listing 10

```
print('Hello')
```

The following code listing is what you will see when you run the program.

Code Listing 11

```
[jason@mac ~]$ python3 hello.py

Hello

[jason@mac ~]$
```

As well as supplying a Python file to the **python3** command, you can also directly execute the file by setting the execute bit on the file, and specifying Python in the interpreter directive on the first line. To set the execute bit on the file, run **chmod +x program_name.py** from the command line. To set the interpreter directive, make sure **#!/usr/bin/env python3** is the very first line in the Python file. Now you can run the Python program by using either a relative or an absolute path to the file.

The following code is the body of the **hello2.py** file.

```
#!/usr/bin/env python3  
  
print('Hello')
```

The following example demonstrates how you can set the executable bit on **hello2.py**, execute it using a relative path, execute it using an absolute path, and execute it by supplying it as an argument to the **python3** command.

```
[jason@mac ~]$ chmod +x hello2.py  
  
[jason@mac ~]$ ./hello2.py  
Hello  
  
[jason@mac ~]$ /Users/jason/hello2.py  
Hello  
  
[jason@mac ~]$ python3 hello2.py  
Hello  
  
[jason@mac ~]$
```

It is important to note that is completely safe to include the interpreter directive, even if the program will be executed using a Windows system. Windows will simply ignore the line and execute the remaining Python code.

Creating and Editing Python Source Code

The IDLE application is useful in that it not only allows you to use the Python interpreter interactively, but also grants you the ability to create, edit, and execute Python programs. To create a new Python program, open the **File** menu and select **New File**. If you are looking to open an existing Python file, go to the **File** menu and select **Open**. From here you can type or edit your Python program. Save your program simply by accessing the **File** menu and selecting **Save**. To run the program, press **F5** or open the **Run** menu and select **Run Module**.

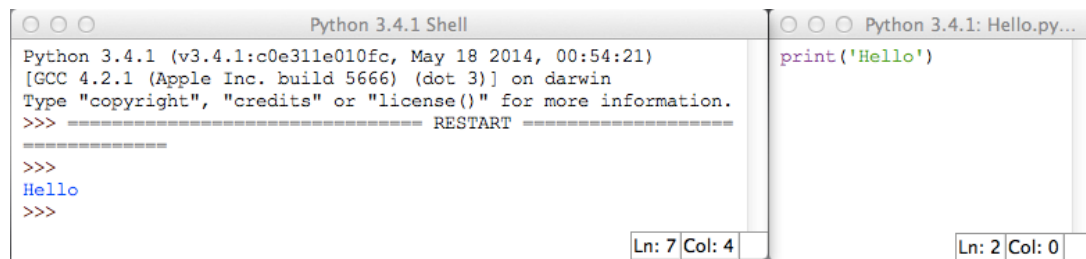


Figure 16: IDLE

Keep in mind that since Python source code is nothing more than a text file, you are not purely limited to using the IDLE editor. Feel free to use your favorite text editor to create Python files, and then execute them from the command line as discussed previously. You don't have to be limited as there are many great text editors available. I've listed below some of my favorite editors for Windows, Mac, and Linux.

Windows

Geany: <http://www.geany.org/>

JEdit: <http://www.jedit.org/>

Komodo Edit: <http://komodoide.com/komodo-edit/>

Notepad++: <http://notepad-plus-plus.org/>

Mac

JEdit: <http://www.jedit.org/>

Komodo Edit: <http://komodoide.com/komodo-edit/>

Sublime Text: <http://www.sublimetext.com/>

TextWrangler: <http://www.barebones.com/products/textwrangler/>

Linux

Emacs: <https://www.gnu.org/software/emacs/>

Geany: <http://www.geany.org/>

JEdit: <http://www.jedit.org/>

Komodo Edit: <http://komodoide.com/komodo-edit/>

Sublime Text: <http://www.sublimetext.com/>

Vim: <http://www.vim.org/>

In Python, indentation is important. The recommended indentation is four spaces but some programmers prefer two or three spaces. However many spaces you use, it's critically important to be consistent. I highly recommend you program your editor to insert these four spaces when you press the Tab key. Also, make sure to configure your editor to save files using Unix line endings. This will ensure that your programs will be cross-platform compatible. If you do this you will then have no issue using the same file on Windows, Mac, and Linux.

Downloading the Source Code Examples

If at any point you would like to download the examples from this book, visit <https://bitbucket.org/syncfusiontech/python-succinctly>. While it may be easier to simply look at the code examples and run them, it is far more beneficial for you to take the extra time to type them out yourself. Typing the source code will help establish and reinforce exactly what you are learning. It also allows you valuable practical experience in fixing the issues that will ultimately arise when you are creating your own code. A key example of this is when you have to find and spot spelling mistakes, as well as locate syntax errors in your code. They may seem like little things, but details like spacing, spelling, capitalization, and punctuation marks are all crucial to writing functional programs. Of course you may get stuck on an exercise and need to refer back to the examples in this book. If so, compare your code to the code you have both downloaded and read about in this book, and from there try to spot the differences.

Review

Install Python. When at all possible use Python 3. If you do need to use Python 2, opt for Python 2.7.

Run Python interactively either by using IDLE, or by executing the Python command at the command line. Use **python** for Windows, and **python3** for Mac and Linux.

Press **F5** or navigate to the **Run** menu and select **Run Module** in order to run Python programs in IDLE. You can also run Python programs from the command line by executing the Python command followed by a Python file. Keep in mind that for Windows the pattern is **python program_name.py**, while for Mac and Linux the pattern is **python3 program_name.py**.

While you can use IDLE to edit your Python source code, you may also opt to use any text editor of your choice.

Download the example source code from <https://bitbucket.org/syncfusiontech/python-succinctly>.

Resources

Integrated Development Environments for Python:

<https://wiki.python.org/moin/IntegratedDevelopmentEnvironments>

Open the Command Prompt in Windows: <http://www.wikihow.com/Open-the-Command-Prompt-in-Windows>

Python 3 Installation Video for Linux: <https://www.youtube.com/watch?v=RLPYBxfAud4>

Python 3 Installation Video for Mac: https://www.youtube.com/watch?v=EZ_6tmtbDSM

Python 3 Installation Video for Windows: <https://www.youtube.com/watch?v=CihHoWzmFe4>

Should I use Python 2 or Python 3 for my development activity?
<https://wiki.python.org/moin/Python2orPython3>

Source Code Examples for this Book: <http://www.LinuxTrainingAcademy.com/python-succinctly>

Chapter 1 Variables and Strings

Variables

Put quite simply, variables are named storage locations. They can also be described as name-value pairs. It is possible for you to assign values to a variable, and then recall those values by the variable name. To assign a value to a variable, use the equal sign. The pattern is **variable_name = value**.

In the following example, the value **asparagus** is assigned to the variable called **vegetable**.

Code Listing 14

```
vegetable = 'asparagus'
```

It is possible to change the value of a variable simply by reassigning it. View the following example to see how to reset the value of the **vegetable** variable to the value **onion**.

Code Listing 15

```
vegetable = 'onion'
```

Note that there is nothing significant about the variable named **vegetable** in this example. We could have just as easily used the words **food**, **crop**, **produce**, or almost any other variable name that we could possibly think of. When choosing a variable name, you want to select something that will ultimately represent the data the variable will hold. While you may know what a variable named **v** represents today, it may not be so fresh in your mind when you return to the code a few months from now. However, if you come across a variable named **vegetable**, chances are you'll have a greater understanding of what data it might hold.

Keep in mind that variable names are always case sensitive. So, variables **Vegetable** and **vegetable** will be two distinct and separate variables. By convention, variables are usually named with all lower case letters, but this is by no means a requirement. While variable names can contain numbers in the body of the name, they must always start with a letter. Another rule to remember is that you cannot use a hyphen (-), plus sign (+), or other various symbols in variable names. You can however use the underscore (**_**) character.

The following are some examples of valid variable names.

Code Listing 16

```
last3letters = 'XYZ'
```

```
last_three_letters = 'XYZ'

lastThreeLetters = 'XYZ'
```

Strings

A string is utilized to represent text. In the previous examples strings were represented by the text **asparagus**, **onion**, and **XYZ**. In Python strings are always surrounded by quotes. Let's revisit the first example in this chapter when we created a variable named **vegetable** and assigned it the string **asparagus**.

Code Listing 17

```
vegetable = 'asparagus'
```

Strings can also be encapsulated by the use of double quotes.

Code Listing 18

```
vegetable = "asparagus"
```

Using Quotes within Strings

It is important to remember that Python requires matching quotation marks for all strings that you enter. Whenever you begin a string definition by using a double quotation mark, Python will interpret the next double quotation mark you enter as the end of that string. The same will be true when using single quotation marks. If you begin a string with a single quotation mark, the next single quotation mark will represent the end of that particular string.

In instances where you want to include double quotations in a string, make sure to place them inside single quotation marks as in the following example.

Code Listing 19

```
sentence = 'He said, "That asparagus tastes great!"'
```

If you want to incorporate single quotes in a string, make sure to enclose the entire string in double quotation marks.

```
sentence = "That's some great tasting asparagus!"
```

What if you were looking to use both single and double quotes in the same string? At this point you would need to escape the offending quotation character by prepending it with a backslash (\). The following code listing demonstrates how to escape the following string when making use of double and single quotes.

```
He said, "That's some great tasting asparagus!"
```

```
sentence_in_double = "He said, \"That's some great tasting asparagus!\""
sentence_in_single = 'He said, "That\'s some great tasting asparagus!"'
```

Indexing

It is important to note that each character in a string will be assigned an index. All string indices are zero based, which means that the first character in any string will have an index of 0, the second character will have an index of 1, and so on.

```
String:  a s p a r a g u s
Index:   0 1 2 3 4 5 6 7 8
```

In order to access the character at a given index, append **[N]** to a string where **N** is the index number. The following example creates a variable which is named **a** and assigns it the character in position 0 of the string **asparagus**. Similarly, a variable of **r** is created using the character from position 4 of **asparagus**.

```
a = 'asparagus'[0]
r = 'asparagus'[4]
```

Since variables are quite simply names that represent their values, the **[N]** syntax will also work with any other variable. In the following example, **first_char** will be assigned the value **a**.

Code Listing 24

```
vegetable = 'asparagus'
first_char = vegetable[0]
```

Built-in Functions

A function is an action-performing section of reusable code. A function will always have a name and will be called, or executed, by that name. Optionally, functions are able to accept arguments as well as return data.

The print() Function

The **print()** function is just one of Python's many built-in functions. Any time a value is provided as an argument to the **print()** function, it will display that value to the screen. You can supply literal values like **"cat"** or **7** to the **print** statement or opt to pass in variables.

Code Listing 25

```
vegetable = 'asparagus'
print(vegetable)
print('onion')
```

Output:

Code Listing 26

```
asparagus
onion
```

The len() Function

Another useful built-in Python function is the **len()** function. When a string is passed as an argument to the **len()** function, it returns the length of that string. Put more simply, **len()** returns the number of characters in a string.

In the following example the value **asparagus** is assigned to the variable named **vegetable**. From there we assign the result of **len(vegetable)** to the **vegetable_len** variable. Finally we display that value to the screen by making use of the **print(vegetable_len)** function.

Code Listing 27

```
vegetable = 'asparagus'
vegetable_len = len(vegetable)
print(vegetable_len)
```

Output:

Code Listing 28

```
9
```

You can also skip the intermediary step of assigning it to a variable and pass the **len()** function directly to the **print()** function. This works because **len(vegetable)** is evaluated first, and from there its value is used by the **print()** function.

Code Listing 29

```
vegetable = 'asparagus'
print(len(vegetable))
```

Output:

Code Listing 30

```
9
```

If you're so inclined you can even skip using variables all together.

Code Listing 31

```
print(len('asparagus'))
```

Output:

String Methods

Without delving too deeply into the subject of object-oriented programming (OOP), it can be helpful to understand a few key concepts before continuing. One of the first things you should know is that absolutely everything in Python is an object. In turn, every object has a type. Though you are currently learning about the string data type, we will cover various other types throughout the course of this book.

For now let's focus our attention on strings. For example, `'asparagus'` is an object with a type of `str`, which is short for string. Simply put, `'asparagus'` is a string object. If we assign the value `asparagus` to the variable `vegetable` using `vegetable = 'asparagus'`, then `vegetable` is also a string object. Keep in mind that variables are names that represent their values.

As mentioned previously, a function is a section of reusable code that will perform an action. Up to this point you have been using built-in functions like `print()` and `len()`. Objects also have functions, but they are not usually described as such. In fact, they are called methods. Methods are merely functions that are run against an object. In order to call a method on an object, simply follow the object with a period, then the method name, and finally a set of parentheses. Make sure to enclose any parameters within the parentheses.

The `lower()` String Method

The `lower()` method of a string object will return a copy of the string in all lowercase letters.

Code Listing 33

```
vegetable = 'Asparagus'
print(vegetable.lower())
```

Output:

Code Listing 34

```
asparagus
```

The `upper()` String Method

Conversely, the `upper()` string method will return a copy of the string in all uppercase letters.

```
vegetable = 'Asparagus'  
print(vegetable.upper())
```

Output:

```
ASPARAGUS
```

String Concatenation

To concatenate, or combine two strings, use the plus sign. A simple way of thinking about this is imagining that you were adding strings together. You can concatenate multiple strings by using additional plus signs and strings. In the following example take note of how spaces are included in the strings. String concatenation only combines the strings as they are.

```
print('Python ' + ' is ' + 'fun.')  
print('Python' + ' is' + ' Python.')
```

Output:

```
Python is fun.  
Python is fun.
```

If you fail to include extra spaces, it will be reflected in your output, as in the following example.

```
print('Python' + 'is' + 'fun.')
```

Output:

```
Pythonisfun.
```

The next example demonstrates string concatenation using variables combined with the space character literal.

```
first = 'Python'
second = 'is'
third = 'fun'
sentence = first + ' ' + second + ' ' + third + '.'
print(sentence)
```

Output:

```
Python is fun.
```

Repeating Strings

It is important to note that whenever you are working with strings, the asterisk is the repetition operator. The pattern is `'string' * number_of_times_to_repeat`. For example, if you want to display a hyphen twelve times, use `'-' * 12`.

```
print('-' * 12)
```

Output:

```
-----
```

Keep in mind that you don't have to use repetition with just single character strings.

Code Listing 45

```
good_times = 'fun ' * 3
print(good_times)
```

Output:

Code Listing 46

```
fun fun fun
```

The **str()** Function

In a later chapter of this book you will learn about numeric data types. For now though, just know that unlike strings, numbers will not be enclosed within quotation marks. To concatenate a string with a number, you must first convert the number to a string with the built-in **str()** function. The **str()** function will turn non-strings, such as numbers, into strings.

Code Listing 47

```
version = 3
print('Python ' + str(version) + ' is fun.')
```

Output:

Code Listing 48

```
Python 3 is fun.
```

The following example shows you what will happen if a number is not converted to a string before you attempt concatenation.

Code Listing 49

```
version = 3
print('Python ' + version + ' is fun.')
```

Output:

Code Listing 50

```
File "string_example.py", line 2, in <module>
    print('Python ' + version + ' is fun.')
TypeError: Can't convert 'int' object to str implicitly
```

Formatting Strings

Calling the **format()** method on a string to produce the format you desire is an alternative to directly concatenating strings. Do this by creating placeholders, also known as format fields, by using curly braces in the string and passing in values for those fields to **format()**.

By default the first pair of curly braces will always be replaced by the first value passed to **format()**. The second pair of curly braces will be replaced by the second value passed to **format()**, and so on. The following example illustrates this.

Code Listing 51

```
print('Python {} fun.'.format('is'))
print('{} {} {}'.format('Python', 'is', 'fun.'))
```

Output:

Code Listing 52

```
Python is fun.
Python is fun.
```

Be sure to note that when you pass multiple objects to a function or method you must separate them using a comma.

Also, you can implicitly state which positional parameter will be used for a format field simply by providing a number inside the braces. **{0}** will be replaced with the first item passed to **format()**, **{1}** will be replaced by the second item passed in, and so on.

Code Listing 53

```
print('Python {0} {1} and {1} {0} awesome!'.format('is', 'fun'))
```

Output:

Code Listing 54

```
Python is fun and fun is awesome!
```

The following formatting example makes use of variables.

Code Listing 55

```
first = 'Python'
second = 'is'
third = 'fun'
print('{} {} {}'.format(first, second, third))
```

Output:

Code Listing 56

```
Python is fun.
```

With what we've learned we can now rewrite our previous example combining strings and numbers by using the **format()** method. This completely eliminates the need to use the **str()** function.

Code Listing 57

```
version = 3
print('Python {} is fun.'.format(version))
```

Output:

```
Python 3 is fun.
```

When needed, you can also supply a format specification. Format specifications will be confined within the curly braces. To create a field with a minimum character width, simply supply a number after the colon. The format field `{0:9}` will translate to “use the first value provided to `format()` and make it at least nine characters wide.” The format field `{1:8}` means “use the second value provided to `format()` and make it at least eight characters wide.” This method can be useful in many instances, including the creation of tables.

```
print('{0:9} | {1:8}'.format('Vegetable', 'Quantity'))
print('{0:9} | {1:8}'.format('Asparagus', 3))
print('{0:9} | {1:8}'.format('Onions', 10))
```

Output:

```
Vegetable | Quantity
Asparagus |      3
Onions    |     10
```

In order to control the alignment, always use `<` for left, `^` for center, and `>` for right. If no particular alignment is specified, left alignment will always be assumed. Making use of our previous example, let's try to left align the numbers.

```
print('{0:9} | {1:<8}'.format('Vegetable', 'Quantity'))
print('{0:9} | {1:<8}'.format('Asparagus', 3))
print('{0:9} | {1:<8}'.format('Onions', 10))
```

Output:

```
Vegetable | Quantity
Asparagus | 3
Onions    | 10
```

If needed, you can also specify a data type. The most common instance of this is to use **f** which will represent a float. Floats, or floating point numbers, will be addressed in depth in the following chapter. Also, you can stipulate the number of decimal places by using **.Nf** where **N** is the number of decimal places. A common currency format would be **.2f** which specifies two decimal places. The following is an idea of what our table might look like once we've taken a few nibbles out of our asparagus.

```
print('{0:8} | {1:<8}'.format('Vegetable', 'Quantity'))
print('{0:9} | {1:<8.2f}'.format('Asparagus', 2.33333))
print('{0:9} | {1:<8.2f}'.format('Onions', 10))
```

Output:

```
Vegetable | Quantity
Asparagus | 2.33
Onions    | 10.00
```

Getting User Input

To accept standard input use the built-in function **input()**. By default, standard input originates from a person typing at a keyboard. This will allow you to prompt the user directly for their input. In more complex cases standard input can come from other sources. For example, you are able to send the output from one command as the standard input to another command just by using pipes. (For more info on this topic refer to [Linux for Beginners](http://www.linuxtrainingacademy.com/linux) at <http://www.linuxtrainingacademy.com/linux>.)

Keep in mind that you can pass in a prompt to display to the **input()** function.

```
vegetable = input('Enter a name of a vegetable: ')\nprint('{} is a lovely vegetable.'.format(vegetable))
```

Output:

```
Name a vegetable: asparagus\nasparagus is a lovely vegetable.
```

Review

Variables are names that store values.

Variable names may contain letters, numbers, and underscores, but must always begin using a letter.

Values can be assigned to variables using the **variable_name = value** pattern.

Strings are always surrounded by single or double quotation marks.

An index is assigned to each character in a string.

A function is an action performing reusable code.

Built-in functions:

- **print()**: Displays values.
- **len()**: Returns the length of an item.
- **str()**: Returns a string object.
- **input()**: Reads a string.

Absolutely everything in Python is an object.

It is possible for objects to have methods.

Methods are functions that will operate on an object.

String methods:

- `upper()`: Returns a copy of the string in uppercase.
- `lower()`: Returns a copy of the string in lowercase.
- `format()`: Returns a formatted version of the string.

Exercises

Animal, Vegetable, Mineral

Try to write a Python program that makes use of three variables. The variables you will use in your program will be **animal**, **vegetable**, and **mineral**. Make sure to assign a string value to each one of these independent variables. Your program should be able to display “Here is an animal, a vegetable, and a mineral.” From there, display the value for **animal**, followed by **vegetable**, and then finally **mineral**. Each one of the values should be printed on their own individual line. The output should be four lines in total.

Sample output:

Code Listing 67

```
Here is an animal, a vegetable, and a mineral.  
  
Deer  
  
spinach  
  
aluminum
```

I strongly encourage you to successfully create a Python program that is capable of producing the output in the previous code listing before continuing. For the remainder of this book the solutions to the exercises will follow the exercise explanation and sample output. If you want to attempt the exercise on your own—and I encourage you to do so—stop reading now.

Solution

Code Listing 68

```
animal = 'deer'  
  
vegetable = 'spinach'
```

```
mineral = 'aluminum'

print('Here is an animal, a vegetable, and a mineral.')
print(animal)
print(vegetable)
print(mineral)
```

Copy Cat

Try writing a Python program that directly prompts the user for input, and then simply repeats the information the user entered.

Sample output:

Code Listing 69

```
Please type something and press enter: Hello world!

You entered:

Hello world!
```

The following is one possible solution. It may be that your program looks slightly different, but ideally it should be fairly similar. One example of a possible difference is that you may find you have used a different variable name. If you successfully reproduced the previous output, keep at it! You're doing great!

Code Listing 70

```
user_input = input('Please type something and press enter: ')
print('You entered:')
print(user_input)
```

Pig Speak

Try writing a Python program that will prompt for input and then display a pig “saying” whatever text was provided by the user. Place the input you receive from the user inside a speech bubble. Expand or contract the speech bubble in order to make it fit around the input provided.

Sample output:

Code Listing 71

```

      _____
    < Feed me and I'll oink! >
    -----
      /
    ^..^ /
~( ( oo )
  , , , ,
```

Solution

Code Listing 72

```
text = input('What would you like the pig to say? ')
text_length = len(text)

print('          {}'.format('_' * text_length))
print('          < {} >'.format(text))
print('          {}'.format('-' * text_length))
print('          /')
print('      ^..^ /')
print('~( ( oo )')
print('  , , , ,')
```

Output:

What would you like the pig to say? Oink

```

      _____
    < Oink >
      ----
      /
    ^..^ /
~( ( oo )
  ' ' ' '

```

Resources

Common String Operations: <https://docs.python.org/3/library/string.html>

input() documentation: <https://docs.python.org/3/library/functions.html?highlight=input#input>

len() documentation: <https://docs.python.org/3/library/functions.html?highlight=input#len>

print() documentation: <https://docs.python.org/3/library/functions.html?highlight=input#print>

str() documentation: <https://docs.python.org/3/library/functions.html?highlight=input#func-str>

Chapter 2 Numbers, Math, and Comments

While we discussed in the previous chapter how to create strings by placing text within quotation marks, it is important to note that numbers in Python require no such special treatment. If you'd like to use a number, simply include it in your source code. If you want to assign a number to a variable, use the pattern **variable_name = number** as shown in the following example.

Code Listing 74

```
height = 70  
  
temperature = 98.6
```

It is important to note that Python supports both integers and floating point numbers. Integers are numbers without a decimal point, otherwise known as whole numbers. Floating point numbers however will always contain a decimal point. The data type for integers is **int**, while the data type for floating point numbers is **float**.

Numeric Operations

Keep in mind that the Python interpreter is capable of performing several operations using numbers. The following table lists the most commonly used numeric operations.

Table 1: Numeric Operators

Symbol	Operation
+	add
-	subtract
*	multiply
/	divide
**	exponentiate

Symbol	Operation
%	modulo

You are most likely familiar with the common symbols `+`, `-`, `*`, and `/`. The `**` operator represents exponentiation, otherwise known as “raising to the power of.” For example, `2 ** 4` means “2 raised to the power of 4.” The written out equivalent to this is `2 * 2 * 2 * 2`, which will result in an outcome of **16**.

The modulo operation is performed by the percent sign. Put quite simply, it will return the remainder. For example, `3 % 2` is **1** because 3 divided by 2 is 1 with a remainder of 1. `4 % 2` returns **0** since 4 divided by 2 is 2 with a remainder of 0. In general, modulo arithmetic is done using non-negative integers. Modulo arithmetic with negative numbers can be very tricky. For example, `-5 % 4` returns **3**.

By making use of these symbols, Python allows you to perform mathematical calculations directly within the interpreter.

Code Listing 75

```
[jason@mac ~]$ python3
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
>>> exit()
[jason@mac ~]$
```

You can also assign the resulting value of a mathematical operation to a variable.

Code Listing 76

```
sum = 3 + 2
difference = 88 - 2
```

```
product = 4 * 2
quotient = 16 / 4
power = 3 ** 5
remainder = 7 % 3

print('Sum: {}'.format(sum))
print('Difference: {}'.format(difference))
print('Product: {}'.format(product))
print('Quotient: {}'.format(quotient))
print('Power: {}'.format(power))
print('Remainder: {}'.format(remainder))
```

Output:

Code Listing 77

```
Sum: 5
Difference: 86
Product: 8
Quotient: 4.0
Power: 243
Remainder: 1
```

Take note that even though the result of `16 / 4` is the integer `4`, the floating point number `4.0` was displayed in the output created using the example in Code Listing 76. The division operator (`/`) performs floating point division, and will in every case return a floating point number and not an integer. Also, be aware that if you add an integer to a floating point number the result will always be a float.

The following example demonstrates the capability of Python to perform mathematical operations using variables.

Code Listing 78

```
sum = 3 + 4  
difference = 200 - 2  
new_number = sum + difference  
print(new_number)  
print(sum / sum)  
print(sum + 1)
```

Output:

Code Listing 79

```
205  
1.0  
8
```

Strings and Numbers

The following example establishes a variable named **quantity** and assigns it the numeric value **4**. It also creates a variable named **quantity_string** and assigns it the string **4**.

Code Listing 80

```
quantity = 4  
quantity_string = '4'
```

Keep in mind that if you try to perform a mathematical operation against a string, you will encounter an error. Try to be aware that if you surround a number with quotes it will become a string.

Code Listing 81

```
quantity_string = '4'
```

```
total = quantity_string + 1
```

Output:

Code Listing 82

```
Traceback (most recent call last):  
  File "string_test.py", line 2, in <module>  
    total = quantity_string + 1  
TypeError: Can't convert 'int' object to str implicitly
```

The int() Function

If you are looking to convert a string into an integer, use the **int()** function and pass in the string to convert.

Code Listing 83

```
quantity_string = '4'  
total = int(quantity_string) + 1  
print(total)
```

Output:

Code Listing 84

```
5
```

The float() Function

In order to convert a string into a floating point number, use the **float()** function and pass in the string to convert.

```
quantity_string = '4'  
quantity_float = float(quantity_string)  
print(quantity_float)
```

Output:

```
4.0
```

Comments

Comments can be a great benefit to us humans, but will be totally ignored by Python. The main benefit of comments is that they give you a way to document your code. For example, a comment can help summarize what is about to happen in a complex piece of code. This can be incredibly helpful if you or a fellow programmer need to look at the code at a later date. Using comments can quickly explain what the intention of the code was at the time it was written.

A single-line comment is prefixed with an octothorpe (#), which is also known as a pound sign, number sign, or hash.

```
# This is a comment. Python simply skips comments.
```

If desired, you can also chain multiple single-line comments together.

```
# The following code:  
  
#     Computes the hosting costs for one server.  
  
#     Determines the duration of hosting that can be purchased given a budget.
```

Another option is to create multi-line comments by using triple quotes. You can use either single quotes or double quotes. The comment will begin directly after the first set of triple quotes and will end directly before the following set of triple quotes.

Code Listing 89

```
""" The comment starts here.  
This is another line in the comment.  
Here is the last line of the comment. """
```

Here is another example.

Code Listing 90

```
"""  
This starts a comment down here!  
Python will not attempt to interpret these lines as they are comments.  
"""
```

It is even possible to create a single line quote by using the triple quote syntax.

Code Listing 91

```
"""Yet another comment."""
```

If we go back to our [“Pig Speak”](#) exercise in the previous chapter, you can practice adding in some of your own comments to make your code clearer.

Code Listing 92

```
# Get the input from the user.  
text = input('What would you like the pig to say? ')  
  
# Determine the length of the input.  
Text_length = len(text)  
  
# Make the border the same size as the input.
```

```

Print('          {}'.format('_' * text_length))
print('        < {} >'.format(text))
print('          {}'.format('-' * text_length))
print('        /')
print('    ^..^  /')
print('~( ( oo )')
print('  , ,  , ,')

```

Review

Unlike strings, numbers require no special decoration. When you enclose a number in quotes it will become a string.

Use the **int()** function to convert a string to an integer.

Use the **float()** function to convert a string to a float.

An octothorpe (#) will begin a single line comment.

Multiline comments must be enclosed with triple quotes (""").

Exercises

Calculate the Cost of Cloud Hosting

In this exercise let's assume that you are planning to build a social networking service using your new Python skills. You make the decision to host your application on servers running in the cloud. Once you've selected a hosting provider, you want to know how much it will cost to operate per day and per month. You will launch your service using one server, and your provider will charge \$1.02 per hour.

Try to write a Python program that will display the answers to the following questions:

How much will it cost to operate one server per day?

How much will it cost to operate one server per month?

Solution

The following is one way to use Python to find the answers to the preceding questions. Take note of the fact that comments are used throughout the code. Also, while this is one possible solution, keep in mind that there are multiple ways to go about solving the same problem.

Code Listing 93

```
# The cost of one server per hour.
Cost_per_hour = 1.02

# Compute the costs for one server.
Cost_per_day = 24 * cost_per_hour
cost_per_month = 30 * cost_per_day

# Display the results.
Print('Cost to operate one server per day is ${:.2f}'.format(cost_per_day))
print('Cost to operate one server per month is ${:.2f}'.format(cost_per_month))
```

Output:

Code Listing 94

```
Cost to operate one server per day is $24.48.
Cost to operate one server per month is $734.40.
```

Calculate the Cost of Cloud Hosting, Continued

Building upon the previous example, let's add some more information. Assuming that you have saved \$1,836 to fund your new business venture, you are now wondering how many days you can keep one server running before your money runs out. Ideally though, you are hoping that your social network becomes incredibly popular and ultimately requires 20 servers to keep up with the demand. So, factoring in this information, how much will it cost to operate at that point?

Try writing a Python program that will display answers to the following questions:

How much will it cost to operate one server per day?

How much will it cost to operate one server per month?

How much will it cost to operate twenty servers per day?

How much will it cost to operate twenty servers per month?

How many days can I operate one server with \$1,836?

Solution

Code Listing 95

```
# The cost of one server per hour.
Cost_per_hour = 1.02

# Compute the costs for one server.
Cost_per_day = 24 * cost_per_hour
cost_per_month = 30 * cost_per_day

# Compute the costs for twenty servers
cost_per_day_twenty = 20 * cost_per_day
cost_per_month_twenty = 20 * cost_per_month

# Budgeting
budget = 1836
operational_days = budget / cost_per_day

# Display the results.
Print('Cost to operate one server per day is ${:.2f}'.format(cost_per_day))
print('Cost to operate one server per month is ${:.2f}'.format(cost_per_month))
```

```
print('Cost to operate twenty servers per day is  
${:.2f}'.format(cost_per_day_twenty))  
  
print('Cost to operate twenty servers per month is  
${:.2f}'.format(cost_per_month_twenty))  
  
print('A server can operate on a ${0:.2f} budget for {1:.0f} days.'.format(budget,  
operational_days))
```

Output:

Code Listing 96

```
Cost to operate one server per day is $24.48.  
Cost to operate one server per month is $734.40.  
Cost to operate twenty servers per day is $489.60.  
Cost to operate twenty servers per month is $14688.00.  
A server can operate on a $1836.00 budget for 75 days.
```

Chapter 3 Booleans and Conditionals

A Boolean is a specific data type that is only capable of having one of two possible values: **True** or **False**. Another way to think of a Boolean is to consider it either on or off. To assign a Boolean to a variable use **variable_name = boolean**, where **boolean** is either **True** or **False**. Do not use quotes around **True** or **False**. Remember, quotes should only be used for strings.

Code Listing 97

```
the_true_boolean = True
the_other_boolean = False
print(the_true_boolean)
print(the_other_boolean)
```

Output:

Code Listing 98

```
True
False
```

Comparators

The following chart lists six operators that compare one numeric value with another and will result in a Boolean.

Table 2: Comparison Operators

Operator	Description
<code>==</code>	Equal to
<code>></code>	Greater than

Operator	Description
>=	Greater than or equal
<	Less than
<=	Less than or equal
!=	Not equal

When you see `1 == 2` you can think “Is 1 equal to 2?” If the answer is yes, then it is **True**. If the answer is no, then it is **False**. In the following example the answer is no, so the condition will be **False**. Note that while `=` assigns a value to a variable, `==` performs a comparison.

Code Listing 99

```
is_two_equal_to_three = 2 == 3
print(is_two_equal_to_three)
```

Output:

Code Listing 100

```
False
```

Let’s try running the numbers **1** and **2** through all six comparators interactively within the Python interpreter.

Code Listing 101

```
>>> 1 == 2
False
>>> 1 > 2
False
```

```
>>> 1 >= 2
False
>>> 1 < 2
True
>>> 1 <= 2
True
>>> 1 != 2
True
```

Boolean Operators

Keep in mind that Boolean logic is used extensively in the field of computer programming. There are only three Boolean operators: **and**, **or**, and **not**. Each one of them can be used to compare two conditions or negate a condition. Like comparators, they will result in a Boolean.

Table 3: Boolean Logic Operators

Operator	Description
and	Evaluates to True if both statements are true. Otherwise evaluates to False .
or	Evaluates to True if either of the statements is true. Otherwise evaluates to False .
Not	Evaluates to the opposite of the statement.

The following is a truth table that clearly explains Boolean operators and their outcomes.

Code Listing 102

```
True and True is True
True and False is False
```

```
False and True is False
False and False is False

True or True is True
True or False is True
False or True is True
False or False is False

Not True is False
Not False is True
```

Let's take a moment and evaluate two statements using the Boolean **and** operator. The first statement is **43 > 29** and it evaluates to **True**. The second statement is **43 < 44** and it also evaluates to **True**. **43 > 29 and 43 < 44** evaluates to **True** because **True and True** evaluates to **True**.

Code Listing 103

```
>>> 43 > 29
True
>>> 43 < 44
True
>>> 43 > 29 and 43 < 44
True
>>>
```

What is the result of **43 > 29 or 43 < 44**?

Code Listing 104

```
>>> 43 > 29 or 43 < 44
True
```

The **not** Boolean operator will evaluate to the reverse of the statement. Since **43 > 29** is **True**, **not 43 > 29** is **False**.

Code Listing 105

```
>>> 43 > 29
True
>>> not 43 > 29
False
```

The order of operations for Boolean operators is:

1. **not**
2. **and**
3. **or**

Just as an example, **True and False or not False** is **True**. First, **not False** is assessed and is **True**. Next, **True and False** is calculated and is **False**. Finally, **True or False** is evaluated and is **True**.

Code Listing 106

```
>>> not False
True
>>> True and False
False
>>> True or False
True
>>> True and False or not False
True
```

To control the order of operations, use parentheses. Anything surrounded by parentheses will be evaluated first and as its own independent unit. **True and False or not False** is the same as **(True and False) or (not False)**. It's also the same as **((True and False) or (not False))**. Using parentheses will allow you to avoid memorizing the order of operations, and more importantly ensure that your intentions are explicit and clear.

Conditionals

The **if** statement will evaluate a Boolean expression and if it is **True** the code associated with it will be executed. Let's look at the following example to see this demonstrated.

Code Listing 107

```
if 43 < 44:  
    print('Forty-three is less than forty-four.')
```

Output:

Code Listing 108

```
Forty-three is less than forty-four.
```

Because the Boolean expression **43 < 44** is **True**, the code indented under the **if** statement will be executed. This indented code is referred to as a code block. Any statements that are the same distance to the right will also belong to that code block. A code block can contain one or more lines, and its ending will be marked by a line that is less indented than the current code block. Also, keep in mind that code blocks can be nested. The following code listing is a logical view of code blocks.

Code Listing 109

```
Block One  
    Block Two  
    Block Two  
        Block Three  
Block One  
Block One
```

In most cases code blocks are indented using four spaces, but this occurs more out of sense of convention and is not strictly enforced. Python will allow you to use other levels of indentation within your programs. For example, while using four spaces is the most popular option for indentation, using two spaces is the next most popular choice. It is important however that whichever you choose you use it consistently. If you make the decision to use two spaces for indentation, then continue to use two spaces throughout the entire program. When in doubt though, it is always best to follow established conventions unless you have a particular reason not to do so. Also, if you encounter the following error, it means you have a problem within your spacing.

```
IndentationError: expected an indented block
```

Now let's get back to the **if** statement. Notice that the line containing the **if** statement will always end with a colon.

```
age = 32
if age >= 35:
    print('You are old enough to be the President.')

print('Have a nice day!')
```

Output:

```
Have a nice day!
```

In this example, since **age >= 35** is **False**, the Python code indented underneath the **if** statement was not executed. The final **print** function will always execute because it exists outside of the **if** statement. Notice that it is not indented.

The **if** statement can also be paired with **else**. The code indented under **else** will execute in instances where the **if** statement is false. Try to think of the **if/else** statement as meaning, “If the statement is true, run the code underneath **if**. Otherwise run the code underneath **else**.”

```
age = 32
if age >= 35:
    print('You are old enough to be the President.')
else:
```

```
print('You are not old enough to be the President.')

print('Have a nice day!')
```

Output:

Code Listing 114

```
You are not old enough to be the President.

Have a nice day!
```

You can also evaluate multiple conditions by using **elif**, which is short for “else if.” Such as in the case of **if** and **else**, you want to end the line of the **elif** statement with a colon, as well as indent the code to execute underneath it.

Code Listing 115

```
age = 32

if age >= 35:
    print('You are old enough to be a Senator or the President.')
elif age >= 30:
    print('You are old enough to be a Senator.')
else:
    print('You are not old enough to be a Senator or the President.')

print('Have a nice day!')
```

Output:

Code Listing 116

```
You are old enough to be a Senator.

Have a nice day!
```

In this instance, since `age >= 35` is **False**, the code underneath the **if** statement was not executed. Since `age >= 30` is **True**, the code underneath **elif** did execute. The code under **else** will only execute in instances where all of the preceding **if** and **elif** statements evaluate to **False**. Also, the first **if** or **elif** statement to evaluate to **True** will execute, with any remaining **elif** or **else** blocks not executing. The following listing is a final example to illustrate the points explained previously.

Code Listing 117

```
age = 103
if age >= 35:
    print('You are old enough to be a Representative, Senator, or the President.')
elif age >= 30:
    print('You are old enough to be a Senator.')
elif age >= 25:
    print('You are old enough to be a Representative.')
else:
    print('You are not old enough to be a Representative, Senator, or the President.')

print('Have a nice day!')
```

Output:

Code Listing 118

```
You are old enough to be a Representative, Senator, or the President.
Have a nice day!
```

Review

Booleans are always either **True** or **False**.

Comparators contrast one numeric value with another and will result in a Boolean.

Boolean operators (**and**, **or**, **not**) either compare or negate two conditions and will result in a Boolean.

Parentheses can be utilized to control the order of operations.

A code block is marked by a section of code at the same level of indentation.

Conditional keywords include `if`, `if/else`, and `if/elif/else`.

Exercises

Walk, Drive, or Fly

Try creating a program that will ask the user how far they wish to travel. If they express a desire to travel less than three miles, have the program tell them to walk. If they desire to travel more than three miles, but less than three hundred miles, advise them that they should drive. In any instance where they want to travel three hundred or more miles, tell them to fly.

Sample output:

Code Listing 119

```
What distance are you traveling in miles? 3125  
I suggest flying to your destination.
```

Solution

Code Listing 120

```
# Ask for the distance.  
Distance = input(' What distance are you traveling in miles? ')  
  
# Convert the distance into an integer.  
Distance = int(distance)  
  
# Determine what transportation to use.  
if distance < 3:  
    transportation = 'walking'
```

```
elif distance < 300:
    transportation = 'driving'
else:
    transportation = 'flying'

# Display the result.
Print('I suggest {} to your destination.'.format(transportation))
```

Resources

Built-in Types: <https://docs.python.org/3/library/stdtypes.html>

Order of Operations (PEMDAS): <http://www.purplemath.com/modules/orderops.htm>

Style Guide for Python Code (PEP 8): <http://legacy.python.org/dev/peps/pep-0008/>

Chapter 4 Functions

Among computer programmers there is an important concept known as DRY: Don't Repeat Yourself. Instead of repeating several lines of code every time you want to perform a particular task, use a function that will allow you to write a block of Python code once and then use it many times. This can help reduce the overall length of your programs, as well as give you a single place to change, test, troubleshoot, and document any given task. Ultimately, this makes your application much easier to maintain in the long run.

To create a function, use the **def** keyword followed by the name of the function. Always follow the function name with a set of parentheses. If your function accepts parameters you may include the names of those parameters within the parentheses, separating them with commas. Finally, conclude the function definition line with a colon. The code block that follows the function definition will be executed any time the function is called. The pattern is **def function_name():**. The following is a very simple function.

Code Listing 121

```
def say_hello():  
    print('Hello!')
```

If you were to attempt to execute this code, no output would be presented because the function is defined but never called. When calling a function you must ensure that you include the parentheses.

Code Listing 122

```
def say_hello():  
    print('Hello!')
```



```
say_hello()
```

Output:

Code Listing 123

```
Hello!
```

A function will have to be defined before it can be called. Define your functions at the top of your Python program. Here you'll see what will happen if you try to use a function that has not yet been defined.

Code Listing 124

```
say_hello()

def say_hello():
    print('Hello!')
```

Output:

Code Listing 125

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

Now let's extend the function so that it accepts a parameter. Try to think of parameters as variables that can be used inside of the function. The pattern is **def function_name(parameter_name):**.

Code Listing 126

```
def say_hello(name):
    print('Hello {}'.format(name))

say_hello('Erin')
say_hello('everybody')
```

Output:

Code Listing 127

```
Hello Erin!
Hello everybody!
```

Once you've defined a parameter, the function will expect and require a value for that parameter. The actual value specified for a function parameter when the function is invoked is sometimes called the argument. If one is not provided you will encounter an error.

Code Listing 128

```
def say_hello(name):  
    print('Hello {}!'.format(name))  
  
say_hello()
```

Output:

Code Listing 129

```
File "say_hello.py", line 4, in <module>  
    say_hello()  
TypeError: say_hello() missing 1 required positional argument: 'name'
```

If you'd like to make the parameter optional, set a default value for it by using the equals sign. The pattern is **def function_name(parameter_name = default_value):**.

Code Listing 130

```
def say_hello(name = 'there'):  
    print('Hello {}!'.format(name))  
  
say_hello()  
say_hello('Erin')
```

Output:

Code Listing 131

```
Hello there!
```

```
Hello Erin!
```

Keep in mind that functions are capable of accepting multiple parameters. All you need to do is include them within the parentheses of the function definition, and separate them with a comma. When you are calling the function, always supply the arguments and separate them with commas as well.

Code Listing 132

```
def say_hello(first, last):  
    print('Hello {} {}!'.format(first, last))  
  
say_hello('Josiah', 'Carberry')
```

Output:

Code Listing 133

```
Hello Josiah Carberry!
```

When parameters are accepted by a function they can also be called positional parameters. This is because their order is important. Notice here that **Josiah** was assigned to **first** while **Carberry** was assigned to **last**. You can also explicitly pass values into a function by name. Whenever you are calling the function make sure to supply the parameter name, followed by the equal sign, and then the value for that parameter. When using named parameters, order is not important. Here's an example.

```
Def say_hello(first, last):  
    print('Hello {} {}!'.format(first, last))  
  
say_hello(first = 'Josiah', last = 'Carberry')  
say_hello(last = 'Carberry', first = 'Hank')
```

Output:

Code Listing 134

```
Hello Josiah Carberry!  
Hello Hank Carberry!
```

It is also possible to combine required and optional parameters as in the following example. If you use both required and optional parameters, the required parameters should come first.

Code Listing 135

```
def say_hello(first, last='Carberry'):  
    print('Hello {} {}'.format(first, last))  
  
say_hello('Josiah')  
say_hello('Hank', 'Mobley')
```

Output:

Code Listing 136

```
Hello Josiah Carberry!  
Hello Hank Mobley!
```

Often, the first statement of a function is a documentation string, or docstring for short. You can create a docstring by surrounding text with three double quotes. This docstring will offer a brief summary of the function. When writing the docstring make sure to ask yourself, “What does this function do?” or “Why does this function exist?” You can access this docstring by using the built-in **help()** function. Pass the name of the function you want more information about to **help()**. Type **q** to exit the help screen.

Code Listing 137

```
def say_hello(first, last='Carberry'):  
    """Say hello."""  
    print('Hello {} {}'.format(first, last))
```

```
help(say_hello)
```

Output:

Code Listing 138

```
Help on function say_hello in module __main__:
```

```
say_hello(first, last='Carberry')
```

```
    Say hello.
```

Not only are functions capable of performing a task, they can also return data by using the **return** statement. This statement makes it possible for you to return any data type that you require. Once the **return** statement is called, no further code in the function will be executed. The following code is a function that returns a string.

Code Listing 139

```
def even_or_odd(number):  
    """Determine if a number is odd or even."""  
    if number % 2 == 0:  
        return 'Even'  
    else:  
        return 'Odd'  
  
even_or_odd_string = even_or_odd(9)  
print(even_or_odd_string)
```

Output:

Code Listing 140

```
Odd
```

See the following example for a similar function that returns a Boolean.

Code Listing 141

```
def is_odd(number):  
    """Determine if a number is odd."""  
    if number % 2 == 0:  
        return False  
    else:  
        return True  
  
print(is_odd(9))
```

Output:

Code Listing 142

```
True
```

It is entirely possible for you to create functions that call other functions. The following listing is an example.

Code Listing 143

```
def get_name():  
    """Get and return a name"""  
    name = input('What is your name? ')  
    return name  
  
def say_name(name):  
    """Say a name"""  
    print('Your name is {}'.format(name))
```

```
def get_and_say_name():  
    """Get and display name"""  
    name = get_name()  
    say_name(name)  
  
get_and_say_name()
```

Output:

Code Listing 144

```
What is your name? Erin  
Your name is Erin.
```

Review

A function is a block of reusable code that can perform an action as well as optionally return data.

A function can be called only after it has been defined.

A function is defined by the basic pattern: **def function_name(parameter_name):**.

A function is capable of accepting parameters. To make a parameter optional simply supply a default value for that particular parameter.

You can supply a docstring as the first line of your function.

The **return** statement will exit the function and pass back anything that follows **return**.

Use the built-in **help()** function to get assistance with a particular object. When supplying a function to **help()**, the docstring enclosed within the function will be displayed.

Exercises

Fill in the Blank Word Game

This exercise involves creating a fill in the blank word game. Try prompting the user to enter a noun, verb, and an adjective. Use the provided responses to fill in the blanks and then display the story.

First, write a short story. Remove a noun, verb, and an adjective.

Create a function that asks for input from the user.

Create another function that will fill the blanks in the story you've just created.

Ensure that each function contains a docstring.

After the noun, verb, and adjective have been collected from the user, display the story that has been created using their input.

Solution

Code Listing 145

```
def get_word(word_class):
    """Get a word from standard input and return that word."""
    if word_class.lower() == 'adjective':
        article = 'an'
    else:
        article = 'a'
    return input('Enter a word that is {0} {1}: '.format(article, word_class))

def fill_in_the_blanks(noun, verb, adjective):
    """Fills in the blanks and returns a completed story."""
    story = "I never knew anyone that hadn't {1} at least once in their life, except for {2}, old Aunt Polly. She never {1}, not even when that {0} came to town.".format(noun, verb, adjective)
    return story
```

```

def print_story(story):
    """Prints a story."""
    print()
    print('Here is the story you made.  Enjoy!')
    print()
    print(story)

def create_story():
    """Creates a story by collecting the input and printing a finished story."""
    noun = get_word('noun')
    verb = get_word('verb')
    adjective = get_word('adjective')

    the_story = fill_in_the_blanks(noun, verb, adjective)
    print_story(the_story)

create_story()

```

Output:

Code Listing 146

```

Enter a word that is a noun: unicorn
Enter a word that is a verb: hid
Enter a word that is an adjective: vivacious

Here is the story you created.  Enjoy!

```

I never knew anyone that hadn't hid at least once in their life, except for vivacious, old Aunt Polly. She never hid, not even when that unicorn came to town.

Resources

DRY: https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

Documentation for the `help()` built-in function: <https://docs.python.org/3/library/functions.html#help>

Docstring Conventions (PEP 257): <http://legacy.python.org/dev/peps/pep-0257/>

Chapter 5 Lists

In the previous chapters we've addressed string, integer, float, and Boolean data types. In this chapter we'll look at lists, a data type that holds an organized collection of items. The items, or values, that are contained within the list can be various data types themselves. In fact, you can even have lists that exist within lists.

Note that lists are created by using comma separated values between square brackets. The pattern is **list_name = [item_1, item_2, item_N]**. To create an empty list use: **list_name = []**. Items within a list can always be accessed by index. List indices are always zero based, which means that the first item in the list will have an index of 0, the second item an index of 1, and so on. To access any item in a list using an index, simply enclose the index in square brackets directly following the list name. The pattern is **list_name[index]**.

Code Listing 147

```
animals = ['toad', 'lion', 'seal']  
print(animals[0])  
print(animals[1])  
print(animals[2])
```

Output:

Code Listing 148

```
toad  
lion  
seal
```

Keep in mind that not only can you access values by index, you can also set values by index.

Code Listing 149

```
animals = ['toad', 'lion', 'seal']  
print(animals[0])
```

```
animals[0] = 'sheep'  
print(animals[0])
```

Output:

Code Listing 150

```
toad  
sheep
```

Also, you can access items starting at the end of the list by making use of a negative index. The **-1** index will represent the last item on the list, with **-2** representing the second to last item on the list, and so on.

Code Listing 151

```
animals = ['toad', 'lion', 'seal']  
print(animals[-1])  
print(animals[-2])  
print(animals[-3])
```

Output:

Code Listing 152

```
seal  
lion  
toad
```

Adding Items to a List

To add an item to the end of a list use the **append()** method and pass in the item that you wish to add.

Code Listing 153

```
animals = ['toad', 'lion', 'seal']  
animals.append('fox')  
print(animals[-1])
```

Output:

Code Listing 154

```
fox
```

If you wish to add multiple items to the end of a list, use the **extend()** method. The **extend()** method takes a list. You pass in a list by name or create one by surrounding a list of items within brackets.

Code Listing 155

```
animals = ['toad', 'lion', 'seal']  
animals.extend(['fox', 'owl'])  
print(animals)  
  
more_animals = ['whale', 'elk']  
animals.extend(more_animals)  
print(animals)
```

Output:

Code Listing 156

```
['toad', 'lion', 'seal', 'fox', 'owl']  
['toad', 'lion', 'seal', 'fox', 'owl', 'whale', 'elk']
```

It is also possible to add a single item at any point in the list simply by making use of the **insert()** method. Pass in the index where you want to add the item, follow it with a comma, and then the item itself. All of the current items in the list will be moved over by one.

```
animals = ['toad', 'lion', 'seal']
animals.insert(0, 'whale')
print(animals)

animals.insert(2, 'owl')
print(animals)
```

Output:

```
['whale', 'toad', 'lion', 'seal']
['whale', 'toad', 'owl', 'lion', 'seal']
```

Slices

To access a selected portion of a list, which can be referred to as a slice, specify two indices and then separate them with a colon placed within brackets. The slice will begin at the first index and go up to, but not include, the very last index. If the first index is omitted then `0` is assumed. If the second index is omitted the number of items in the list is implied.

```
animals = ['toad', 'lion', 'seal', 'fox', 'owl', 'whale']

some_animals = animals[1:4]
print('Some animals:      {}'.format(some_animals))

first_two = animals[0:2]
print('First two animals: {}'.format(first_two))
```

```
first_two_again = animals[:2]
print('First two animals: {}'.format(first_two_again))

last_two = animals[4:6]
print('Last two animals: {}'.format(last_two))

last_two_again = animals[-2:]
print('Last two animals: {}'.format(last_two_again))
```

Output:

Code Listing 160

```
Some animals:      ['lion', 'seal', 'fox']
First two animals: ['toad', 'lion']
First two animals: ['toad', 'lion']
Last two animals:  ['owl', 'whale']
Last two animals:  ['owl', 'whale']
```

String Slices

It is possible to use slices with strings. Just think of a string as a list of characters.

Code Listing 161

```
part_of_a_whale = 'whale'[1:3]
print(part_of_a_whale)
```

Output:

```
ha
```

Finding an Item in a List

Keep in mind that the `index()` method will accept a value as a parameter and then return the index of the first value on the list. For example, if there were two incidences of **lion** in the **animals** list, then **`animals.index('lion')`** would return the index of the first occurrence of **lion**. If the value is not discovered on the list, then Python will raise an exception.

Code Listing 163

```
animals = ['toad', 'lion', 'seal']  
lion_index = animals.index('lion')  
print(lion_index)
```

Output:

Code Listing 164

```
1
```

Exceptions

An exception is most often a clear indication that something has either unexpectedly occurred, or has just generally gone wrong within your program. If you don't account for, or handle exceptions within, your program, Python will print out a message which explains the exception, as well as halt the execution of the program. The following is an example of an exception that hasn't been handled.

Code Listing 165

```
animals = ['toad', 'lion', 'seal']  
sheep_index = animals.index('sheep')  
print(sheep_index)
```

Output:

Code Listing 166

```
Traceback (most recent call last):  
  File "exception_example.py", line 2, in <module>  
    sheep_index = animals.index('sheep')  
ValueError: 'sheep' is not in list
```

These messages that Python provides can be a valuable resource for correcting mistakes that exist within your code. As you can see from the previous example, Python clearly displayed the line number as well as the code that caused the exception.

A key thing to remember though is that you often need to prevent Python from exiting whenever it encounters an exception. In order to avoid this you need to tell your program what it should do whenever it encounters an exception. Do this by surrounding any code you think may raise an exception in a **try/except** block. Let's update the previous example with a **try/except** block.

Code Listing 167

```
animals = ['toad', 'lion', 'seal']  
  
try:  
    sheep_index = animals.index('sheep')  
except:  
    sheep_index = 'No sheep found.'  
  
print(sheep_index)
```

Output:

Code Listing 168

```
No sheep found.
```

If any exception is raised while you are executing the code in the **try:** code block, the code in the **except:** code block will be executed. If no exception is met in the **try:** code block, the code in the **except:** code block is omitted and will not be executed.

Looping through a List

If you are looking to perform an action on every item in a list, use a **for** loop. The pattern you will use is **for item_variable in list_name:**. Like **if** statements and function definitions, the **for** statement will always end in a colon. The code block that follows the **for** statement will be executed for every item within the list. Effectively what happens is that the first item in the list, **list[0]**, is assigned to **item_variable** and the code block is executed. The next item in the list, **list[1]**, is assigned to **item_variable** and the code block is executed. This process lasts until the list is finished. If there are no items in the list, the code block will not be executed.

The following example prints the uppercase version of every item in the **animals** list.

Code Listing 169

```
animals = ['toad', 'lion', 'seal']  
  
for animal in animals:  
    print(animal.upper())
```

Output:

Code Listing 170

```
TOAD  
  
LION  
  
SEAL
```

In addition to the **for** loop, Python also has a **while** loop. The pattern is **while condition:** followed by a code block. As long as this condition evaluates to true, the code block following the **while** statement will be executed. Typically, the code block will modify a variable that is part of the condition. At some point the condition will evaluate to false and the program continues after the **while** loop. In cases where the condition never evaluates to false it will become an infinite loop. To halt the execution of a Python program press Ctrl+C. Ctrl+C will allow you an out if you accidentally create an infinite loop, breaking you out of your program.

The following example creates an **index** variable to store an integer, and will be employed as the index of the **animals** list. The **while** loop will execute when the index is less than the length of the **animals** list. During the code block the index variable will be incremented by one. The plus-equals operator adds a value to the variable's existing value, and assigns the new value to that variable. Using **index += 1** will increment the index variable by one. Note that unlike many programming languages, Python does not have a **++** increment operator.

Code Listing 171

```
animals = ['toad', 'lion', 'seal', 'fox', 'owl', 'whale', 'elk']

index = 0

while index < len(animals):
    print(animals[index])
    index += 1
```

Output:

Code Listing 172

```
toad
lion
seal
fox
owl
whale
elk
```

Sorting a List

To sort a list, call the **sort()** method on the list, making sure to avoid using any arguments. This will reorder the current list. If you wish to create a new list, simply use the built-in **sorted()** function and provide a list as an argument.

Code Listing 173

```
animals = ['toad', 'lion', 'seal']
```

```
sorted_animals = sorted(animals)
print('Animals list:          {}'.format(animals))
print('Sorted animals list:   {}'.format(sorted_animals))
animals.sort()
print('Animals after sort method: {}'.format(animals))
```

Output:

Code Listing 174

```
Animals list:          ['toad', 'lion', 'seal']
Sorted animals list:   ['lion', 'seal', 'toad']
Animals after sort method: ['lion', 'seal', 'toad']
```

List Concatenation

To concatenate or combine two or more lists, use the plus sign.

Code Listing 175

```
animals = ['toad', 'lion', 'seal']
more_animals = ['fox', 'owl', 'whale']
all_animals = animals + more_animals
print(all_animals)
```

Output:

Code Listing 176

```
['toad', 'lion', 'seal', 'fox', 'owl', 'whale']
```

To determine the number of items on a list, use the built-in **len()** function and pass in a list.

Code Listing 177

```
animals = ['toad', 'lion', 'seal']  
print(len(animals))  
animals.append('fox')  
print(len(animals))
```

Output:

Code Listing 178

```
3  
4
```

Ranges

Another important built-in function is the **range()** function, which creates a list of numbers and is often combined with the **for** statement. This function is useful when you want to complete an action a given number of times, or when you want to have access to the index of a list.

The **range()** function necessitates at least one parameter that will denote a stop. By default, **range()** produces a list that begins at zero and continues up to, but not including, the stop. To generate a list that encompasses **N** items, pass **N** to **range()** like so: **range(N)**. For example, if you want to get a list of 4 items use **range(4)**. The list starts at zero and will contain the numbers **0**, **1**, **2**, and **3**.

Code Listing 179

```
for number in range(4):  
    print(number)
```

Output:

Code Listing 180

```
0
```

```
1  
2  
3
```

It is possible for you to define the start as well as the stop. The pattern is **range(start, stop)**. To begin a list at 2 and stop at 4, use **range(2, 4)**. This will produce a list that is made up of only two items, 2 and 3.

Code Listing 181

```
for number in range(2, 4):  
    print(number)  
2  
3
```

In addition to the start and stop parameters, the **range()** function is also capable of accepting a step parameter. In cases where all three parameters are being utilized, the list will begin at the start value, cease just before the stop value, and increment the list by the step value. If there is no step value specified, as in the previous examples, its default value is **1**. Let's try generating a list that incorporates all of the even numbers from 0 to 8.

Code Listing 182

```
for number in range(0, 10, 2):  
    print(number)
```

Output:

Code Listing 183

```
0  
2  
4  
6  
8
```

The following is an example of using the **range()** function in conjunction with a list to print every other item in that list.

Code Listing 184

```
animals = ['toad', 'lion', 'seal', 'fox', 'owl', 'whale', 'elk']
for number in range(0, len(animals), 2):
    print(animals[number])
```

Output:

Code Listing 185

```
toad
seal
owl
elk
```

Review

Lists can be made using square brackets to enclose comma separated values. The pattern is **list_name = [item_1, item_2, item_N]**.

An index can be used to access items in a list. List indices are zero based. The pattern is **list_name[index]**.

Use negative indices to access items from the end of the list. The last item in a list is **list_name[-1]**.

Items can be added to a list by using the **append()** or **extend()** list methods.

A slice allows you to access a portion of a list. The pattern is **list_name(start, stop)**

The list **index()** method will accept a value as a parameter and return the index of the first value in the list, or create an exception if the value is not found within the list. The pattern is **list_name.index(value)**.

Loop through a list by utilizing a **for** loop. The pattern is **for item_variable in list_name:** followed by a code block.

The code block in a **while** loop will execute as long as the condition evaluates to true. The pattern is **while condition:** followed by a code block.

To sort a list, use the built-in **sorted()** function or the **sort()** list method.

The built-in **range()** function will produce a list of numbers. The pattern is **range(start, stop, step)**.

Unhandled exceptions will cause Python programs to terminate. You can avoid this by handling exceptions using **try/except** blocks.

Exercises

Grocery List

Try creating a Python program that will capture and display a person's grocery shopping list. Make it so the program will continually prompt the user for another item until the point where they enter a blank item. After all the items have been entered, try displaying the shopping list back to the user.

Sample output:

Code Listing 186

```
Enter an item for your grocery list. Press <ENTER> when done: bread
Item added.
Enter an item for your grocery list. Press <ENTER> when done: milk
Item added.
Enter an item for your grocery list. Press <ENTER> when done: coffee
Item added.
Enter an item for your grocery list. Press <ENTER> when done:

Your Grocery List:
bread
milk
coffee
```

Solution

Code Listing 187

```
# Create a list to hold the grocery items.
Grocery_list = []
finished = False
while not finished:
    item = input('Enter an item for your grocery list. Press <ENTER> when done:
    ')
    if len(item) == 0:
        finished = True
    else:
        grocery_list.append(item)
        print('Item added.')

# Display the grocery list.
Print()
print('Your Grocery List:')
print('-' * 18)
for item in grocery_list:
    print(item)
```

Resources

Data Structures (Lists): <https://docs.python.org/3/tutorial/datastructures.html>

Exceptions: <https://docs.python.org/3/library/exceptions.html>

For Loops: <https://wiki.python.org/moin/ForLoop>

Handling Exceptions: <https://wiki.python.org/moin/HandlingExceptions>

Sorted: <https://docs.python.org/3/library/functions.html#sorted>

While Loops: <https://wiki.python.org/moin/WhileLoop>

Chapter 6 Dictionaries

Another data type in Python is a dictionary. A dictionary holds key-value pairs, which are referred to as items. You will hear dictionaries referred to in different ways including: associative arrays, hashes, or hash tables.

Dictionaries are generated using comma separated items surrounded by curly braces. The item begins with a key, followed by a colon, and concluded with a value. The pattern is **dictionary_name = {key_1: value_1, key_N: value_N}**. In order to create an empty dictionary, use **dictionary_name = {}**.

Any items in a dictionary can be retrieved by key. To do so, enclose the key within brackets immediately following the dictionary name. The pattern is **dictionary_name[key]**.

Code Listing 188

```
contacts = {'David': '555-0123', 'Tom': '555-5678'}
davids_phone = contacts['David']
toms_phone = contacts['Tom']

print('Dial {} to call David.'.format(davids_phone))
print('Dial {} to call Tom.'.format(toms_phone))
```

Output:

Code Listing 189

```
Dial 555-0123 to call David.
Dial 555-5678 to call Tom.
```

Not only are you able to access values by key, you can also set values by key. The pattern is **dictionary_name[key] = value**.

```
Contacts = {'David': '555-0123', 'Tom': '555-5678'}
contacts['David'] = '555-0000'
```

```
davids_phone = contacts['David']  
print('Dial {} to call David.'.format(davids_phone))
```

Output:

Code Listing 190

```
Dial 555-0000 to call David.
```

Adding Items to a Dictionary

Keep in mind that you can easily add new items to a dictionary through the process of assignment. The pattern for this is **dictionary_name[new_key] = value**. In order to determine the number of items in a dictionary, first use the **len()** built-in function and pass in a dictionary.

Code Listing 191

```
contacts = {'David': '555-0123', 'Tom': '555-5678'}  
contacts['Nora'] = '555-2413'  
print(contacts)  
print(len(contacts))
```

Output:

Code Listing 192

```
{'Nora': '555-2413', 'Tom': '555-5678', 'David': '555-0123'}  
3
```

Removing Items from a Dictionary

To remove an item from a dictionary, use the **del** statement. The pattern is **del dictionary_name[key]**.

Code Listing 193

```
contacts = {'David': '555-0123', 'Tom': '555-5678'}  
del contacts['David']  
print(contacts)
```

Output:

Code Listing 194

```
{'Tom': '555-5678'}
```

Keep in mind that the values within a dictionary do not have to be of the same data type. In the following example you'll see that while the value for the **David** key is a list, the value for the **Tom** key is a string.

Code Listing 195

```
contacts = {  
    'David': ['555-0123', '555-0000'],  
    'Tom': '555-5678'  
}  
print('David:')  
print(contacts['David'])  
print('Tom:')  
print(contacts['Tom'])
```

Output:

Code Listing 196

```
David:  
['555-0123', '555-0000']
```

Tom:

555-5678

When you are assigning the items to the **contacts** dictionary you can use additional spaces as it will greatly improve readability

As a result of the fact **dictionary_name('key_name')** is capable of storing its associated value, you can act upon it like you would the actual values. To illustrate this, let's use a **for** loop for all of David's phone numbers.

Code Listing 197

```
contacts = {  
    'David': ['555-0123', '555-0000'],  
    'Tom': '555-5678'  
}  
  
for number in contacts['David']:  
    print('Phone: {}'.format(number))
```

Output:

Code Listing 198

Phone: 555-0123

Phone: 555-0000

Finding a Key in a Dictionary

If you would like to find out whether a certain key exists within a dictionary, use the **value in dictionary_name.keys()** syntax. If the value is in fact a key in the dictionary, **True** will be returned. If it is not, then **False** will be returned.

```
contacts = {  
    'David': ['555-0123', '555-0000'],  
    'Tom': '555-5678'  
}  
  
if 'David' in contacts.keys():  
    print("David's phone number is:")  
    print(contacts['David'][0])  
  
if 'Nora' in contacts.keys():  
    print("Nora's phone number is:")  
    print(contacts['Nora'][0])
```

Output:

```
David's phone number is:  
555-0123
```

Take note that **'David' in contacts** evaluates to **True**, so the code block which follows the **if** statement will be executed. Since **'Nora' in contacts** evaluates to **False**, the code block which follows that statement will not execute. Also, since **contacts['David']** holds a list, you can act on it as a list. Accordingly, **contacts['David'][0]** will return the first value in the list.

Finding a Value in a Dictionary

Using the **values()** dictionary method returns a list of values within the dictionary. Use the **value in list** syntax to determine if the value actually exists within the list. If the value is in the list, **True** will be returned. Otherwise **False** will be returned.

```
contacts = {  
    'David': ['555-0123', '555-0000'],  
    'Tom': '555-5678'  
}  
  
print ('555-5678' in contacts.values())
```

Output:

```
True
```

Looping through a Dictionary

If you are looking to loop through items in a dictionary, one pattern you can use is **for key_variable in dictionary_name:**. The code block that follows after the **for** statement will then be executed for every item listed in the dictionary. To access the value of the item in the **for** loop, use the **dictionary_name[key_variable]** pattern. Unlike lists, dictionaries are unordered. The **for** loop will ensure that all of the items in the dictionary will be processed; however, there is absolutely no guarantee that they will be processed in the order you desire.

It is a common practice to name dictionaries by using a plural noun, such as in the case of contacts. The standard pattern of the **for** loop will use the singular form of the dictionary name as the key variable. For example, **for contact in contacts** or **for person in people**.

```
contacts = {  
    'David': '555-0123',  
    'Tom': '555-5678'  
}  
  
for contact in contacts:
```

```
print('The number for {0} is {1}.'.format(contact, contacts[contact]))
```

Output:

Code Listing 204

```
The number for Tom is 555-5678.
```

```
The number for David is 555-0123.
```

You may also opt to utilize two variables when defining a **for** loop in order to process items within a dictionary. While the first variable comprises the key, the second one will contain the value. The pattern is **for key_variable, value_variable in dictionary_name.items():**.

Code Listing 205

```
contacts = {'David': '555-0123', 'Tom': '555-5678'}  
  
for person, phone_number in contacts.items():  
    print('The number for {0} is {1}.'.format(person, phone_number))
```

Output:

Code Listing 206

```
The number for Tom is 555-5678.
```

```
The number for David is 555-0123.
```

Nesting Dictionaries

Since the values contained in a dictionary can be of any data type you have the ability to nest dictionaries. In the following example, names are the keys for the **contacts** dictionary, while **phone** and **email** are the keys used within the nested dictionary. Each individual in this contact list has both a phone number and an email address. If you want to know David's email address you can retrieve that information using **contacts['David']['email']**.

Make sure to pay close attention to the location of colons, quotation marks, commas, and braces. Try using additional white space when you are coding these types of data structures to help visually represent the data structure.

```
contacts = {  
    'David': {  
        'phone': '555-0123',  
        'email': 'david@gmail.com'  
    },  
    'Tom': {  
        'phone': '555-5678',  
        'email': 'tom@gmail.com'  
    }  
}  
  
for contact in contacts:  
    print("{}'s contact info:".format(contact))  
    print(contacts[contact]['phone'])  
    print(contacts[contact]['email'])
```

Output:

```
Tom's contact info:  
555-5678  
tom@gmail.com  
David's contact info:  
555-0123  
david@gmail.com
```

Review

Dictionaries hold key-value pairs, known as items. **Dictionary_name = {key_1: value_1, key_N: value_N}**

A key allows you to access the values stored in a dictionary. **Dictionary_name[key]**

Assignments allow you to add or change values in a dictionary. **Dictionary_name[key] = value**

The **del** statement removes items from a dictionary. **Del dictionary_name[key]**

To determine if a key exists within a dictionary, use the **value in dictionary_name** syntax, which will return a Boolean.

The **values()** dictionary method will return a list of the values that are stored in that dictionary.

Loop through a dictionary using the **for key_variable in dictionary_name:** syntax.

Dictionary values can be made up of any data type, including other dictionaries.

Exercises

Interesting Facts

Try to create a dictionary that has a listing of people and includes one interesting fact about each of them. Display each person and their interesting fact on the screen. From there, alter a fact about one of the people on the list. Also, add an extra person and corresponding fact to the list. Display the newly created list of people and facts. Try running the program multiple times and take note of whether the order changes.

Sample output:

Code Listing 209

```
Jeff: Was born in France.  
David: Was a mascot in college.  
Anna: Has arachnophobia.  
  
Dylan: Has a pet hedgehog.  
Jeff: Was born in France.
```

David: Can juggle.

Anna: Has arachnophobia.

Solution

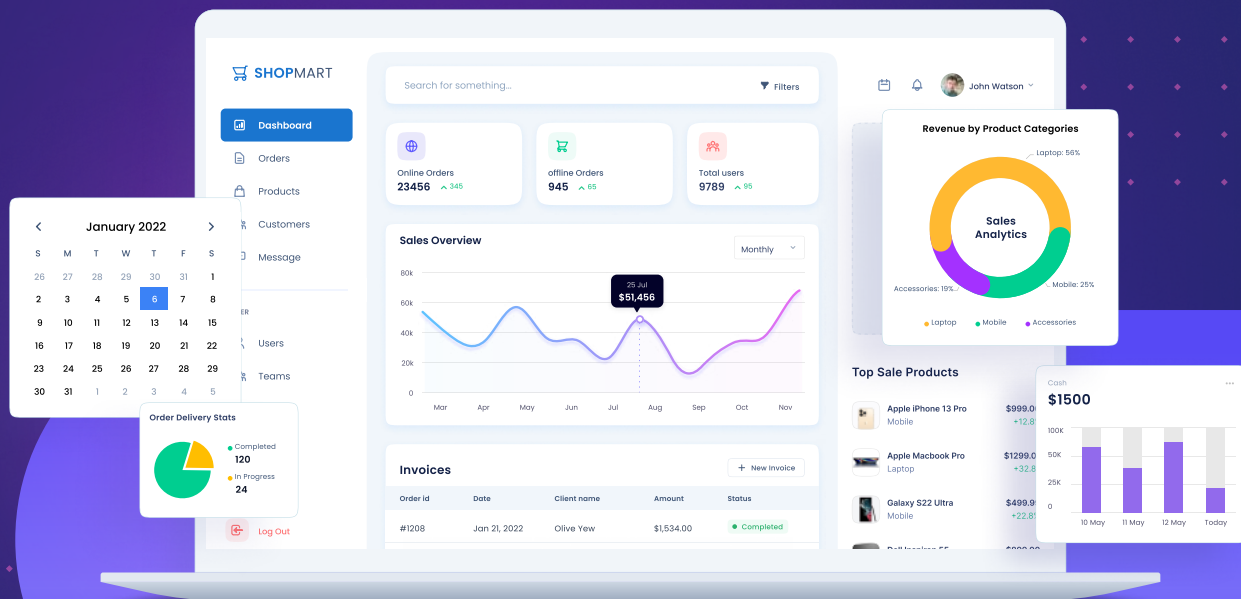
Code Listing 210

```
def display_facts(facts):  
    """Displays facts"""  
    for fact in facts:  
        print('{}: {}'.format(fact, facts[fact]))  
    print()  
  
facts = {  
    'David': 'Was a mascot in college.',  
    'Jeff': 'Was born in France.',  
    'Anna': 'Has arachnophobia.'  
}  
  
display_facts(facts)  
  
facts['David'] = 'Can juggle.'  
facts['Dylan'] = 'Has a pet hedgehog.'  
  
display_facts(facts)
```

Resources

Data Structures (Dictionaries): <https://docs.python.org/3/tutorial/datastructures.html>

THE WORLD'S BEST **UI COMPONENT SUITE** FOR BUILDING POWERFUL APPS



GET YOUR **FREE** .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense



- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies



IBM

SIEMENS



VISA

 Syncfusion

Chapter 7 Tuples

A tuple is an immutable list, meaning that once it is defined it cannot be changed. This is different from normal lists in which you can add, remove, and change the values. With tuples none of these actions are an option. Where tuples are similar to lists is that they are ordered in the same fashion, and the values in the tuple can be still be accessed by index. In fact, you can perform many of the same operations on a tuple that you can on a list. You can concatenate tuples, you can iterate over the values in a tuple with a **for** loop, you can access values from the end of the tuple using negative indices, and you can access slices of a tuple. Tuples are created using comma separated values between parentheses. The pattern is **tuple_name = (item_1, item_2, item_N)**. If you only want a single item in a tuple, that single item must always be followed by a comma. The pattern is **tuple_name = (item_1,)**.

Tuples are key for organizing and holding data that will not, or should not change at any point during the execution of your program. Using a tuple is a great way to ensure that the values are not accidentally altered. For example, the months of the year should not change.

Code Listing 211

```
months_of_the_year = ('January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December')

jan = months_of_the_year[0]

print(jan)

print()

for month in months_of_the_year:
    print(month)

# You cannot modify values in a tuple. This will raise an exception.
Months_of_the_year[0] = 'New January'
```

```
January

January
February
March
April
May
June
July
August
September
October
November
December

Traceback (most recent call last):

  File "tuples.py", line 10, in <module>
    months_of_the_year[0] = 'New January'

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

Even though you are unable to change the values within a tuple, you can always remove the entire tuple during the execution of your program by making use of the previously mentioned **del** statement.

```
months_of_the_year = ('January', 'February', 'March', 'April', 'May', 'June',
                      'July', 'August', 'September', 'October', 'November', 'December')

print(months_of_the_year)
```

```
del months_of_the_year

# This will raise an exception since the tuple was deleted.

Print(months_of_the_year)
```

Output:

Code Listing 214

```
('January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December')
```

Traceback (most recent call last):

```
File "tuples2.py", line 5, in <module>
```

```
    print(months_of_the_year)
```

NameError: name 'months_of_the_year' is not defined

Switching between Tuples and Lists

In order to make a list from a tuple, use the `list()` built-in function and pass in the tuple. To create a tuple from a list, use the `tuple()` built-in function. The built-in function `type()` will display an object's type.

Code Listing 215

```
months_of_the_year_tuple = ('January', 'February', 'March', 'April', 'May',
'June', 'July', 'August', 'September', 'October', 'November', 'December')
```

```
months_of_the_year_list = list(months_of_the_year_tuple)
```

```
print('months_of_the_year_tuple is {}'.format(type(months_of_the_year_tuple)))
```

```
print('months_of_the_year_list is {}'.format(type(months_of_the_year_list)))
```

```
animals_list = ['toad', 'lion', 'seal']
```

```
animals_tuple = tuple(animals_list)
```

```
print('animals_list is {}'.format(type(animals_list)))  
print('animals_tuple is {}'.format(type(animals_tuple)))
```

Output:

Code Listing 216

```
months_of_the_year_tuple is <class 'tuple'>.  
Months_of_the_year_list is <class 'list'>.  
Animals_list is <class 'list'>.  
Animals_tuple is <class 'tuple'>.
```

Looping through a Tuple

If you are looking to perform a particular action on every item within a tuple, use a **for** loop. The pattern is **for item_variable in tuple_name** followed by a code block.

Code Listing 217

```
months_of_the_year = ('January', 'February', 'March', 'April', 'May', 'June',  
                      'July', 'August', 'September', 'October', 'November', 'December')  
  
for month in months_of_the_year:  
    print(month)
```

Output:

Code Listing 218

```
January  
February  
March  
April  
May
```

```
June
July
August
September
October
November
December
```

Tuple Assignment

You can use tuples to assign values to multiple variables at the same time. In the following example, the variables **jan**, **feb**, **mar**, **apr**, **may**, **jun**, **jul**, **aug**, **sep**, **oct**, **nov**, and **dec** are assigned the months of the year from the **months_of_the_year** tuple.

Code Listing 219

```
months_of_the_year = ('January', 'February', 'March', 'April', 'May', 'June',
    'July', 'August', 'September', 'October', 'November', 'December')

(jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec) = months_of_the_year

print(jan)

print(dec)
```

Output:

Code Listing 220

```
January
December
```

It is also possible to use tuple assignment with lists.

```

contact_info = ['555-0123', 'david@gmail.com']
(phone, email) = contact_info
print(phone)
print(email)

```

Output:

```

555-0123
david@gmail.com

```

Tuple assignment can also be used with functions as well. For example, you could create a function that returns a tuple and assigns those values to different variables.

The following example uses the built-in `max()` and `min()` functions. The `max()` built-in function will return the largest item that is passed to it. The `min()` built-in function will return the smallest item that is passed to it.

```

def high_and_low(numbers):
    """Determine the highest and lowest number"""
    highest = max(numbers)
    lowest = min(numbers)
    return (highest, lowest)

lucky_numbers = [37, 71, 47, 13, 17, 67]
(highest, lowest) = high_and_low(lucky_numbers)
print('The highest number is: {}'.format(highest))
print('The lowest number is: {}'.format(lowest))

```

Output:

Code Listing 224

```
The highest number is:71
```

```
The lowest number is: 13
```

You can also use tuple assignment in a **for** loop. In the following example the **contacts** list is comprised of a series of tuples. Each time the **for** loop is performed the variables **name** and **phone** will be populated with the contents of a tuple from the **contacts** list.

Code Listing 225

```
contacts = [('David', '555-0123'), ('Tom', '555-5678')]  
for (name, phone) in contacts:  
    print("{}'s phone number is {}".format(name, phone))
```

Output:

Code Listing 226

```
David's phone number is 555-0123.  
Tom's phone number is 555-5678.
```

Review

A tuple is an immutable list, which means that once it has been defined the values in the tuple cannot be changed.

The **del** statement can be used to delete a tuple. **Del tuple_name**

It is possible to convert a tuple to a list using the **list()** built-in function.

Lists can also be converted to tuples by using the **tuple()** built-in function.

You can use tuple assignment to assign values to multiple variables at the same time. **(var_1, var_N) = (value_1, value_N)**

Tuple assignment can be used in **for** loops.

The `max()` built-in function will return the largest item that is passed to it.

The `min()` built-in function will return the smallest item that is passed to it.

Exercises

ZIP Codes

Try creating a list of cities that will include a series of tuples that contain both a city's name and its ZIP code. Loop through the list and utilize tuple assignment. Assign one variable to denote the city name and another variable to represent the ZIP code. Display the city's name and ZIP code to the screen.

Sample output:

Code Listing 227

```
The ZIP code for Short Hills, NJ is 07078.  
The ZIP code for Fairfax Station, VA is 22039.  
The ZIP code for Weston, CT is 06883.  
The ZIP code for Great Falls, VA is 22066.
```

Solution

Code Listing 228

```
cities = [  
    ('Short Hills, NJ', '07078'),  
    ('Fairfax Station, VA', '22039'),  
    ('Weston, CT', '06883'),  
    ('Great Falls, VA', '22066')  
]  
  
for (city, zip_code) in cities:  
    print('The ZIP code for {} is {}'.format(city, zip_code))
```

Resources

`list()` documentation: <https://docs.python.org/3/library/functions.html#func-list>

`max()` documentation: <https://docs.python.org/3/library/functions.html#max>

`min()` documentation: <https://docs.python.org/3/library/functions.html#min>

`type()` documentation: <https://docs.python.org/3/library/functions.html#type>

`tuple()` documentation: <https://docs.python.org/3/library/functions.html#func-tuple>

Chapter 8 File I/O

In previous chapters you've learned how to use the built-in `input()` function to accept standard input from the keyboard. You've also learned how to send data to standard output—the screen—using the `print()` function. While understanding and utilizing standard input and output will work well for certain types of applications, you will often need a place to store the data generated by your program. Also, you will need a way to retrieve saved data as well. One of the most common places to store data is within a file. You can read input and write output to a file, just like you can read input from a keyboard and display output on a screen.

To open a file, use the built-in `open()` function. The pattern for this is `open(path_to_file)`. The `path_to_file` can be either an absolute or a relative path, and it includes the file name. An absolute path will contain the entire path beginning at the root of the file system, be that a `/` in Mac or Linux, or a drive letter in Windows. Examples of absolute paths are `/var/log/messages` and `C:\Log\Messages\data.txt`. A relative path however, will comprise just the file name or a portion of the path which starts at the current working directory. An example of a relative path is `log/messages`. This example supposes the current working directory is `/var`.

Making use of forward slashes as a directory separator will be familiar to most of us, even those that have never worked on a Unix or Unix-like operating system. Python however recognizes forward slashes even when running on the Windows operating system. The Windows operating system uses backslashes as the directory separator. For instance, `C:/Users/david/Documents/python-notes.txt` is a valid absolute path within Python. Also, `Documents/python-notes.txt` is a valid relative path.

The `open()` function will return a file object, which is sometimes referred to as a stream object. This can be used to perform operations on the file passed to the `open()` function. To read the entire file in at once, use the `read()` method on the file object. The `read()` method returns a string comprising the file's contents. The following code listing is an example.

Code Listing 229

```
hosts = open('/etc/hosts')
hosts_file_contents = hosts.read()
print(hosts_file_contents)
```

Output:

Code Listing 230

```
127.0.0.1 localhost
```

In order to modify the previous example to work on a Windows system, set the **hosts** variable to **C:/Windows/System32/drivers/etc/hosts**.

Code Listing 231

```
hosts = open('C:/Windows/System32/drivers/etc/hosts')
```

File Position

Whenever a file is read, Python will keep track of your current position within that file. In cases where the **read()** method returns the entire file, the current position will always be at the end of the file. If you were to call **read()** again, an empty string would be returned since there is no more data to return at your current position in the file. To change the current file position, use the **seek()** method and pass in a byte offset. For instance, to go back to the beginning of the file, use **seek(0)**. If, however, you are looking to start at the fifth byte of the file, use **seek(5)**. Take note that in many cases the Nth byte will correspond to the Nth character in the file. However, in some cases it will not, so be aware of that. For UTF-8 encoded files you will often come across characters that are longer than one byte. You will encounter this situation when using Kanji, Korean, or Chinese. In order to determine your current position in the file, use the **tell()** method.

Code Listing 232

```
hosts = open('/etc/hosts')

print('Current position: {}'.format(hosts.tell()))
print(hosts.read())

print('Current position: {}'.format(hosts.tell()))
print(hosts.read())

hosts.seek(0)
print('Current position: {}'.format(hosts.tell()))
print(hosts.read())
```

Output:

```
Current position: 0
127.0.0.1 localhost

Current position: 20

Current position: 0
127.0.0.1 localhost
```

The **read()** method will accept the number of bytes/characters to read. The following example demonstrates reading the first three characters of the **hosts** file. In this case, the first three characters correspond with the first three bytes.

Code Listing 234

```
hosts = open('/etc/hosts')
print(hosts.read(3))
print(hosts.tell())
```

Output:

Code Listing 235

```
127
3
```

Closing a File

It is always a best practice to completely close a file once you are finished with it. Keep in mind that if your Python application opens too many files during its execution you could be faced with a “too many open files” error. To close a file, simply use the **close()** method on the file object.

Code Listing 236

```
hosts = open('/etc/hosts')
hosts_file_contents = hosts.read()
print(hosts_file_contents)
hosts.close()
```

Output:

Code Listing 237

```
127.0.0.1 localhost
```

Note that each file object has a **closed** attribute that returns **True** if the file is closed and **False** if it is not. You can make use of this attribute to ensure that a file is indeed closed.

Code Listing 238

```
hosts = open('/etc/hosts')
hosts_file_contents = hosts.read()
print('File closed? {}'.format(hosts.closed))
if not hosts.closed:
    hosts.close()
print('File closed? {}'.format(hosts.closed))
```

Output:

Code Listing 239

```
File closed? False
File closed? True
```

Automatically Closing a File

To automatically close a file use the **with** statement. The pattern is **with open(file_path) as file_object_variable_name** followed directly by a code block. Whenever the code block finishes, Python will automatically close the file. Also, in cases where the code block is interrupted for any reason, including an exception, the file will be closed.

Code Listing 240

```
print('Started reading the file.')
with open('/etc/hosts') as hosts:
    print('File closed? {}'.format(hosts.closed))
    print(hosts.read())
print('Finished reading the file.')
print('File closed? {}'.format(hosts.closed))
```

Output:

Code Listing 241

```
Started reading the file.
File closed? False
127.0.0.1 localhost
Finished reading the file.
File closed? True
```

Reading a File One Line at a Time

To read a file one line at a time, use a **for** loop. The pattern is **for line_variable in file_object_variable:** directly followed by a code block.

Code Listing 242

```
with open('file.txt') as the_file:
```

```
for line in the_file:  
    print(line)
```

Output:

Code Listing 243

```
This is the first line of the file.  
  
Here is the second line.  
  
Finally! This is the third and last line!
```

The following code listing is the content of **file.txt**.

Code Listing 244

```
This is the first line of the file.  
  
Here is the second line.  
  
Finally! This is the third and last line!
```

The output will contain a blank line between each one of the lines in the file. This is because the **line** variable encompasses the complete line from the file which includes a carriage return, or new line, character. To remove any trailing white space, including the new line and carriage return characters, use the **rstrip()** string method.

Code Listing 245

```
with open('file.txt') as the_file:  
    for line in the_file:  
        print(line.rstrip())
```

Output:

```

This is the first line of the file.

Here is the second line.

Finally! This is the third and last line!

```

File Modes

Whenever you open a file you have the option of specifying a mode. The pattern is **`open(path_to_file, mode)`**. So far in this book we have been relying on the default file mode of **`r`** which opens a file in read-only mode. If you want to write to a file, clearing any of its current contents, use the **`w`** mode. If you want to create a new file and write to it, use the **`x`** mode. In cases where the file already exists an exception will be raised. Using the **`x`** mode will prevent you from accidentally overwriting existing files. If you are looking to keep the contents of an existing file and append or add additional data to it, use the **`a`** mode. With both the **`w`** and **`a`** modes, if the file does not already exist, it will be created. If you want to read and write to the same file, use the **`+`** mode.

Table 4: File Modes

Mode	Description
r	Open for reading (default).
w	Open for writing, truncating the file first.
x	Create a new file and open it for writing.
A	Open for writing, appending to the end of the file if it exists.
b	Binary mode.
t	Text mode (default).

Mode	Description
+	Open a disk file for updating (reading and writing).

Keep in mind that you can also specify if the file you are working with is a text file or a binary file. By default, all files are opened as text files unless you directly specify otherwise. Simply append a **t** or **b** to one of the read or write modes. For example, to open a file for reading in binary mode, use **rb**. To append to a binary file, use **ab**.

Also keep in mind that while text files contain strings, binary files contain a series of bytes. Put simply, text files are readable by humans, while binary files are not. Examples of binary files include images, videos, and compressed files.

To look into the current mode of a file, examine the **mode** attribute on a file object.

Code Listing 247

```
with open('file.txt') as the_file:
    print(the_file.mode)
```

Output:

Code Listing 248

```
r
```

Writing to a File

Now that you are familiar with the different file modes, let's try writing some data to a file. This is as simple as calling the **write()** method on the file object and then supplying the text you wish to write to that file.

Code Listing 249

```
with open('file2.txt', 'w') as the_file:
    the_file.write('This text will be written to the file.')
    the_file.write('Here is some more text.')
```

```
with open('file2.txt') as the_file:
    print(the_file.read())
```

Output:

Code Listing 250

```
This text will be written to the file.Here is some more text.
```

Keep in mind however that the output you receive may not be exactly what you expected. The `write()` method will write exactly what was supplied to the file. In the previous example no carriage return or line feed was provided. As a result, all the text ended up on the same line. The `\r` sequence represents the carriage return character and `\n` represents a new line. Let's work through the example again, but this time let's ensure that we are using a new line character at the end of the line.

Code Listing 251

```
with open('file2.txt', 'w') as the_file:
    the_file.write('This text will be written to the file.\n')
    the_file.write('Here is some more text.\n')

with open('file2.txt') as the_file:
    print(the_file.read())
```

Output:

Code Listing 252

```
This text will be written to the file.
Here is some more text.
```

Keep in mind that Unix-style line endings will only contain the `\n` character. Mac and Linux files use this type of line ending. Windows-style line endings can be formed by using `\r\n`.

Binary Files

The key thing to remember when you are dealing with binary files is that you are working with bytes, not characters. The `read()` method will always accept bytes as an argument when dealing with binary files. Remember that the `read()` method will accept characters whenever the file is opened as a text file.

Code Listing 253

```
with open('pig.jpg', 'rb') as pig_picture:
    pig_picture.seek(2)
    pig_picture.read(4)
    print(pig_picture.tell())
    print(pig_picture.mode)
```

Output:

Code Listing 254

```
6
rb
```

Exceptions

Whenever you are working with anything that exists outside of your program you greatly increase the chance of errors and exceptions. Working with files falls squarely into this category. An example of this may occur when a file you are attempting to write to may be read-only. Or, a file you are attempting to read from may not be available. In a previous chapter we briefly examined the **try/except** block. In the following example we'll see how it can be put to use.

Code Listing 255

```
# Open a file and assign its contents to a variable.
# If the file is unavailable, create an empty variable.
try:
```

```
contacts = open('contacts.txt').read()
except:
    contacts = []

print(len(contacts))
```

Output:

Code Listing 256

```
3
```

If the file was unable to be read, the output would be:

Code Listing 257

```
0
```

Review

Use the built-in **open()** function to open a file. The pattern is **open(path_to_file, mode)**.

If **mode** is not supplied when opening a file it will default to read-only.

Forward slashes can be used as directory separators, even when you are using Windows.

Using the **read()** file object method will return the entire contents of the file as a string.

Use the **close()** file object method to close a file.

Use the **with** statement to automatically close a file. The pattern is **with open(file_path) as file_object_variable_name:** directly followed by a code block.

Use a **for** loop to read a file one line at a time. The pattern is **for line_variable in file_object_variable:**.

Use the **rstrip()** string method to remove any trailing white space.

Write data to a file using the **write()** file object method.

The **read()** file object method accepts the number of bytes to read when a file is opened in binary mode. When a file is opened in text mode, which is the default, **read()** will accept characters.

In the majority of cases a character will be one byte in length, but keep in mind that this does not hold true in every situation.

Always plan for exceptions when you are working with files. Use **try/except** blocks.

Exercises

Line Numbers

Try creating a program that opens **file.txt**. Read each line of the file and then prepend it with a line number.

Sample output:

Code Listing 258

```
1: This is the first line of the file.  
2: Here is the second line.  
3: Finally! This is the third and last line!
```

Solution

Code Listing 259

```
with open('file.txt') as file:  
    line_number = 1  
    for line in file:  
        print('{}: {}'.format(line_number, line.rstrip()))  
        line_number += 1
```

Alphabetize

Try reading the contents of **animals.txt** and from there create a file named **animals-sorted.txt** that is sorted alphabetically.

The following code listing is the body of the **animals.txt** file.

Code Listing 260

```
toad  
lion  
seal  
fox  
owl  
whale  
elk
```

Once the program has been executed, the contents of **animals-sorted.txt** should look like the following.

Code Listing 261

```
elk  
fox  
lion  
owl  
seal  
toad  
whale
```

Solution

Code Listing 262

```
unsorted_file_name = 'animals.txt'  
sorted_file_name = 'animals-sorted.txt'  
animals = []
```

```
try:
    with open(unsorted_file_name) as animals_file:
        for line in animals_file:
            animals.append(line)
        animals.sort()
except:
    print('Could not open {}'.format(unsorted_file_name))

try:
    with open(sorted_file_name, 'w') as animals_sorted_file:
        for animal in animals:
            animals_sorted_file.write(animal)
except:
    print('Could not open {}'.format(sorted_file_name))
```

Resources

Core tools for working with streams: <https://docs.python.org/3/library/io.html>

Handling Exceptions: <https://wiki.python.org/moin/HandlingExceptions>

`open()` documentation: <https://docs.python.org/3/library/functions.html#open>

Chapter 9 Modules

Modules

A Python module is a file that has a **.py** extension. These can be used to implement a set of attributes (variables), methods (functions), and classes (types). You can include a module in another Python program simply by using the **import** statement followed by the module name. To import a module named **time**, include **import time** within your Python program. You can now access the methods within the **time** module by calling **time.method_name()** or attributes, sometimes called variables, by calling **time.attribute_name**. The following code listing is an example using the **asctime()** method and the **timezone** attribute from the **time** module. The **timezone** attribute includes the number of seconds between UTC and the local time.

Code Listing 263

```
import time

print(time.asctime())

print(time.timezone)
```

Output:

Code Listing 264

```
Tue Aug 25 14:28:03 2015

21600
```

Whenever you **import module_name**, all of the methods in that module will be available as **module_name.method_name()**. If you opt to use a single method in a module you can import just that method using the **from module_name import method_name** syntax. Now the method can be accessed in your program by name. Instead of calling **module_name.method_name()**, you can now simply call **method_name()**.

Code Listing 265

```
from time import asctime

print(asctime())
```

Output:

```
Tue Aug 25 14:28:03 2015
```

It is possible to do the same thing using module attributes and classes. If you are looking to import more than one item from a module you can create a separate **from module_name import method_name** line for each one. You can also opt to provide a comma separated list like this: **from module_name import method_name1, method_name2, method_nameN**. Let's import the **asctime()** and **sleep()** methods from the **time** module. The **sleep()** method pauses the execution of your program for an allotted number of seconds.

Code Listing 267

```
from time import asctime, sleep
print(asctime())
sleep(5)
print(asctime())
```

Output:

Code Listing 268

```
Tue Aug 25 14:28:03 2015
Tue Aug 25 14:28:08 2015
```

One of the primary benefits of importing either a single method or list of methods from a module is that you can access it directly by name without having to precede it with the module name. For example, **sleep(5)** versus **time.sleep(5)**. If you want to be able to access everything from a module, you could use an asterisk instead of a list of methods to import. However, this is not a practice I recommend. It is worth mentioning here only because you will see it used from time to time. The main reason it is best to avoid this approach is that you run the risk of overriding an existing function or variable if you import everything into your program. Also, when you import multiple methods using an asterisk, you will find it hard to determine what exactly came from where.

Code Listing 269

```
from time import *
print(timezone)
```

```
print(asctime())  
  
sleep(5)  
  
print(asctime())
```

Output:

Code Listing 270

```
21600  
  
Tue Aug 25 14:28:03 2015  
  
Tue Aug 25 14:28:08 2015
```

Peeking Inside a Module

You can use the built-in `dir()` function to discover which attributes, methods, and classes exist within any one module.

Code Listing 271

```
>>> import time  
  
>>> dir(time)  
  
['_STRUCT_TM_ITEMS', '__doc__', '__file__', '__loader__', '__name__',  
 '__package__', '__spec__', 'altzone', 'asctime', 'clock', 'ctime', 'daylight',  
 'get_clock_info', 'gmtime', 'localtime', 'mktime', 'monotonic', 'perf_counter',  
 'process_time', 'sleep', 'strftime', 'strptime', 'struct_time', 'time',  
 'timezone', 'tzname', 'tzset']
```

The Module Search Path

Keep in mind that you can always view the default module search path by examining `sys.path`. Whenever you supply an `import module_name` statement, Python will look for the module in the first path on the list. If Python cannot find it then the next path will be examined and so on. This will continue either until the module is found or until all of the module search paths are completely exhausted. The module search path may include zip files as well as directories. Python will search within the zip file for a matching module as well. It is important to note that the default module search path will vary depending on your installation of Python, the Python version, and the operating system. The following code listing is an example from a Python installation on a Mac.

```
# show_module_path.py
import sys
for path in sys.path:
    print(path)
```

Output:

```
/Users/david
/Library/Frameworks/Python.framework/Versions/3.4/lib/python34.zip
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/plat-darwin
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages
```

The **show_module_path.py** file was located in **/Users/david** when I first executed **python3 show_module_path.py**. Notice that **/Users/david** is listed first in the module search path. The other directories were decided by the Python installation.

If you want to ask Python to search other locations for modules you will need to manipulate the module search path. There are two ways that you can do this. The first method is to modify **sys.path** just as you would any other list. For example, you can append directory locations by using a string data type.

```
import sys
sys.path.append('/Users/david/python')
for path in sys.path:
    print(path)
```

Output:

```

/Users/david

/Library/Frameworks/Python.framework/Versions/3.4/lib/python34.zip
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/plat-darwin
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages
/Users/david/python

```

You are also able to manipulate the **PYTHONPATH** environment variable. This variable acts in a very similar manner to the **PATH** environment variable. On Mac and Linux systems, **PYTHONPATH** can be populated with a list of directories separated by colons. On Windows systems the **PYTHONPATH** environment variable requires the use of a semicolon in order to separate the list of directories. The directories listed in **PYTHONPATH** are inserted after the directory where the script resides and before the default module search path.

In the following example, **/Users/david** is the directory where the **show_module_path.py** Python program resides. The **/Users/david/python** and **/usr/local/python/modules** paths are included in **PYTHONPATH**. The **export** command makes **PYTHONPATH** available to programs started from the shell.

```

[david@mac ~]$ export PYTHONPATH=/Users/david/python:/usr/local/python/modules
[david@mac ~]$ pwd
/Users/david
[david@mac ~]$ python3 show_module_path.py
/Users/david
/Users/david/python
/usr/local/python/modules
/Library/Frameworks/Python.framework/Versions/3.4/lib/python34.zip
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/plat-darwin

```

```
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages
[david@mac ~]$
```

If you are unable to find a module within the search path, an **ImportError** exception will be raised.

Code Listing 277

```
import say_hello
```

Output:

Code Listing 278

```
Traceback (most recent call last):
  File "test_say_hello.py", line 1, in <module>
    import say_hello
ImportError: No module named 'say_hello'
```

The Python Standard Library

As we've worked through previous examples, we have been using the `time` module which comes included with Python. In fact, Python is supplied with a large library of modules for you to take advantage of. I would highly recommend that you take some time to really look at what the Python standard library has to offer before you even think of writing any of your own code. For example, if you are looking to read and write CSV (comma-separated values) files, don't waste your time creating something from scratch when it already exists. Just use Python's pre-existing **csv** module. Are you looking to enable logging in your program? Well, there's a **logging** module which can help you do that. Do you want to make an HTTP request to a web service and then parse the JSON response? Try using the **urllib.request** and **json** modules. To explore these modules and more, check out the list of what is available in the Python Standard Library located at <https://docs.python.org/3/library/>.

Now, let's use the **exit()** method from the **sys** module to cleanly terminate your program if it detects an error. In the following example, the file **test.txt** is opened. If the program encounters an error while the file is opening, the code block following **except:** will execute. If the reading of **test.txt** is mandatory for the remaining code to function correctly, there is no need to continue. The **exit()** method can accept an exit code as an argument. If no exit code is provided, **0** will be used. By convention, when an error causes a program to exit, a non-zero exit code is expected.

```
import sys
file_name = 'test.txt'
try:
    with open(file_name) as test_file:
        for line in test_file:
            print(line)
except:
    print('Could not open {}'.format(file_name))
    sys.exit(1)
```

Creating Your Own Modules

Just as Python has a library of its own reusable code, so can you. It's quite simple to create your own module. Just remember that in its least complex form, modules are files that have a **.py** extension. Simply create a Python file with your code and **import** it from another Python program.

The following code listing is the content of **say_hello.py**.

Code Listing 280

```
def say_hello():
    print('Hello!')
```

Note how you can import and use the **say_hello** module. To call the **say_hello()** method within the **say_hello** module, use **say_hello.say_hello()**.

Code Listing 281

```
import say_hello
say_hello.say_hello()
```

Output:

```
Hello!
```

The following example is another simple module called **say_hello2**. The following code is the body of **say_hello2.py**.

```
def say_hello():  
    print('Hello!')  
  
print('Hello from say_hello2.py!')
```

Let's find out what happens when you import the **say_hello2** module.

```
import say_hello2  
say_hello2.say_hello()
```

Output:

```
Hello from say_hello2.py!  
Hello!
```

So what happened? Well, when **say_hello2** is imported its contents are executed. First, the **say_hello()** function is defined. From there the **print** function is executed. In this way Python enables you to create programs that behave one way when they are executed, and another way when they are imported. If you would like to be able to reuse functions from an existing Python program, but have no desire to execute the main program, you can account for that.

Using main

Whenever a Python file is executed as a program, the special variable `__name__` will be set to `__main__`. Notice that there are two underscore characters on each side of the names of these special variables. In instances where it is imported, the `__name__` variable will not be populated. Ultimately you can use this to control the behavior of your Python program. The following code sample is the `say_hello3.py` file.

Code Listing 286

```
def say_hello():
    print('Hello!')

def main():
    print('Hello from say_hello3.py!')
    say_hello()

if __name__ == '__main__':
    main()
```

Whenever it is executed as a program, the code block following `if __name__ == '__main__'` will be executed. In the following example it simply calls `main()`. This is a very common pattern and you will see this within many Python applications. When `say_hello3.py` is imported as a module nothing will be executed unless explicitly called from the importing program.

Code Listing 287

```
[david@mac ~]$ python3 say_hello3.py
Hello from say_hello3.py!
Hello!
[david@mac ~]$
```

Review

Python modules are files that have a .py extension and are capable of implementing a set of variables, functions, and classes.

Use the **import module_name** syntax to import a module.

The default module search path will be determined by your Python installation.

To manipulate the module search path, modify **sys.path** or set the **PYTHONPATH** environment variable.

The Python standard library is a sizeable collection of code that can be reused within your Python programs.

Use the **dir()** built-in function to find out exactly what exists within a module.

You can establish your own personal library by writing your own modules.

You can influence how a Python program behaves based on whether it is run interactively or imported by checking the value of **__name__**.

The **if __name__ == '__main__':** syntax is a common Python idiom.

Exercises

Pig Speak, Redux

Update the "[Pig Speak](#)" program we discussed in Chapter 1 so that it can be imported as a module or run directly. When run as a program it should prompt for input, as well as display a pig "saying" what was provided by the user. Place the input provided by the user inside a speech bubble. This speech bubble can be expanded or contracted to fit around the input provided by the user.

The following code listing shows the sample output when run interactively.

Code Listing 288

```

    _____
  < Pet me and I will oink >
    -----
      /
^..^ /
```

```
~( ( oo )
    , , , ,
```

From here, create a new program called **pig_talk.py** that imports the **pig_say** module. Try using a function from the **pig_say()** module to display a variety of messages to the screen.

The following code listing shows the sample output when used as a module.

Code Listing 289

```

    _____
    < Feed me. >
    -----
    /
    ^..^ /
~( ( oo )
    , , , ,

    _____
    < Oink. Oink. >
    -----
    /
    ^..^ /
~( ( oo )
    , , , ,
```

Solution

The following code is the content of **pig_say.py**.

Code Listing 290

```
def pig_say(text):
```

```

"""Generate a picture of a pig saying something"""
text_length = len(text)

print('          {}'.format('_' * text_length))
print('        < {} >'.format(text))
print('          {}'.format('-' * text_length))
print('        /')
print('     ^..^  /')
print('~(  ( oo )')
print('  ,,  ,,')

def main():
    text = input('What would you like the pig to say? ')
    pig_say(text)

if __name__ == '__main__':
    main()

```

The following code is the content of **pig_talk.py**.

Code Listing 291

```

import pig_say

def main():
    pig_say.pig_say('Feed me.')

```

```
pig_say.pig_say('Oink. Oink.')

if __name__ == '__main__':
    main()
```

Resources

`__main__` documentation: https://docs.python.org/3/library/__main__.html

Idioms and Anti-Idioms in Python: <https://docs.python.org/3.1/howto/doanddont.html>

Linux for Beginners: <http://www.linuxtrainingacademy.com/linux>.

`PYTHONPATH` documentation: <https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH>

The Python Standard Library: <https://docs.python.org/3/library/>

The `sys` module: <https://docs.python.org/3/library/sys.html>

`sys.path` documentation: <https://docs.python.org/3/library/sys.html#sys.path>

`virtualenv` documentation: <https://pypi.python.org/pypi/virtualenv>

Conclusion

Although we've reached the end of this book, I sincerely hope that this is just the beginning of your Python journey. Growing steadily in popularity over the last decade, Python is key to know as it is increasingly used in all areas of computing. In fact, Python powers many popular websites such as Pinterest, Instagram, and Reddit. Python is also used in scientific computing and is currently running on supercomputers all around the world. But Python's use might even be more far reaching than you know. Utilized in system administration tasks like configuration and package management (with YUM and Anaconda being prime examples), Python has also been used to create popular games such as *EVE Online* and *Toontown*. No matter what direction your programming interests lie in, Python's possibilities for learning, exploring, and growing are endless.

Here's one last Python program.

Code Listing 292

```
import this
```

Output:

Code Listing 293

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
```

```
Explicit is better than implicit.
```

```
Simple is better than complex.
```

```
Complex is better than complicated.
```

```
Flat is better than nested.
```

```
Sparse is better than dense.
```

```
Readability counts.
```

```
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
```

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than **right** now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Appendix A: Trademarks

BSD/OS is a trademark of Berkeley Software Design, Inc. in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Mac and OS X are trademarks of Apple Inc., registered in the U.S. and other countries.

Open Source is a registered certification mark of Open Source Initiative.

Python is a registered trademark of the Python Software Foundation.

UNIX is a registered trademark of The Open Group.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

All other product names mentioned herein are the trademarks of their respective owners.