

actionable

LEARNING WITHOUT THE BLAH BLAH BLAH

PYTHON 3.8

A WEBUCATOR™ GUIDE

```
False
ist.txt', 'r')
f.read().splitlines()
n in items:
ems.remove(item)
tem_found = True
print('"' + item + '" not found in list.')

n_found:
th open('list.txt', 'w') as f:
f.write('\n'.join(items) + '\n')

te_list():
After confirming user really wants to delete list,
etes the entire contents of the list by opening
st.txt for writing. """
input('Are you sure you want to delete the list?')
```

Nat Dunn

ac•tion•a•ble

adjective - Able to be put into practice; having a practical or usable value.



Python 3.8



with examples and hands-on exercises

WEBUCATOR

Copyright © 2020 by Webucator. All rights reserved.

No part of this book may be reproduced or used in any manner without written permission of the copyright owner.

Version: PYT-111.1.1.4

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at [@natdunn](https://twitter.com/natdunn) and Webucator at [@webucator](https://twitter.com/webucator).

Stephen Withrow (editor)

Stephen has over 30 years of experience in training, development, and consulting in a variety of technology areas including Python, Java, C, C++, XML, JavaScript, Tomcat, JBoss, Oracle, and DB2. His background includes design and implementation of business solutions on client/server, Web, and enterprise platforms. Stephen has a degree in Computer Science and Physics from Florida State University.

Class Files

Download the class files used in this book at

<https://www.webucator.com/class-files/index.cfm?versionId=4734>.

Errata

Corrections to errors in the book can be found at
<https://www.webucator.com/books/errata.cfm>.

Acknowledgments

A huge thanks to the phenomenal trainers who have taught Python using variations of this material for Webucator over the years: Stephen Withrow, Roger Sakowski, Mark Copley, and others.

And another huge thanks to Dan Salamida for designing the book cover.

And one final thanks to all the hard work the Python team does. You can join Webucator in supporting them at <https://www.python.org/psf/sponsorship/sponsors/>.

Actionable

adj.

- ~~1. Giving cause for legal action: an actionable statement.~~
- ~~2. Relating to or being information that allows a decision to be made or action to be taken.~~
- 3. Capable of being put into practice.**

Source: <https://ahdictionary.com/word/search.html?q=actionable>

In this book, we cover material that you will be able to immediately put into practice.

How to Read this Book

(DON'T SKIP THIS)

Struggle, failure, triumph: while triumph is the thing sought, struggle has its joy, and failure is not without its uses.

*– Brave Men and Women – Their Struggles, Failures, and Triumphs,
O.E. Fuller, A.M.*

Learning to program is hard. Don't let anyone tell you different. I recently taught a once-a-week Python class for homeschooled high school students. Many of them were in the class because they or their parents thought that Python was a good thing to learn. If you bought or are considering buying this book because someone told you that you *should* learn Python, then maybe you should reconsider. The big question to ask yourself is "Do you want to learn Python?" Certainly, this book will help you answer that question, but be prepared to struggle, and to make mistakes, and to get frustrated, and to bang your head against the keyboard. This is Pandora's box, baby, but...

If you get excited by solving hard problems... if you're into Sudoku, or crosswords, or jigsaw puzzles, or if you'll spend hours trying to figure out how to separate two metal looping thingamajigs that you found on someone's shelf..., or you like tinkering with engines, or figuring out the best recipe for chocolate chip cookies... in other words, if you like a challenge, then you might love Python programming.

It's not easy, but it's easy enough to get started. Pick up a book (good start!), and start going through it. Be patient with yourself. Pretend you're the teacher as well as the student. When the student fails, the good teacher doesn't insult or chastise. The good teacher encourages. The good teacher knows that failure is a part of learning, and that overcoming failure is exciting and leads to more learning. So, be a good teacher and treat yourself well.

But be a good student too. When reading this book, you should be sitting at your computer. When you first begin, plan to spend at least two uninterrupted hours. You should do your best to get through the first two lessons in the first sitting. That will get you through the foundational learning. From there on out, you'll be hopping back and forth between the book and viewing, editing, and writing code. Go through the book slowly and methodically. Read every demo carefully. Work through every exercise. You cannot learn to code through reading alone. You must practice.

Table of Contents

1. [Setting Up Your Computer](#)
 - A. [Demo and Exercise Files](#)
 - B. [Visual Studio Code](#)
2. [Python Basics](#)
 - A. [Getting Familiar with the Terminal](#)
 - B. [Running Python](#)
 - C. [Running a Python File](#)
 - D. [Exercise 1: Hello, world!](#)
 - E. [Literals](#)
 - F. [Python Comments](#)
 - G. [Data Types](#)
 - H. [Exercise 2: Exploring Types](#)
 - I. [Variables](#)
 - J. [Exercise 3: A Simple Python Script](#)
 - K. [Constants](#)
 - L. [Deleting Variables](#)
 - M. [Writing a Python Module](#)
 - N. [print\(\) Function](#)
 - O. [Collecting User Input](#)
 - P. [Exercise 4: Hello, You!](#)
 - Q. [Reading from and Writing to Files](#)
 - R. [Exercise 5: Working with Files](#)
3. [Functions and Modules](#)
 - A. [Defining Functions](#)
 - B. [Variable Scope](#)
 - C. [Global Variables](#)
 - D. [Function Parameters](#)

- E. [Exercise 6: A Function with Parameters](#)
- F. [Default Values](#)
- G. [Exercise 7: Parameters with Default Values](#)
- H. [Returning Values](#)
- I. [Importing Modules](#)
- J. [Methods vs. Functions](#)

4. [Math](#)

- A. [Arithmetic Operators](#)
- B. [Exercise 8: Floor and Modulus](#)
- C. [Assignment Operators](#)
- D. [Precedence of Operations](#)
- E. [Built-in Math Functions](#)
- F. [The math Module](#)
- G. [The random Module](#)
- H. [Exercise 9: How Many Pizzas Do We Need?](#)
- I. [Exercise 10: Dice Rolling](#)

5. [Python Strings](#)

- A. [Quotation Marks and Special Characters](#)
- B. [String Indexing](#)
- C. [Exercise 11: Indexing Strings](#)
- D. [Slicing Strings](#)
- E. [Exercise 12: Slicing Strings](#)
- F. [Concatenation and Repetition](#)
- G. [Exercise 13: Repetition](#)
- H. [Combining Concatenation and Repetition](#)
- I. [Python Strings are Immutable](#)
- J. [Common String Methods](#)
- K. [String Formatting](#)

- L. [Exercise 14: Playing with Formatting](#)
 - M. [Formatted String Literals \(f-strings\)](#)
 - N. [Built-in String Functions](#)
 - O. [Exercise 15: Outputting Tab-delimited Text](#)
6. [Iterables: Sequences, Dictionaries, and Sets](#)
- A. [Definitions](#)
 - B. [Sequences](#)
 - C. [Lists](#)
 - D. [Sequences and Random](#)
 - E. [Exercise 16: Remove and Return Random Element](#)
 - F. [Tuples](#)
 - G. [Ranges](#)
 - H. [Converting Sequences to Lists](#)
 - I. [Indexing](#)
 - J. [Exercise 17: Simple Rock, Paper, Scissors Game](#)
 - K. [Slicing](#)
 - L. [Exercise 18: Slicing Sequences](#)
 - M. [min\(\), max\(\), and sum\(\)](#)
 - N. [Converting Sequences to Strings with str.join\(seq\)](#)
 - O. [Splitting Strings into Lists](#)
 - P. [Unpacking Sequences](#)
 - Q. [Dictionaries](#)
 - R. [The len\(\) Function](#)
 - S. [Exercise 19: Creating a Dictionary from User Input](#)
 - T. [Sets](#)
 - U. [*args and **kwargs](#)
7. [Virtual Environments, Packages, and pip](#)
- A. [Exercise 20: Creating, Activating, Deactivating, and Deleting a](#)

[Virtual Environment](#)

- B. [Packages with pip](#)
- C. [Exercise 21: Working with a Virtual Environment](#)

8. [Flow Control](#)

- A. [Conditional Statements](#)
- B. [Compound Conditions](#)
- C. [The is and is not Operators](#)
- D. [all\(\) and any\(\)](#)
- E. [Ternary Operator](#)
- F. [In Between](#)
- G. [Loops in Python](#)
- H. [Exercise 22: All True and Any True](#)
- I. [break and continue](#)
- J. [Looping through Lines in a File](#)
- K. [Exercise 23: Word Guessing Game](#)
- L. [Exercise 24: for...else](#)
- M. [The enumerate\(\) Function](#)
- N. [Generators](#)
- O. [List Comprehensions](#)

9. [Exception Handling](#)

- A. [Exception Basics](#)
- B. [Wildcard except Clauses](#)
- C. [Getting Information on Exceptions](#)
- D. [Exercise 25: Raising Exceptions](#)
- E. [The else Clause](#)
- F. [The finally Clause](#)
- G. [Using Exceptions for Flow Control](#)
- H. [Exercise 26: Running Sum](#)

- I. [Raising Your Own Exceptions](#)
- 0. [Python Dates and Times](#)
 - A. [Understanding Time](#)
 - B. [The time Module.](#)
 - C. [Time Structures](#)
 - D. [Times as Strings](#)
 - E. [Time and Formatted Strings](#)
 - F. [Pausing Execution with time.sleep\(\)](#)
 - G. [The datetime Module](#)
 - H. [datetime.datetime Objects](#)
 - I. [Exercise 27: What Color Pants Should I Wear?](#)
 - J. [datetime.timedelta Objects](#)
 - K. [Exercise 28: Report on Departure Times](#)
- 1. [File Processing](#)
 - A. [Opening Files](#)
 - B. [Exercise 29: Finding Text in a File](#)
 - C. [Writing to Files](#)
 - D. [Exercise 30: Writing to Files](#)
 - E. [Exercise 31: List Creator](#)
 - F. [The os Module](#)
 - G. [Walking a Directory](#)
 - H. [The os.path Module](#)
 - I. [A Better Way to Open Files](#)
 - J. [Exercise 32: Comparing Lists](#)
- 2. [PEP8 and Pylint](#)
 - A. [PEP8](#)
 - B. [Pylint](#)

3. [Advanced Python Concepts](#)

- A. [Lambda Functions](#)
- B. [Advanced List Comprehensions](#)
- C. [Exercise 33: Rolling Five Dice](#)
- D. [Collections Module](#)
- E. [Exercise 34: Creating a defaultdict](#)
- F. [Counters](#)
- G. [Exercise 35: Creating a Counter](#)
- H. [Mapping and Filtering](#)
- I. [Mutable and Immutable Built-in Objects](#)
- J. [Sorting](#)
- K. [Exercise 36: Converting list.sort\(\) to sorted\(iterable\)](#)
- L. [Sorting Sequences of Sequences](#)
- M. [Creating a Dictionary from Two Sequences](#)
- N. [Unpacking Sequences in Function Calls](#)
- O. [Exercise 37: Converting a String to a datetime.date Object](#)
- P. [Modules and Packages](#)

4. [Regular Expressions](#)

- A. [Regular Expression Tester](#)
- B. [Regular Expression Syntax](#)
- C. [Python's Handling of Regular Expressions](#)
- D. [Exercise 38: Green Glass Door](#)

5. [Working with Data](#)

- A. [Virtual Environment](#)
- B. [Relational Databases](#)
- C. [Passing Parameters](#)
- D. [SQLite](#)
- E. [Exercise 39: Querying a SQLite Database](#)

- F. [SQLite Database in Memory](#)
 - G. [Exercise 40: Inserting File Data into a Database](#)
 - H. [Drivers for Other Databases](#)
 - I. [CSV](#)
 - J. [Exercise 41: Finding Data in a CSV File](#)
 - K. [Creating a New CSV File](#)
 - L. [Exercise 42: Creating a CSV with DictWriter](#)
 - M. [Getting Data from the Web](#)
 - N. [Exercise 43: HTML Scraping](#)
 - O. [XML](#)
 - P. [JSON](#)
 - Q. [Exercise 44: JSON Home Runs](#)
6. [Testing and Debugging](#)
- A. [Testing for Performance](#)
 - B. [Exercise 45: Comparing Times to Execute](#)
 - C. [The unittest Module](#)
 - D. [Exercise 46: Fixing Functions](#)
 - E. [Special unittest.TestCase Methods](#)
7. [Classes and Objects](#)
- A. [Attributes](#)
 - B. [Behaviors](#)
 - C. [Classes vs. Objects](#)
 - D. [Attributes and Methods](#)
 - E. [Exercise 47: Adding a roll\(\) Method to Die](#)
 - F. [Private Attributes](#)
 - G. [Properties](#)
 - H. [Exercise 48: Properties](#)
 - I. [Objects that Track their Own History](#)

- J. [Documenting Classes](#)
 - K. [Exercise 49: Documenting the Die Class](#)
 - L. [Inheritance](#)
 - M. [Exercise 50: Extending the Die Class](#)
 - N. [Extending a Class Method](#)
 - O. [Exercise 51: Extending the roll\(\) Method](#)
 - P. [Static Methods](#)
 - Q. [Class Attributes and Methods](#)
 - R. [Abstract Classes and Methods](#)
 - S. [Understanding Decorators](#)
8. [What Now?](#)
- A. [Where to Go from Here](#)

LESSON 1

Setting Up Your Computer

Topics Covered

- Getting the demo and exercise files.
- Installing software.

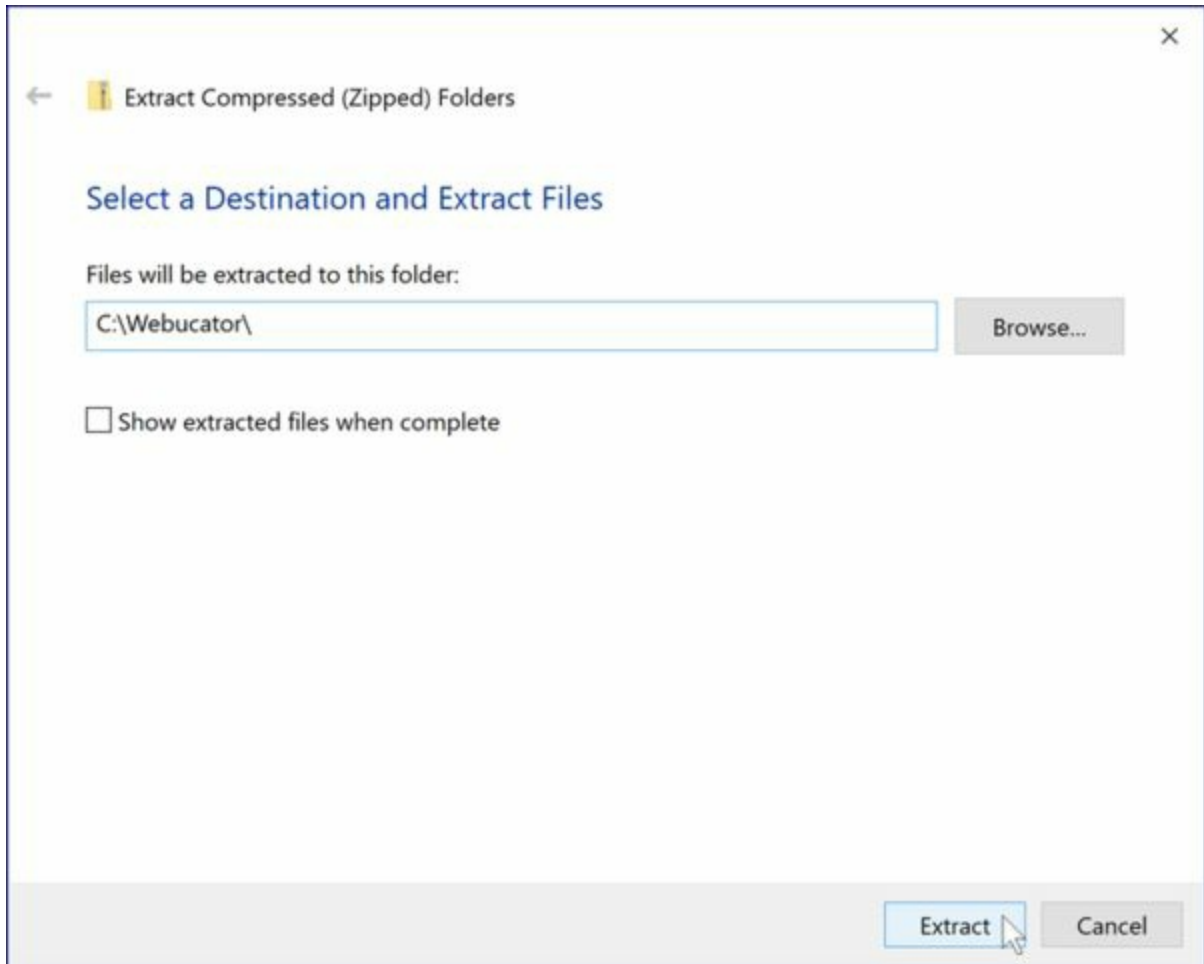
The mechanic, who wishes to do his work well, must first sharpen his tools.

– Confucius

Demo and Exercise Files

To get started, you need to get the demo and exercise files used throughout these lessons.

1. Create a new folder named Webucator on your computer wherever you want, but make sure you remember where it is.
2. Download the files used in this book.¹
3. Save the file into the Webucator folder you created in step 1 above.
4. Unzip the files into the Webucator folder:



5. This will create a ClassFiles folder in the Webucator folder. Rename the ClassFiles folder Python. The structure should look like this:



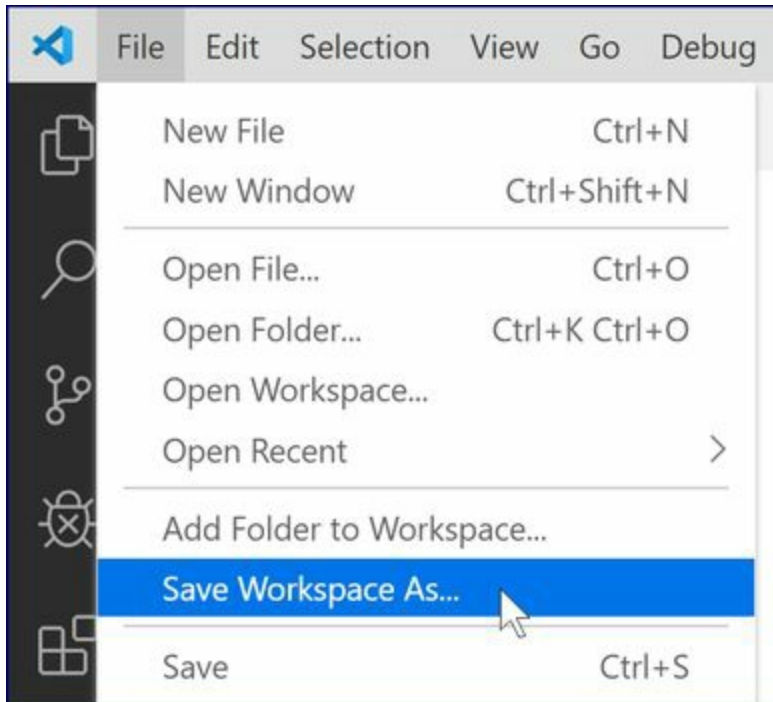
Visual Studio Code

We recommend that you use Visual Studio Code as your editor.

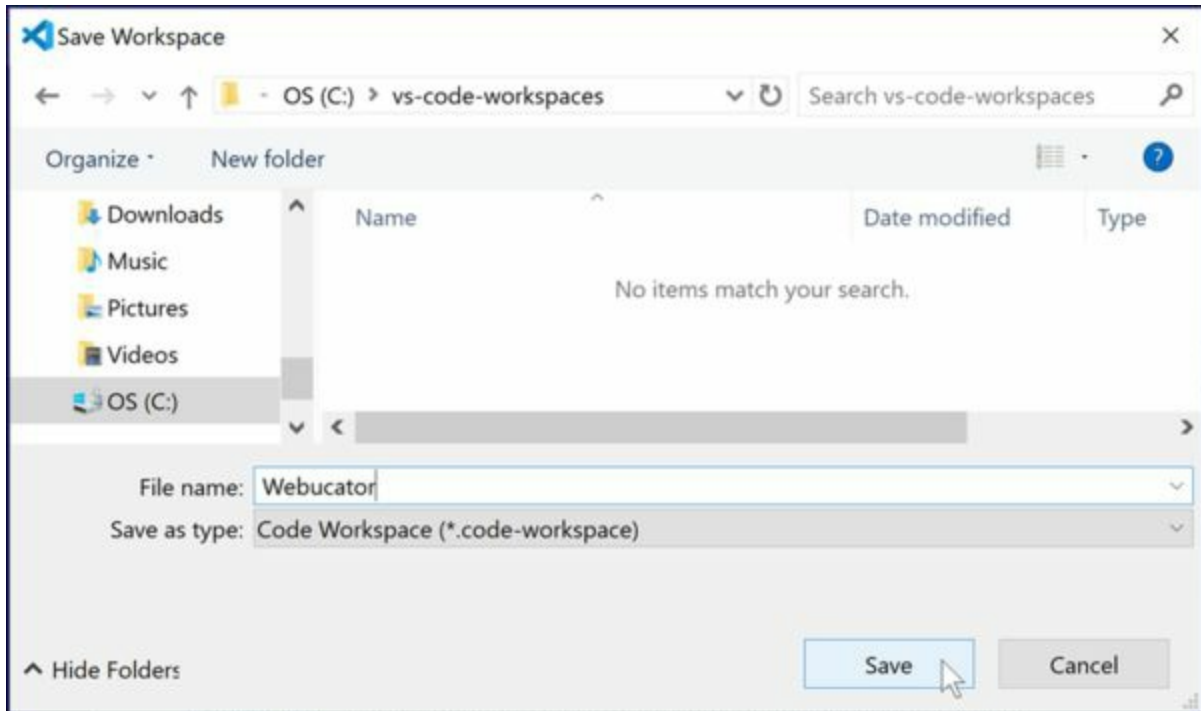
1. Download Visual Studio Code for your operating system at <https://code.visualstudio.com>.
2. Install Visual Studio Code.
 - A. Instructions for Windows:
<https://code.visualstudio.com/docs/setup/windows>
 - B. Instructions for Mac:
<https://code.visualstudio.com/docs/setup/mac>
3. Create a folder somewhere on your computer for storing Visual

Studio Code workspaces. Name the folder vs-code-workspaces or something similar.

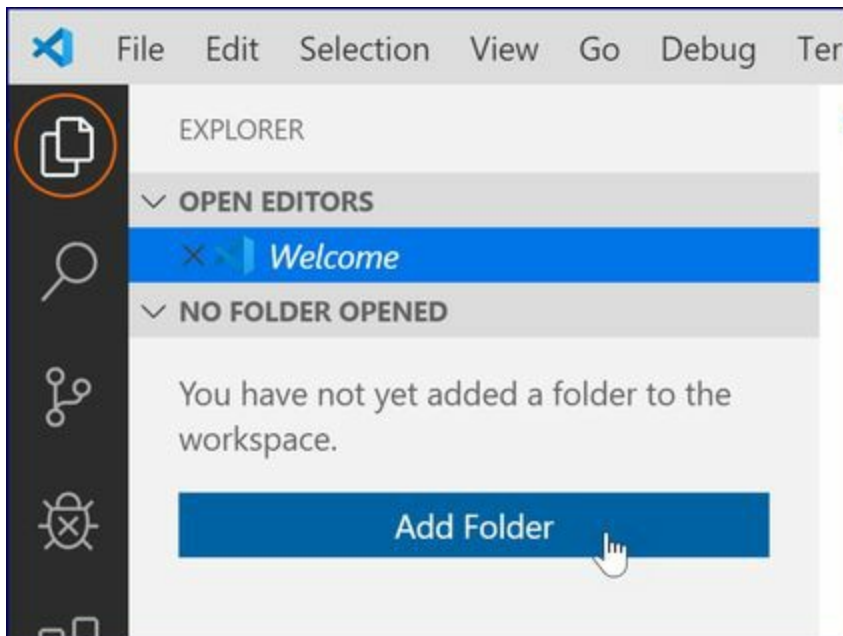
4. Open Visual Studio Code.
5. From the **File** menu, select **Save Workspace As...**



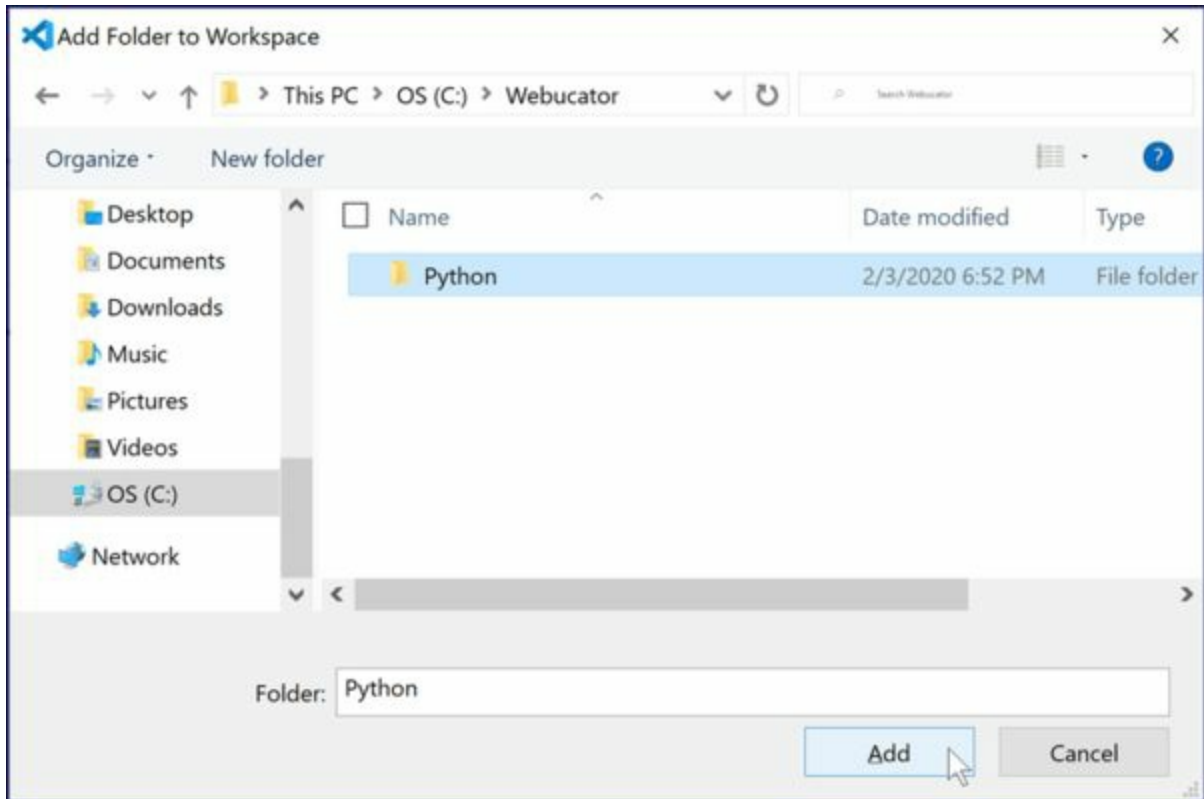
6. Save the workspace as Webucator within the workspaces folder you created earlier:



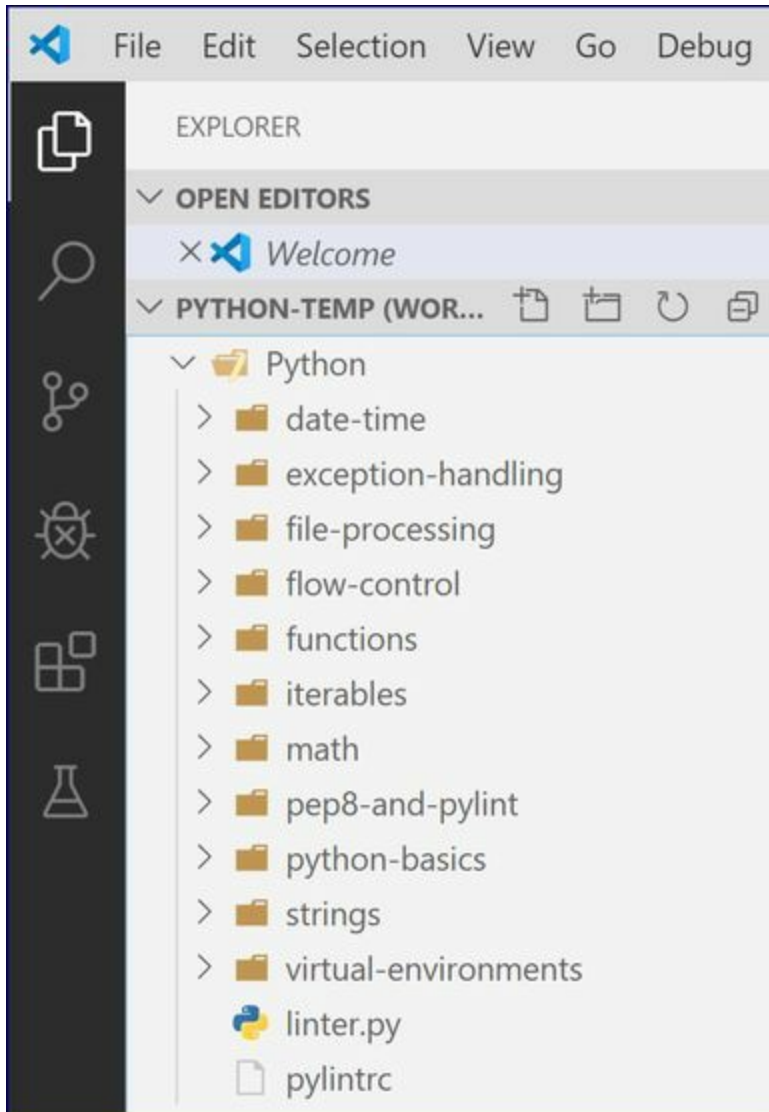
7. If Visual Studio Code's **Explorer** panel isn't open, open it by clicking the files icon in the upper left. Then click the **Add Folder** button:



8. Select the Python folder and click **Add**:



9. You will now see the Python folder in Visual Studio Code's **Explorer** panel:

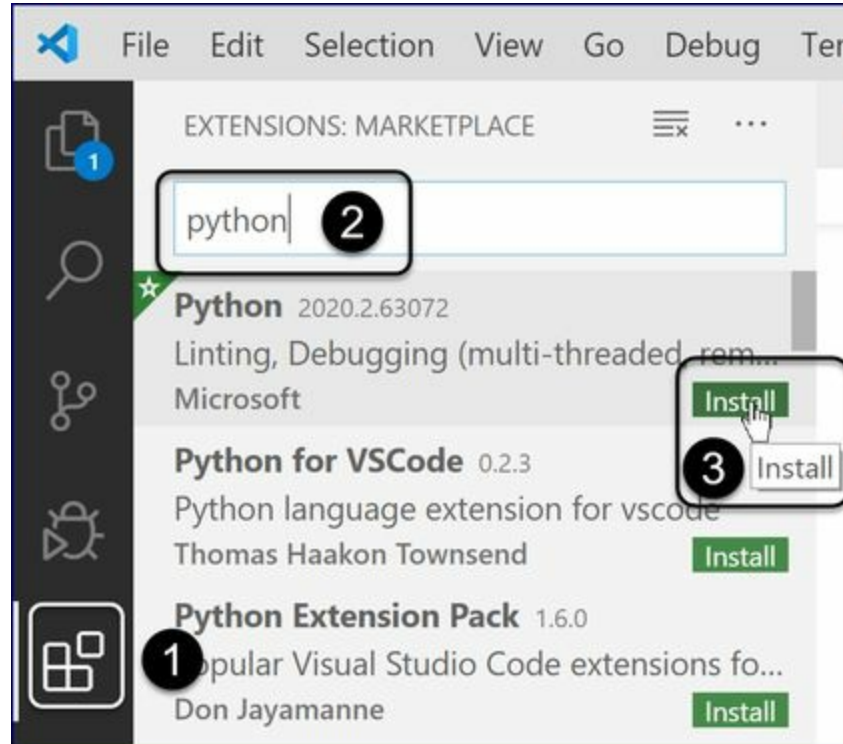


The Python Extension

Visual Studio Code has many freely available extensions for different programming languages. Python developers should install Microsoft's Python extension. As shown in the screenshot that follow:

1. Click the **Extensions** icon (below the bug) on the left of the **Explorer** panel.
2. Search for "Python".
3. If it doesn't show that the extension is already installed, click the

Install button.



Visual Studio Code Color Themes

You can customize the Visual Studio Code color theme by selecting **File > Preferences > Color Theme**. The default is a dark theme. We use a light theme for our screenshots.

Conclusion

Your computer should now be all set for viewing and editing the files in this book.

LESSON 2

Python Basics

Topics Covered

- How Python works.
- Python's place in the world of programming languages.
- Python literals.
- Python comments.
- Variables and Python data types.
- Simple modules.
- Outputting data with `print()`.
- Collecting user input.

*The pythons had entered into Mankind. No man knew at what moment
he might be Possessed!*

– Plague of Pythons, Frederik Pohl

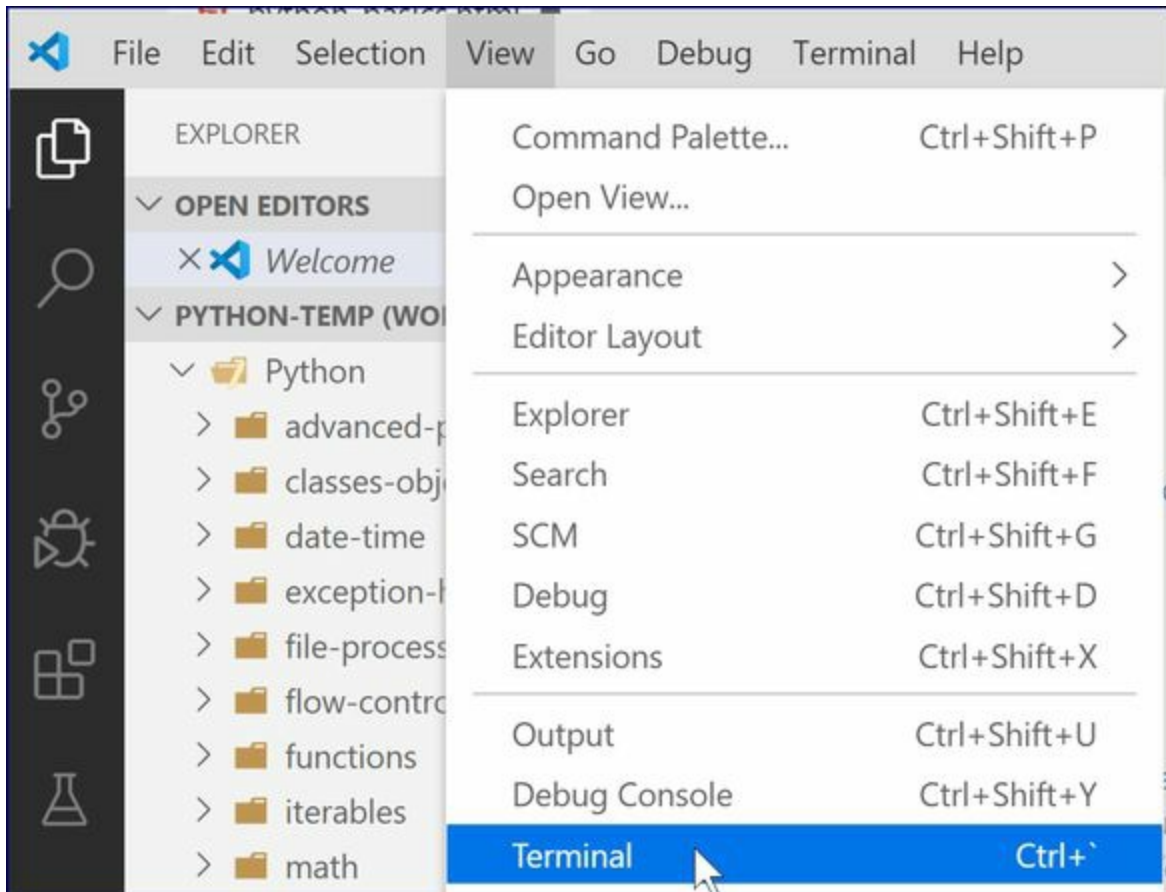
Python, which first appeared in 1991, is one of the most popular programming languages used today.² Python is a high-level programming language, meaning that it uses a syntax that is relatively human readable, which gets translated by a Python Interpreter into a language your computer can understand. Examples of other popular high-level programming languages are C#, Objective-C, Java, PHP, and JavaScript. Interestingly, all of these other languages, unlike Python, share a C-like syntax. If you use one or more of those languages, you may find Python's syntax a little strange. But give it a little time. You'll find it's quite programmer friendly.

Getting Familiar with the Terminal

Python developers need to be comfortable navigating around and

running commands at the terminal.³ We'll walk you through some basics:

1. Open a terminal. In Visual Studio Code, you can open a terminal by selecting **Terminal** from the **View** menu:



You can also open a terminal by pressing **Ctrl+`**. The **`** key is on the upper left of most keyboards. It looks like this:



The terminal will open at the root of your Visual Studio Code workspace. If you're working in the workspace we had you set up, that should be in a Webucator\Python directory (the words *folder* and *directory* are used interchangeably). The prompt on Windows will read something like:

```
PS C:\Webucator\Python>
```

On a Mac, it will show some combination of your computer name, the directory you are in, and your username, followed by a \$ or % sign. For example:

```
NatsMac:Python natdunn$
```

```
natdunn@NatsMac:Python %
```

2. Use `cd` to **change directories**. From the Webucator\Python directory, run:

```
PS C:\Webucator\Python> cd python-basics
```

```
PS ...\Python\python-basics>
```

Your prompt now shows that you are in the python-basics directory.

3. Move up to the parent directory by running:

```
cd ..
```

You will now be back in your Python directory.

4. Run:

```
cd python-basics/Demos
```

Your prompt will show that you are in the Demos directory. Depending on your environment, it may also show one or more directories above the Demos directory. To get the full path to your current location, run `pwd` for **p**resent **w**orking **d**irectory:

```
pwd
```

On Windows, that will look something like this:

```
PS ...\python-basics\Demos> pwd
```

```
Path
----
C:\Webucator\Python\python-basics\Demos
```

On a Mac, it will look something like this:

```
natdunn@NatsMac Demos % pwd
/Users/natdunn/Documents/Webucator/Python/python-basics/Demos
```

5. Run `cd ..` to back up to the python-basics directory.
6. Type `cd De` and then press the **Tab** key. On Windows, you should see something like this:

```
PS ...\\Python\\python-basics> cd ..\\Demos\\
```

The “.” at the beginning of the path represents the *current* (or *present*) directory. So, `..\\Demos` refers to the Demos directory within the current directory. A Mac won’t include the current directory in the path. When you press **Tab**, it will just fill out the rest of the folder name.

```
cd Demos
```

Press **Enter** to run the command. Then run `cd ..` to move back up to the python-basics directory.

7. Each of our lesson folders will contain Demos, Exercises, and Solutions folders. Some may contain additional folders. To see the contents of the current directory, run `dir` on Windows or `ls` on Mac/Linux:⁴

Windows PowerShell

```
PS ...\\Python\\python-basics> dir
```

```
Directory: C:\Webucator\Python\python-basics
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----

d-----	2/18/2020	6:13 AM	data
d-----	2/18/2020	6:13 AM	Demos
d-----	2/18/2020	5:09 AM	Exercises
d-----	2/18/2020	6:13 AM	Solutions

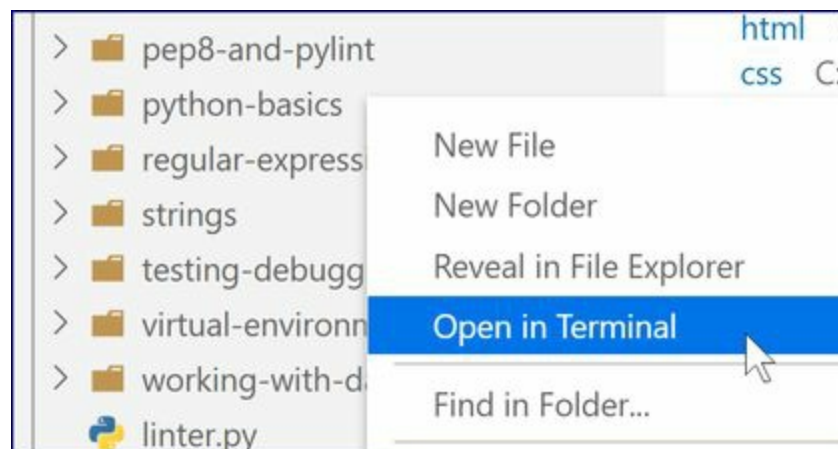
Mac Terminal

```
NatsMac:python-basics natdunn$ ls
Demos                Exercises            Solutions            data
```

8. Play with switching between directories using the `cd` command until you feel comfortable navigating the terminal.

Visual Studio Shortcut

Visual Studio provides a shortcut for opening a specific directory at the terminal. Simply right-click on the directory in the **Explorer** panel and select **Open in Terminal**:



Running Python

Python runs on Microsoft Windows, Mac OS X, Linux, and other Unix-like systems. The first thing to do is to make sure you have a recent version of Python installed:

1. Open the terminal in Visual Studio Code.
2. Run `python -V`:


```
PS C:\Webucator\Python> python -V
Python 3.8.1
```

If you have Python 3.6 or later, you are all set.

If you do not have Python 3.6 or later installed, download it for free at <https://www.python.org/downloads>. After running through the installer, run `python -v` at the terminal again to make sure Python installed correctly.

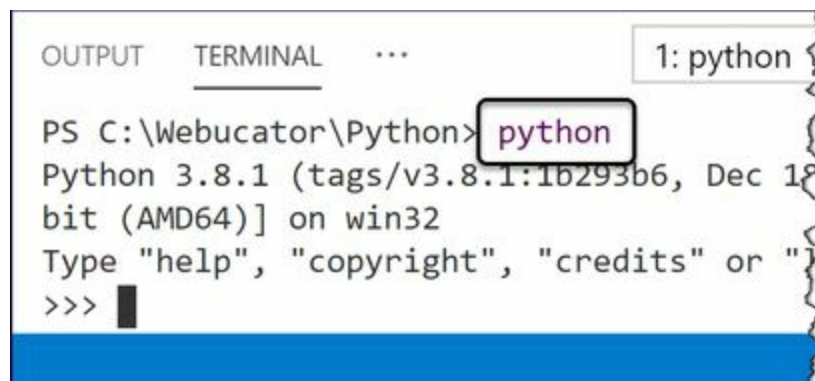
Python Versions on Macs

Your Mac will likely have a version of Python 2 already installed. After you install Python 3, you may find that running `python -v` still shows the Python 2 version. In that case, try running `python3 -v`. That should output the version of Python 3 that you have. If it does, then you should use the `python3` command instead of the `python` command to run Python 3.

If you would prefer to be able to use the `python` command for Python 3 (and who wouldn't), visit <https://www.webucator.com/blog/2020/02/mapping-python-to-python-3-on-your-mac/> to see how you can map `python` to `python3`.

Python Interactive Shell

You can run Python in *Interactive Mode* by running `python` at the terminal:



```
OUTPUT  TERMINAL  ...  1: python
PS C:\Webucator\Python> python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019) [AMD64] on win32
Type "help", "copyright", "credits" or "quit()"
>>>
```

This will open up the *Python shell*, at which you can run Python commands. For example, to print out “Hello, world!”, you would run:

```
print('Hello, world!')
```

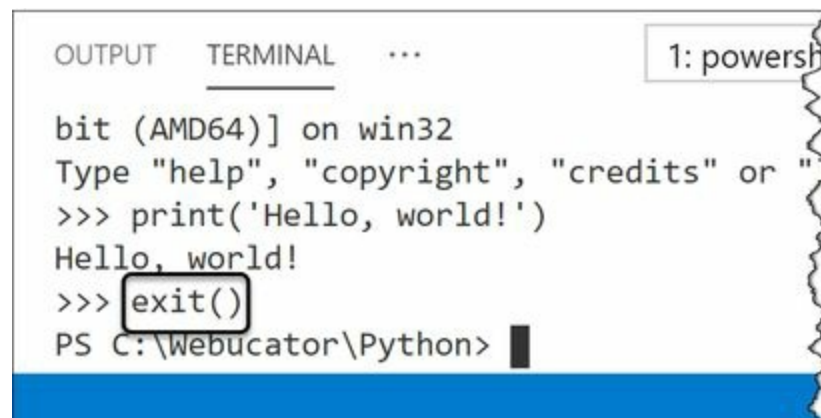
A screenshot of a Python terminal window. The window has tabs labeled 'OUTPUT', 'TERMINAL', and '...'. The 'TERMINAL' tab is active. The title bar on the right says '1: python'. The terminal text shows: 'Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 15:17:04) [AMD64] on win32', 'Type "help", "copyright", "credits" or "quit()" to quit the Python shell.', and the command '>>> print('Hello, world!')' which has been executed, resulting in 'Hello, world!'. The prompt '>>>' is followed by a cursor. The terminal window has a blue background at the bottom.

You can tell that you are at the Python prompt by the three right-angle brackets:

```
>>> █
```

To exit the Python shell, run:

```
exit()
```

A screenshot of a Python terminal window. The window has tabs labeled 'OUTPUT', 'TERMINAL', and '...'. The 'TERMINAL' tab is active. The title bar on the right says '1: powershell'. The terminal text shows: 'Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 15:17:04) [AMD64] on win32', 'Type "help", "copyright", "credits" or "quit()" to quit the Python shell.', and the command '>>> print('Hello, world!')' which has been executed, resulting in 'Hello, world!'. The prompt '>>>' is followed by the command '>>> exit()' which has been executed, resulting in 'PS C:\Webucator\Python>'. The terminal window has a blue background at the bottom.

Now, you are back at the regular prompt.

Running a Python File

While working at the Python shell can be useful in some scenarios,

you will often be writing Python files, so you can save, re-run, and share your code. Let's get started with a simple "Hello, world!" demo using your editor. We will open a *script*, which is simply a file with a .py extension that contains Python code. After the demonstration, you will add another line of code to the script in an exercise.

Here is the script we are going to run:

Demo 1: python-basics/Demos/hello_world.py

```
# Say Hello to the World  
print("Hello, world!")
```

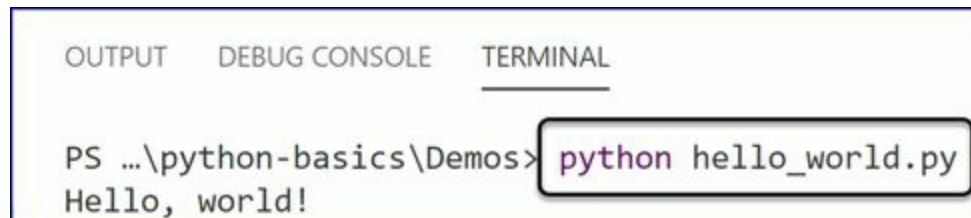
Code Explanation

The `print()` function simply outputs content either to standard output (e.g., the terminal) or to a file if specified.

To run this code:

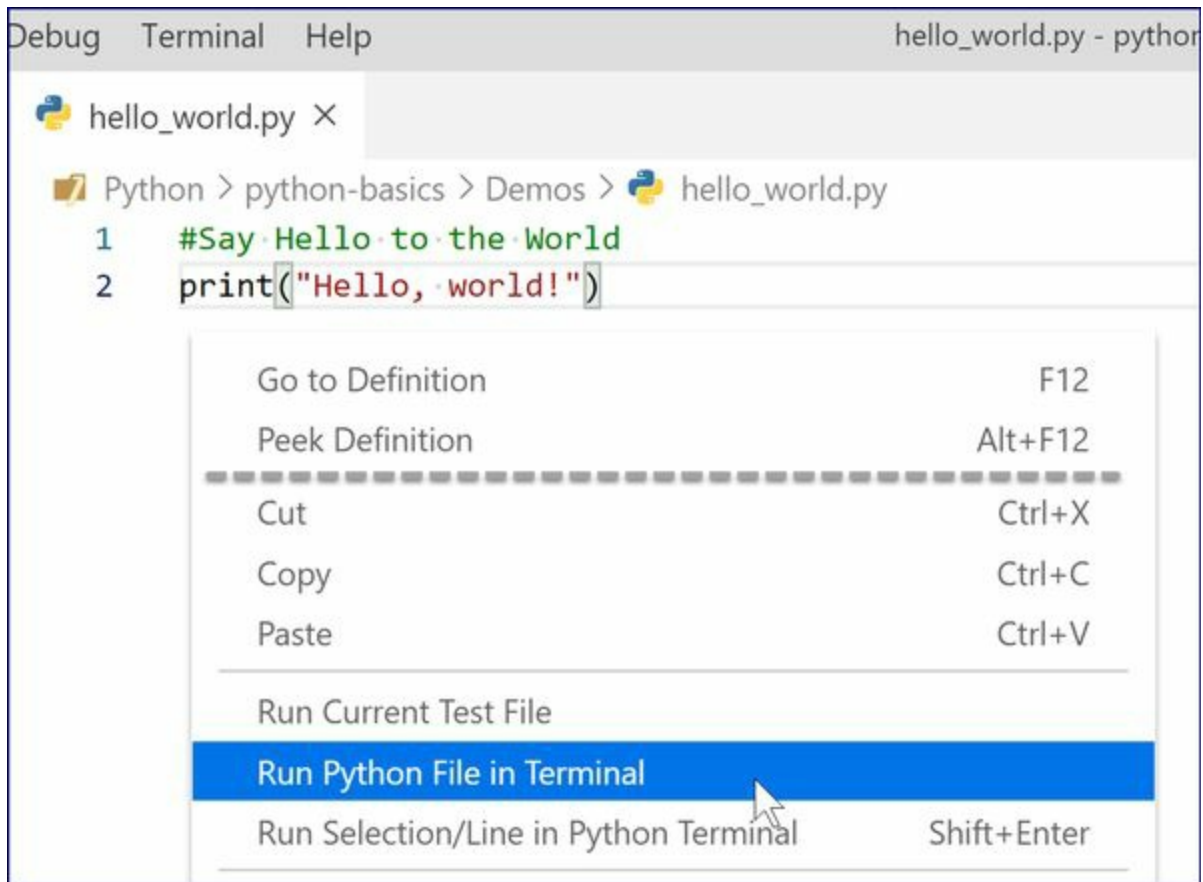
1. Open the terminal at python-basics/Demos.
2. Run `python hello_world.py`
3. The “Hello, world!” message will be displayed.

Here it is in the terminal:



Right-click and Run

Another way to run a file from within Visual Studio Code is to right-click on the open file and select **Run Python File at Terminal**:



If you don't see this option, then you don't have the [Python extension installed](#).

Exercise 1: Hello, world!

5 to 10 minutes.

1. Open `python-basics/Demos/hello_world.py` in your editor.
2. Add the following line after the "Hello, world!" line:

```
print("Hello again, world!")
```
3. Save your changes.
4. Run the Python file just as you did earlier for the demonstration.
5. The output should look like this:

```
Hello, world!  
Hello again, world!
```

Solution: python-basics/Solutions/hello_again_world.py

```
# Say Hello to the World (twice!)
print("Hello, world!")
print("Hello again, world!")
```

Code Explanation

The extra line of code will cause a second message to be printed to the standard output.

Literals

When a value is hard coded into an instruction, as "Hello" is in `print("Hello")`, that value is called a *literal* because the Python interpreter should not try to interpret it but should take it *literally*. If we left out the quotation marks (i.e., `print(Hello)`), Python would output an error because it would not know how to interpret the name `Hello`.

Either single quotes or double quotes can be used to create *string* literals. Just make sure that the open and close quotation marks match.

Literals representing numbers (e.g., 42 and 3.14) are not enclosed in quotation marks.

Python Comments

In the previous demo, you may have noticed this line of code:

```
# Say Hello to the World
```

That number sign (or hash or pound sign) indicates a comment. Everything that trails it on the same line will be ignored.

Multi-line Comments

There is no official syntax for creating multi-line comments; however, Guido van Rossum, the creator of Python, [tweeted this tip⁵](#) as a workaround:



Multi-line strings are created with triple quotes, like this:

```
"""This is a
very very helpful and informative
comment about my code!"""
```

Because multi-line strings generate no code, they can be used as pseudo-comments. In certain situations, these pseudo-comments can get confused with *docstrings*, which are used to auto-generate Python documentation, so we recommend you avoid using them until you become familiar with docstrings. Instead, use:

```
# This is a
# very very helpful and informative
# comment about my code!
```

Data Types

In Python programming, objects have different *data types*. The data type determines both what an object can do and what can be done to it. For example, an object of the data type *integer* can be subtracted from another integer, but it cannot be subtracted from a *string* of characters.

In the following list, we show the basic data types used in Python. Abbreviations are in parentheses.

1. boolean (bool) – A `True` or `False` value.
2. integer (int) – A whole number.
3. float (float) – A decimal.
4. string (str) – A sequence of Unicode⁶ characters.
5. list (list) – An ordered sequence of objects, similar to an array in other languages.
6. tuple (tuple) – A sequence of fixed length, in which an element's position is meaningful.
7. dictionary (dict) – An unordered grouping of key-value pairs.
8. set (set) – An unordered grouping of values.

We will cover all of these data types in detail.

Exercise 2: Exploring Types

10 to 15 minutes.

In this exercise, you will use the built-in `type()` function to explore different data types.

1. Open the terminal.
2. Start the Python shell by typing `python` and then pressing **Enter**:

```
PS ...\\python-basics\\Demos> python
```

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MS  
Type "help", "copyright", "credits" or "license" for more inf  
>>>
```

You are now in the Python shell.

3. To check the type of an object, use the `type()` function. For example, `type(3)` will return `<class 'int'>`:

```
>>> type(3)
<class 'int'>
```

4. Find the types of all of the following:

- A. 3
- B. 3.1
- C. '3'
- D. 'pizza'
- E. True
- F. ('1', '2', '3')
- G. ['1', '2', '3']
- H. {'1', '2', '3'}

Solution

When you're done, the output should appear as follows:

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type('3')
<class 'str'>
>>> type('pizza')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(('1', '2', '3'))
<class 'tuple'>
>>> type(['1', '2', '3'])
<class 'list'>
>>> type({'1', '2', '3'})
<class 'set'>
```

Don't worry if you're not familiar with all of the preceding data types. We will cover them all.

Class and Type

You may wonder at Python's use of the word "class" when outputting a data type. In Python, "class" and "type" mean the same thing.

Variables

Variables are used to hold data in memory. In Python, variables are untyped, meaning you do not need to specify the data type when creating the variable. You simply assign a value to a variable. Python determines the type by the value assigned.

Variable Names

Variable names are case sensitive, meaning that `age` is different from `Age`. By convention, variable names are written in all lowercase letters and words in variable names are separated by underscores (e.g., `home_address`). Variable names must begin with a letter or an underscore and may contain only letters, digits, and underscores.

Keywords

The following list of keywords have special meaning in Python and may not be used as variable names:

Python Keywords

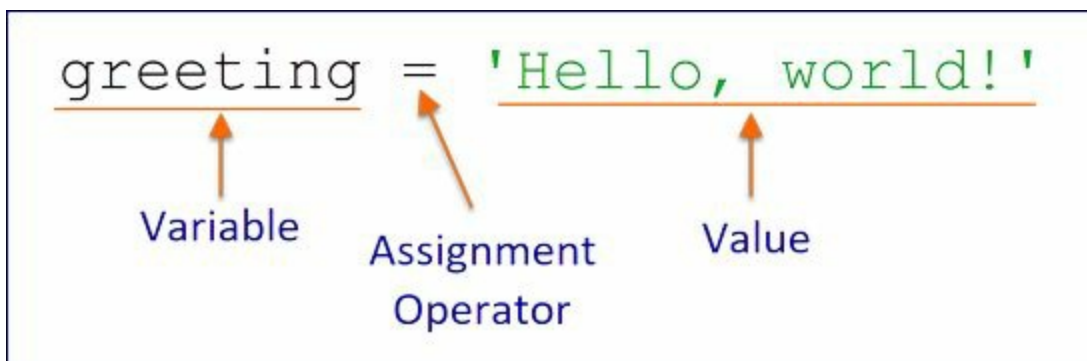
<code>and</code>	<code>del</code>	<code>if</code>	<code>pass</code>
<code>as</code>	<code>elif</code>	<code>import</code>	<code>raise</code>
<code>assert</code>	<code>else</code>	<code>in</code>	<code>return</code>
<code>async</code>	<code>except</code>	<code>is</code>	<code>True</code>
<code>await</code>	<code>False</code>	<code>lambda</code>	<code>try</code>
<code>break</code>	<code>finally</code>	<code>None</code>	<code>while</code>
<code>class</code>	<code>for</code>	<code>nonlocal</code>	<code>with</code>

continue	from	not	yield
def	global	or	

Variable Assignment

There are three parts to variable assignment:

1. Variable name.
2. Assignment operator.
3. Value assigned. This could be any data type.



Here is the “Hello, world!” script again, but this time, instead of outputting a literal, we assign a string to a variable and then output the variable:

Demo 2: python-basics/Demos/hello_variables.py

```
greeting = "Hello, world!"  
print(greeting)
```

Code Explanation

Run this Python file at the terminal. The output will be the same as it was in the [previous demo](#):

```
Hello, world!
```

Simultaneous Assignment

A very cool feature of Python is that it allows for simultaneous assignment. The syntax is as follows:

```
var_name1, var_name2 = value1, value2
```

This can be useful as a shortcut for assigning several values at once, like this:

```
>>> first_name, last_name, company = "Nat", "Dunn", "Webucator"  
>>> first_name  
'Nat'  
>>> last_name  
'Dunn'  
>>> company  
'Webucator'
```

Exercise 3: A Simple Python Script

5 to 10 minutes.

In this exercise, you will write a simple Python script from scratch. The

script will create a variable called `today` that stores the day of the week.

1. In your editor create a new file and save it as `today.py` in the `python-basics/Exercises` folder.
2. Create a variable called `today` that holds the current day of the week as literal text (i.e., in quotes).
3. Use the `print()` function to output the variable value.
4. Save your changes.
5. Test your solution.

Solution: python-basics/Solutions/today.py

```
today = "Monday"
print(today)
```

Constants

In programming, a constant is like a variable in that it is an identifier that holds a value, but, unlike variables, constants are not variable, they are constant. Good name choices, right?

Python doesn't really have constants, but as a convention, variables that are meant to act like constants (i.e., their values are not meant to be changed) are written in all capital letters. For example:

```
PI = 3.141592653589793
RED = "FF0000" # hexadecimal code for red
```

Deleting Variables

Although it's rarely necessary to do so, variables can be deleted using the `del` statement, like this:

```
>>> a = 1
>>> print(a)
1
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Notice that trying to print `a` results in an error, because after `del a`, the `a` variable no longer exists.

Writing a Python Module

A Python module is simply a file that contains code that can be reused. The `today.py` file is really a module, albeit a very simple one.

A module can be run by itself or as part of a larger program. It is too early to get into all the aspects of code reuse and modular programming, but you want to start with good habits, one of which is to use a `main()` function in your programs.

The `main()` Function

Let's look at the basic syntax of a function. A function is created using the `def` keyword like this:

Demo 3: python-basics/Demos/indent_demo.py

```
def main():
    print("I am part of the function.")
    print("I am also part of the function.")
    print("Hey, me too!")
print("Sad not to be part of the function. I've been outdented.")

main()
```

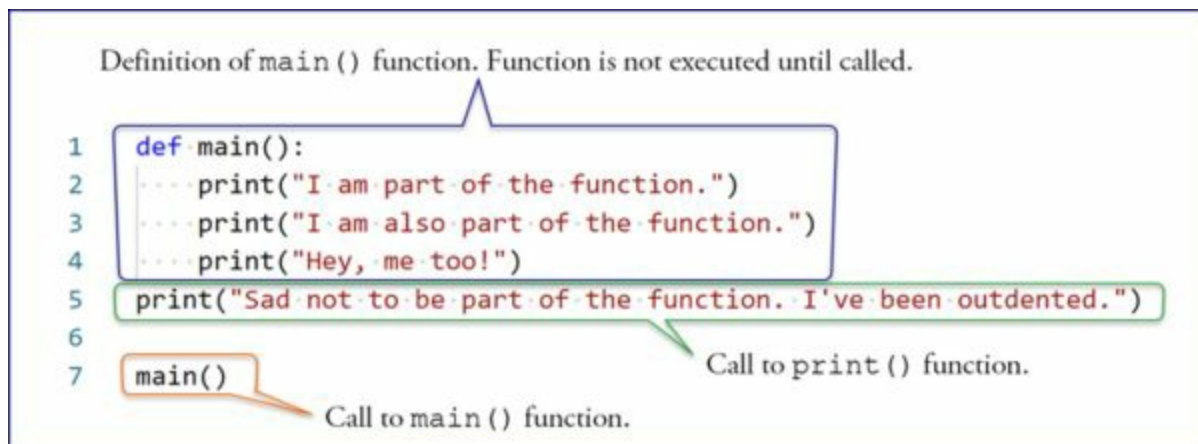
Code Explanation

Running this Python file will render the following:

```
Sad not to be part of the function. I've been outdented.
I am part of the function.
I am also part of the function.
Hey, me too!
```

Notice that the first line of output is the line that is not part of the function. That is because the function does not run until it is called, and it is called after the `print()` function that outputs “Sad not to be part of the function. I’ve been outdented.”

The code is read by the Python interpreter from top to bottom, but the function definition just defines the function; it does not *invoke the function* (programmer speak for “call the function to execute”).



pythontutor.com

Visit <http://bit.ly/pythontutor-indentdemo> to see this demo in pythontutor.com⁷, a web application for visualizing how Python executes code.

A few things to note about functions:

1. Functions are created using the `def` keyword. The content that follows the `def` keyword on the same line is called the *function signature*.
2. The convention for naming functions is the same as that for variables: use all lowercase letters and separate words with underscores.
3. In the function definition, the function name is followed by a pair of parentheses, which may contain parameters (more on that soon), and a colon.
4. The contents of the function starts on the next line and must be indented. Either spaces or tabs can be used for indenting, but spaces are preferred.
5. The first line of code after the function definition that is not indented is not part of the function and effectively marks the end of the function definition.
6. Functions are invoked using the function name followed by the parentheses (e.g., `indent_demo()`).

There is nothing magic about the name “main”. It is simply the name used by convention for the function that starts the program or module running.⁸

Grouping of Statements

A programming *statement* is a unit of code that does something. The following code shows two statements:

```
greeting = "Hello!"
```

```
print(greeting)
```

In Python, statements are grouped using indenting. As we just saw with the `main()` function, lines that are indented below the function signature are part of the function. It is important to understand this: *changes in indentation level denote code groups*. You must be careful with your indenting.

print() Function

You have already seen how to output data using the built-in `print()` function. Now, let's see how to output a variable and a literal string together. The `print()` function can take multiple arguments. By default, it will output them all separated by single spaces. For example, the following code will output "H e l l o !"

```
print('H', 'e', 'l', 'l', 'o', '!')
```

This functionality allows for the combination of literal strings and variables as shown in the following demo:

Demo 4: python-basics/Demos/variable_and_string_output.py

```
def main():
    today = "Monday"
    print("Today is", today, ".")

main()
```

Code Explanation

Run the Python file. It should render the following:

```
Today is Monday .
```

Notice the extra space before the period in the output of the last demo:



We'll get rid of that soon.

Named Arguments

As we have seen with `print()`, functions can take multiple arguments. These arguments can be *named* or *unnamed*. To illustrate, let's look at some more arguments the `print()` function can take:

```
print('H', 'e', 'l', 'l', 'o', '!', sep=' ', end='\n')
```

Those last two arguments are **named arguments**.

- `sep` is short for “separator.” It specifies the character that separates the list of objects to output. The default value is a single space, so specifying `sep=" "` doesn't change the default behavior at all.

- `end` specifies the character to print at the very end (i.e., after the last printed object). The default is a newline character (denoted with `\n`). You can use an empty string (e.g., `' '`) to specify that nothing should be printed at the end.

The following demo shows how the `sep` argument can be used to get rid of the extra space we saw in the previous example:

Demo 5: python-basics/Demos/variable_and_string_output_fixed_spacing.py

```
def main():
    today = "Monday"
    print("Today is ", today, ".", sep="")

main()
```

Code Explanation

Run the Python file. It should render the following:

```
Today is Monday.
```

Collecting User Input

Functions may or may not return values. The `print()` function, for example, does not return a value.

Python provides a built-in `input()` function, which takes a single argument: the prompt for the user's input. Unlike `print()`, the `input()` function *does* return a value: the input from the user as a string. The following demo shows how to use it to prompt the user for the day of the week.

Demo 6: python-basics/Demos/input.py

```
def main():
    today = input("What day is it? ")
    print("Wow! Today is ", today, "? Awesome!", sep="")

main()
```

Code Explanation

Run the Python file. It should immediately prompt the user:

```
What day is it?
```

Enter the day and press **Enter**. It will output something like:

```
Wow! Today is Monday? Awesome!
```

The full output will look like this:

```
What day is it? Monday
Wow! Today is Monday? Awesome!
```

Exercise 4: Hello, You!

5 to 10 minutes.

In this exercise, you will greet the user by name.

1. Open a new script. Save it as hello_you.py in python-basics/Exercises.
2. Write code to prompt for the user's name.
3. After the user has entered their name, output a greeting.
4. Save your changes.
5. Test your solution.

Solution: python-basics/Solutions/hello_you.py

```
def main():
    your_name = input("What is your name? ")
    print("Hello, ", your_name, "!", sep="")

main()
```

Code Explanation

The code should work like this:

```
PS ...\\python-basics\\Solutions> python hello_you.py
What is your name? Nat
Hello, Nat!
```

Reading from and Writing to Files

To built-in `open()` method is used to open a file from the file system. We will cover this in the [File Processing lesson](#). For now, you just need to know how to read from a file and how to write to a file.

Reading from a File

The following code shows the steps to:

1. Open a file and assign the file to a *file handler*.
2. Read the content of the file into a variable.
3. Print that variable.
4. Close the file.

```
f = open("my-file.txt") # Open my-file.txt and assign result to
content = f.read() # Read contents of file into content variable
print(content) # Print content.
f.close() # Close the file.
```


Because we referenced the file name directly, Python will look in the current directory for the file. If the file is located in a different directory, you must provide the path, either as an absolute or relative path.⁹

with Blocks

It is important to close the file to free up the memory space the handler is taking up. It's also easy to forget to do so. Fortunately, Python provides a structure that makes explicitly closing the file unnecessary:

```
with open("my-file.txt") as f:
    content = f.read()
    print(content)
```

When Python reaches the end of the `with` block, it understands that the file is no longer necessary and automatically closes it.

Writing to a File

In addition to the path to the file, the `open()` function takes a second parameter: `mode`, which indicates whether the file is being opened for reading ("r"), writing ("w"), or appending ("a"). The default value for `mode` is "r", which is why we didn't need to pass a value in when opening the file for reading. If we want to write to a file, we do have to pass in a value: "w".

```
with open("my-file2.txt", "w") as f:
    f.write("Hello, world!!!!")
```

When you run the code above, it will open my-file2.txt if it exists and overwrite the file with the text you write to it. If it doesn't find a file named my-file2.txt, it will create it.

Exercise 5: Working with Files

In this exercise, you will open two files that contain lists of popular boys and girls names from 1880,^{[10](#)} read their contents into two variables, and then write the combined content of the two files into a new file.

1. Open a new script. Save it as `files.py` in `python-basics/Exercises`.
2. Write code to open `python-basics/data/1880-boys.txt` and read its contents into a variable called `boys`.
3. Write code to open `python-basics/data/1880-girls.txt` and read its contents into a variable called `girls`.
4. Write code to open a new file named `1880-all.txt` in the `python-basics/data` folder and write the combined contents of the `boys` and `girls` variables into the file. Note that you can combine the two strings like this:

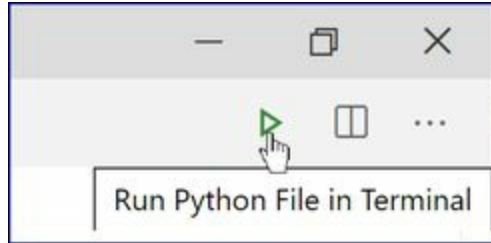
```
boys + "\n" + girls
```

That will place one newline between the content in `boys` and the content in `girls`.

5. Save your file.
6. Test your solution. When you run it, it should create the new file. Look in the `data` folder for the `1880-all.txt` file. Does it exist? If so, open it up. Does it have a list of all the boys and girls names?

Run Python File in Terminal

You may have discovered that you can run Python files in the terminal using the green triangle in the upper-right of Visual Studio Code:



By default, VS Code will run the file from the Workspace root folder and search for any files being referenced with relative paths from that folder. As a result, you may get `FileNotFoundError` errors. You can fix this by changing a setting:

1. From the **File** menu, select **Preferences > Settings**.
2. Search for "execute in file dir".
3. Check the **Python > Terminal: Execute in File Dir** setting.



Solution: `python-basics/Solutions/files.py`

```
with open("../data/1880-boys.txt") as f_boys:
    boys = f_boys.read()

with open("../data/1880-girls.txt") as f_girls:
    girls = f_girls.read()

with open("../data/1880-all.txt", "w") as f:
    f.write(boys + "\n" + girls)
```

Conclusion

In this lesson, you have begun to work with Python. Among other things, you have learned to use variables, to output data, to collect user input, and to write simple Python functions and modules.

LESSON 3

Functions and Modules

Topics Covered

- Defining and calling functions.
- Parameters and arguments.
- Default values for parameters.
- Variable scope.
- Return values.
- Creating and importing modules.

It is your duty as a magistrate, and I believe and hope that your feelings as a man will not revolt from the execution of those functions on this occasion.

– Frankenstein, Mary Shelley

You have seen some of Python's built-in functions. In this lesson, you will learn to write your own.

Defining Functions

We discussed functions a little in the last lesson, but let's quickly review the syntax. Functions are defined in Python using the `def` keyword. The syntax is as follows:

```
def function_name():  
    # content of function is indented  
    do_something()  
# This is no longer part of the function  
do_something_else()
```

Here is a modified solution to the "Hello, You!" exercise:

Demo 7: functions/Demos/hello_you.py

```
def say_hello():  
    your_name = input("What is your name? ")  
    print("Hello, ", your_name, "!", sep="")  
  
def main():  
    say_hello()  
  
main()
```

Code Explanation

The code works in the same way, but the meat of the program has been moved out of the `main()` function and into another function. This is common. Usually, the `main()` function handles the flow of the program, but the actual “work” is done by other functions in the module.

The following expanded demo further illustrates how the `main()` function can be used to control flow.

Demo 8: functions/Demos/hello_you_expanded.py

```
def say_hello():
    your_name = input("What is your name? ")
    insert_separator()
    print("Hello, ", your_name, "!", sep="")

def insert_separator():
    print("===")

def recite_poem():
    print("How about a Monty Python poem?")
    insert_separator()
    print("Much to his Mum and Dad's dismay")
    print("Horace ate himself one day.")
    print("He didn't stop to say his grace,")
    print("He just sat down and ate his face.")

def say_goodbye():
    print("Goodbye!")

def main():
    say_hello()
    insert_separator()
    recite_poem()
    insert_separator()
    say_goodbye()

main()
```

Code Explanation

The preceding code will render the following:

```
What is your name? Nat
===
Hello, Nat!
===
```

```
How about a Monty Python poem?
===
Much to his Mum and Dad's dismay
Horace ate himself one day.
He didn't stop to say his grace,
He just sat down and ate his face.
===
Goodbye!
```

Not All Modules are Programs

Not all Python modules are programs. Some modules are only meant to be used as helper files for other programs. Sometimes these modules are more or less generic, providing functions that could be useful to many different types of programs. And sometimes these modules are written to work with a specific program. Modules that aren't programs probably wouldn't have a `main()` function.

Variable Scope

Question: Why doesn't the `say_goodbye()` function use the user's name (e.g., `print("Goodbye, ", your_name)`)?

Answer: It doesn't know what `your_name` is.

Variables declared within a function are *local* to that function. Consider the following code:

Demo 9: functions/Demos/local_var.py

```
def set_x():  
    x = 1  
  
def get_x():  
    print(x)  
  
def main():  
    set_x()  
    get_x()  
  
main()
```

Code Explanation

Run this and you'll get an error similar to the following:

```
NameError: name 'x' is not defined
```

That's because `x` is defined in the `set_x()` function and is therefore *local* to that function.

A good Python IDE, like Visual Studio Code, will let you know when it detects such an error. In VS Code, a squiggly underline will appear beneath the undefined variable. Depending on how your settings are configured, you may be able to hover over the variable to see the error:

```
Python > functions > Demos > local_var.py > ...  
1 def set_x(x  
2     x=1  
3  
4 def get_x(  
5     print(x)  
6  
7 def main():  
8     set_x()  
9     get_x()  
10  
11 main()
```

Undefined variable: 'x' Python(undefined-variable)
[Peek Problem](#) No quick fixes available

Global Variables

Global variables are defined outside of a function as shown in the following demo:

Demo 10: functions/Demos/global_var.py

```
x = 0

def set_x():
    x = 1
    print("from set_x():", x)

def get_x():
    print("from get_x():", x)

def main():
    set_x()
    get_x()

main()
```

Code Explanation

`x` is first declared outside of a function, which means that it is a *global* variable.

Question: What do you think the `get_x()` function will print: 0 or 1?

Answer: It will print 0. That's because the `x` in `set_x()` is not the same as the global `x`. The former is local to the `set_x()` function.

Global variables, by default, can be referenced but not modified within functions:

- When a variable is *referenced* within a function, Python first looks for a local variable by that name. If it doesn't find one, then it looks for a global variable.
- When a variable is *assigned* within a function, it will be a local variable, even if a global variable with the same name already exists.

To modify global variables within a function, you must explicitly state that you want to work with the global variable. That's done using the `global` keyword, like this:

Demo 11: functions/Demos/global_var_in_function.py

```
x = 0

def set_x():
    global x
    x = 1

def get_x():
    print(x)

def main():
    set_x()
    get_x()

main()
```

Code Explanation

Now, the `set_x()` function explicitly references the global variable `x`, so the code will print 1.

Naming global variables?

Some developers feel that any use of global variables is bad programming. While we won't go that far, we do have two recommendations:

1. Prefix your global variables with an underscore¹¹ (e.g., `_x`). That makes it clear that the variable is global and minimizes the chance of it getting confused with a local variable of the same name. It is also a convention that lets developers know that those variables are not meant to be used outside the module (i.e., by programs importing the module).
2. When possible, rather than using global variables, design your code so that values can be passed from one function to another

using parameters (see next section).

Function Parameters

Consider the `insert_separator()` function in the `hello_you_expanded.py` file that we saw earlier:

```
def insert_separator():  
    print("===")
```

What if we wanted to have different types of separators? One solution would be to create multiple functions, like `insert_large_separator()` and `insert_small_separator()`, but that can get pretty tiresome and hard to maintain. A better solution is to use function parameters using the following syntax:

```
def function_name(param1, param2, param3):  
    do_something(param1, param2, param3)
```

Here is our “Hello, You!” program using parameters:

Demo 12: functions/Demos/hello_you_with_params.py

```
def say_hello(name):
    print('Hello, ', name, '!', sep='')

def insert_separator(s):
    print(s, s, s, sep="")

def recite_poem():
    print("How about a Monty Python poem?")
    insert_separator("-")
    print("Much to his Mum and Dad's dismay")
    print("Horace ate himself one day.")
    print("He didn't stop to say his grace,")
    print("He just sat down and ate his face.")

def say_goodbye(name):
    print('Goodbye, ', name, '!', sep='')

def main():
    your_name = input('What is your name? ')
    insert_separator("-")
    say_hello(your_name)
    insert_separator("=")
    recite_poem()
    insert_separator("=")
    say_goodbye(your_name)

main()
```

Code Explanation

Now that `insert_separator()` takes the separating character as an argument, we can use it to separate lines with any character we like.

We have also modified `say_hello()` and `say_goodbye()` so that they receive the name of the person they are addressing as an argument. This has a couple of advantages:

1. We can move `your_name = input('What is your name? ')` to the `main()` function so we can pass `your_name` into both `say...` functions.
2. We can move the call to `insert_separator()` out of the `say_hello()` function as the separator doesn't have anything to do with saying "hello."

Parameters vs. Arguments

The terms *parameter* and *argument* are often used interchangeably, but there is a difference:

Parameters are the variables in the function definition. They are sometimes called *formal parameters*.

Arguments are the values passed into the function and assigned to the parameters. They are sometimes called *actual parameters*.

```
Python > functions > Demos > parameters_vs_arguments.py > ...
1  def ask_something(something):
2      print(something)
3
4
5
6  ask_something("What is your quest?")
-
```

Using Parameter Names in Function Calls

When calling a function, you can specify the parameter by name when passing in an argument. When you do so, it's called passing in a *keyword argument*. For example, you can call the following `divide()` function in several ways:

```
def divide(numerator, denominator):
```



```
        return numerator / denominator

divide(10, 2)
divide(numerator=10, denominator=2)
divide(denominator=2, numerator=10)
divide(10, denominator=2)
```

As you can see, using keyword arguments allows you to pass in the arguments in an arbitrary order. You can combine non-keyword arguments with keyword arguments in a function call, but you must pass in the non-keyword arguments first.

Later, we'll see that a function can be written to require keyword arguments.

Exercise 6: A Function with Parameters

15 to 25 minutes.

In this exercise, you will write a function for adding two numbers together.

1. Open [functions/Exercises/add_nums.py](#) in your editor and review the code.
2. Now, run the file in Python. The output should look like this:

```
3 + 6 = 9
10 + 12 = 22
```

3. Replace the two crazy `add...()` functions with an `add_nums()` function that accepts two numbers, adds them together, and outputs the equation.

Exercise Code: functions/Exercises/add_nums.py

```
def add_3_and_6():
    total = 3 + 6
    print('3 + 6 = ', total)

def add_10_and_12():
    total = 10 + 12
    print('10 + 12 = ', total)

def main():
    add_3_and_6()
    add_10_and_12()

main()
```

Code Explanation

The first function adds 3 and 6. The second function adds 10 and 12. These functions are only useful if you want to add those specific numbers.

Solution: functions/Solutions/add_nums.py

```
def add_nums(num1, num2):
    total = num1 + num2
    print(num1, '+', num2, ' = ', total)

def main():
    add_nums(3, 6)
    add_nums(10, 12)

main()
```

Code Explanation

The `add_nums()` function is flexible and reusable. It can add any two numbers.

Default Values

Parameters that do not have default values require arguments to be passed in. You can assign default values to parameters using the following syntax:

```
def function_name(param=default):
    do_something(param)
```

For example, the following code would make the “=” sign the default separator:

```
def insert_separator(s="="):
    print(s, s, s, sep="")
```

When an argument is not passed into a parameter that has a default value, the default is used.

See [functions/Demos/hello_you_with_defaults.py](#) to see a working demo.

Exercise 7: Parameters with Default Values

15 to 25 minutes.

In this exercise, you will write a function that can add two, three, four, or five numbers together.

1. Open `functions/Exercises/add_nums_with_defaults.py` in your editor.
2. Notice the `add_nums()` function takes five arguments, adds them together, and outputs the sum.
3. Modify `add_nums()` so that it can accept all of the following calls:

```
add_nums(1, 2)
add_nums(1, 2, 3, 4, 5)
add_nums(11, 12, 13, 14)
add_nums(101, 201, 301)
```

4. For now, it's okay for the function to print out 0s for values not passed in, like this:

```
1 + 2 + 0 + 0 + 0 = 3
1 + 2 + 3 + 4 + 5 = 15
11 + 12 + 13 + 14 + 0 = 50
101 + 201 + 301 + 0 + 0 = 603
```

Exercise Code: functions/Exercises/add_nums_with_defaults.py

```
def add_nums(num1, num2, num3, num4, num5):
    total = num1 + num2 + num3 + num4 + num5
    print(num1, '+', num2, '+', num3, '+', num4, '+', num5, ' =

def main():
    add_nums(1, 2, 0, 0, 0)
    add_nums(1, 2, 3, 4, 5)
    add_nums(11, 12, 13, 14, 0)
    add_nums(101, 201, 301, 0, 0)

main()
```

Solution: functions/Solutions/add_nums_with_defaults.py

```
def add_nums(num1, num2, num3=0, num4=0, num5=0):
    total = num1 + num2 + num3 + num4 + num5
    print(num1, '+', num2, '+', num3, '+', num4, '+', num5, ' = ', total)

def main():
    add_nums(1, 2)
    add_nums(1, 2, 3, 4, 5)
    add_nums(11, 12, 13, 14)
    add_nums(101, 201, 301)

main()
```

Code Explanation

We have given the last three parameters default values of 0, making them optional. The first two parameters don't have default values, so they are still required.

Returning Values

Functions can return values. The `add_nums()` function we have been working with does more than add the numbers passed in, it also prints them out. You can imagine wanting to add numbers for some other purpose than printing them out. Or you might want to print the results out in a different way. We can change the `add_nums()` function so that it just adds the numbers together and returns the sum. Then our program can decide what to do with that sum. Take a look at the following code:

Demo 13: functions/Demos/add_nums_with_return.py

```
def add_nums(num1, num2, num3=0, num4=0, num5=0):  
    total = num1 + num2 + num3 + num4 + num5  
    return total  
  
def main():  
    result = add_nums(1, 2)  
    print(result)  
    result = add_nums(result, 3)  
    print(result)  
    result = add_nums(result, 4)  
    print(result)  
    result = add_nums(result, 5)  
    print(result)  
    result = add_nums(result, 6)  
    print(result)  
  
main()
```

Code Explanation

The `add_nums()` function now returns the sum to the calling function via the `return` statement. We assign the result to a local variable named `result`. Then we print `result` and pass it back to `add_nums()` in subsequent calls.

Note that once a function has returned a value, the function is finished executing and control is transferred back to the code that invoked the function.

Importing Modules

As we saw, part of the beauty of writing functions is that they can be reused. Imagine you write a really awesome function. Or even better, a module with a whole bunch of really awesome functions in it. You'd want to make those functions available to other modules so you (and

other Python developers) could make use of them elsewhere.

Modules can import other modules using the `import` keyword as shown in the following example:

Demo 14: functions/Demos/import_example.py

```
import add_nums_with_return

total = add_nums_with_return.add_nums(1, 2, 3, 4, 5)
print(total)
```

Code Explanation

Now, the `add_nums()` function from `add_nums_with_return.py` is available in `import_example.py`; however, it must be prefixed with “`add_nums_with_return.`” (the module name) when called.

The main() Function

It is common for a module to check to see if it is being imported by checking the value of the special `__name__` variable. Such a module will only run its `main()` function if `__name__` is equal to `'__main__'`, indicating that it is not being imported. The code usually goes at the bottom of the module and looks like this:

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

Note that those are two underscores before and after `name` and before and after `main`.

A short video explanation of this is available at https://bit.ly/python_main.

Another way to import functions from another module is to use the following syntax:

```
from module_name import function1, function2
```

For example:

```
from add_nums_with_return import add_nums
```

```
total = add_nums(1, 2, 3, 4, 5)  
print(total)
```

When you use this approach, it is not necessary to prefix the module name when calling the function. However, it's possible to have naming conflicts, so be careful.

Another option, which is helpful for modules with long names, is to create an alias for a module, so that you do not have to type its full name:

```
import add_nums_with_return as anwr  
  
total = anwr.add_nums(1, 2, 3, 4, 5)
```

Using aliases is also a way of preventing naming conflicts. If you import `do_this` from `foo` and `do_this` from `bar`, you can use an alias to call one of them `do_that`:

```
from foo import do_this  
from bar import do_this as do_that
```

Module Search Path

The Python interpreter must locate the imported modules. When `import` is used within a script, the interpreter searches for the imported module in the following places sequentially:

1. The current directory (same directory as the script doing the importing).
2. The library of standard modules.[12](#)
3. The paths defined in `sys.path`, which you can see by running the

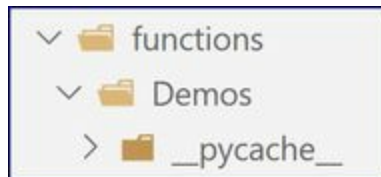
following code at the Python shell:

```
>>> import sys
>>> sys.path
```

This will output a list of paths, which are searched in order for the imported module.

pyc Files

Files with a `.pyc` extension are *compiled* Python files. They are automatically created in a `__pycache__` folder the first time a file is imported:



These files are created so that modules you import don't have to be compiled every time they run. You can just leave those files alone. They will automatically be created/updated each time you import a module that is new or has been changed.

Methods vs. Functions

You have already seen some built-in functions, like `print()` and `input()`. You have also written some of your own, like `insert_separator()` and `divide()`. In the upcoming lessons, you will learn to use many more of Python's built-in functions. You will also learn about *methods*, which are similar to functions, except that they are called on an object using the syntax `object_name.method_name()`.

An example of a simple built-in function is `len()`, which returns the length of the passed-in object:

```
>>> len('Webucator')
9
```

An example of a method of a string object is `upper()`, which returns the string it is called upon in all uppercase letters:

```
>>> Webucator'.upper()  
'WEBUCATOR'
```

Again, you will work with many functions and methods in upcoming lessons.

Conclusion

In this lesson, you have learned to define functions with or without parameters. You have also learned about variable scope and how to import modules.

LESSON 4

Math

Topics Covered

- Basic math in Python.
- The `math` module.
- The `random` module.

I had been to school most all the time and could spell and read and write just a little, and could say the multiplication table up to six times seven is thirty-five, and I don't reckon I could ever get any further than that if I was to live forever. I don't take no stock in mathematics, anyway.

– *Adventures of Huckleberry Finn, Mark Twain*

Python includes some built-in math functions and some additional built-in libraries that provide extended math (and related) functionality. In this lesson, we'll cover the built-in functions and the `math` and `random` libraries.

Arithmetic Operators

The following table lists the arithmetic operators in Python. Most will be familiar. We'll explain the others.

Arithmetic Operators

Operator	Description
+	Addition
	5 + 2 returns 7
-	Subtraction
	5 - 2 returns 3
*	Multiplication

	5 * 2 returns 10
/	Division
	5/2 returns 2.5
%	Modulus
	5 % 2 returns 1
**	Power
	5**2 is 5 to the power of 2. It returns 25
//	Floor Division
	5 // 2 returns 2

Here are the examples in Python:

```
>>> 5+2
7
>>> 5-2
3
>>> 5*2
10
>>> 5/2
2.5
>>> 5%2
1
>>> 5**2
25
>>> 5//2
2
```

Modulus and Floor Division

You may remember doing this in elementary school math:

$$\begin{array}{r} 2^r \ 1 \\ 2 \overline{) 5} \end{array}$$

The remainder of 5 divided by 2 is 1.

The *modulus* operator (%) is used to find the remainder after division:

```
>>> 5 % 2
1
>>> 11 % 3
2
>>> 22 % 4
2
>>> 22 % 3
1
>>> 10934 % 324
242
```

Modulus and Negative Numbers

The results of modulus operations with negative numbers can be surprising, and are different in different programming languages. Python uses the following formula: $x - y*(x//y)$, which will always return a positive number. This is mostly academic. You will likely never have to deal with this, so we will not dig further into it.

The *floor division* operator (//) is the same as regular division, but rounded down:

```
>>> 5 // 2
2
>>> 11 // 3
3
>>> 22 // 4
```

```
5
>>> 22 // 3
7
>>> 10934 // 324
33
>>> -5 // 2 # rounded down, meaning towards negative infinity
-3
```

Exercise 8: Floor and Modulus

5 to 10 minutes.

In this exercise, you will write a small function called `divide()` that takes a numerator and denominator and prints out a response that a fifth grader would understand (e.g., “5 divided by 2 equals 2 with a remainder of 1”).

1. Open [math/Exercises/floor_modulus.py](#) in your editor.
2. Write the `divide()` function.
3. Run the module. It should output the following:

```
5 divided by 2 equals 2 with a remainder of 1
6 divided by 3 equals 2 with a remainder of 0
12 divided by 5 equals 2 with a remainder of 2
1 divided by 2 equals 0 with a remainder of 1
```


Exercise Code: math/Exercises/floor_modulus.py

```
# write the divide() function
```

```
def main():  
    divide(5, 2)  
    divide(6, 3)  
    divide(12, 5)  
    divide(1, 2)
```

```
main()
```

Solution: math/Solutions/floor_modulus.py

```
def divide(num, den):  
    remainder = num % den  
    floor = num // den  
    print(num, "divided by", den, "equals",  
          floor, "with a remainder of", remainder)
```

```
def main():  
    divide(5, 2)  
    divide(6, 3)  
    divide(12, 5)  
    divide(1, 2)
```

```
main()
```

Assignment Operators

The following table lists the assignment operators in Python.

Assignment Operators

Operator	Description
=	Basic assignment
	a = 2
+=	One step addition and assignment
	a += 2 same as a = a + 2
-=	One step subtraction and assignment
	a -= 2 same as a = a - 2
*=	One step multiplication and assignment
	a *= 2 same as a = a * 2
/=	One step division and assignment
	a /= 2 same as a = a / 2

Here are the examples from the table above in a Python script:

```
>>> a = 5  
>>> a
```

```

5
>>> a += 2
>>> a
7
>>> a -= 2
>>> a
5
>>> a *= 2
>>> a
10
>>> type(a)
<class 'int'>
>>> a /= 2
>>> a
5.0
>>> type(a) # Notice division returns a float
<class 'float'>

```

Notice that `a` changes from an integer to a float when division is performed on it.

Precedence of Operations

The order of operations for arithmetic operators is:

1. `**`
2. `*`, `/`, `//`, `%`
3. `+`, `-`

You can use parentheses to change the order of operations and give an operation higher precedence. For example:

- `6 + 3 / 3` is equal to `6 + 1` and will yield 7.
- But `(6 + 3) / 3` is equal to `9 / 3` and will yield 3.

Operations of equal precedence are evaluated from left to right, so `6 / 2 * 3` and `(6 / 2) * 3` are the same.

Built-in Math Functions

Python's [built-in functions](#)¹³ include several math functions.

int(x)

`int(x)` returns `x` converted to an integer.

When converting floats, `int(x)` strips everything after the decimal point, essentially rounding down for positive numbers and rounding up for negative numbers.

When converting strings, the string object must be an accurate representation of an integer (e.g., `'5'`, but not `'5.0'`).

```
>>> int(5)
5
>>> int('5')
5
>>> int(5.4)
5
>>> int(5.9)
5
>>> int(-5.9)
-5
>>> int('5.4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '5.4'
```

float(x)

`float(x)` returns `x` converted to a float.

```
>>> float(5)
5.0
>>> float('5')
5.0
```

```
>>> float(5.4)
5.4
>>> float('5.4')
5.4
>>> float('-5.99')
-5.99
>>> float(-5.99)
-5.99
```

abs(x)

`abs(x)` returns the absolute value of `x` as an integer or a float.

```
>>> abs(-5)
5
>>> abs(5)
5
>>> abs(-5.5)
5.5
>>> abs(5.5)
5.5
```

min() and **max()** [14](#)

The `min(args)` function returns the smallest value of the passed-in arguments.

The `max(args)` function returns the largest value of the passed-in arguments.

```
>>> min(2, 1, 3)
1
>>> min(3.14, -1.5, -300)
-300
>>> max(2, 1, 3)
3
>>> max(3.14, -1.5, -300)
```

min() and max() with Iterables

The `min()` and `max()` functions can also take an *iterable* containing values to compare. We will cover this in the [Iterables lesson](#).

pow(base, exp[, mod])

Passing two arguments to the `pow()` function is the equivalent of using the power operator (`**`):

```
pow(base, exp)
```

is the same as

```
base**exp
```

For example:

```
>>> pow(4, 2)
16
>>> 4**2
16
```

`pow()` can take a third argument: `mod`. `pow(base, exp, mod)` is functionally equivalent to `base**exp % mod`:

```
>>> pow(4, 2, 3)
1
>>> 4**2 % 3
1
```

Square Brackets in Code Notation

Square brackets in code notation indicate that the contained portion is optional. Consider the `pow()` function signature:

```
pow(base, exp[, mod])
```

This means that the `mod` parameter is optional.

round(number[, ndigits])

`round(number[, ndigits])` returns `number` as a number rounded to `ndigits` digits after the decimal. If `ndigits` is `0` or omitted then the function rounds to the nearest integer. If `ndigits` is `-1` then it rounds to the nearest 10 (e.g., `round(55, -1)` returns `60`).

```
>>> round(55, -1)
60
>>> round(3.14)
3
>>> round(-3.14)
-3
>>> round(3.14, 1)
3.1
>>> round(3.95, 1)
4.0
>>> round(1111, 0)
1111
```

sum(iter[, start])

The `sum()` function takes an iterable (e.g., a list) and adds up all of its elements. We will cover this in the [Iterables lesson](#).

The math Module

The `math` module is built in to Python and provides many useful methods. We cover some of them here. [15](#)

To access any of these methods, you must first import `math`:

```
>>> import math
```

Common Methods of the math Module

math.ceil(x)

x rounded up to the nearest whole number as an integer.

```
>>> math.ceil(5.4)
6
>>> math.ceil(-5.4)
-5
>>>
```

math.floor(x)

x rounded down to the nearest whole number as an integer.

```
>>> math.floor(5.6)
5
>>> math.floor(-5.6)
-6
```

math.trunc(x)

x with the fractional truncated, effectively rounding towards 0 to the nearest whole number as an integer.

```
>>> math.trunc(5.6)
5
>>> math.trunc(-5.6)
-5
```

math.fabs(x)

The absolute value of float x. This is similar to the built-in `abs(x)`

function except that `math.fabs(x)` always returns a float whereas `abs(x)` returns a number of the same data type as `x`.

```
>>> math.fabs(-5)
5.0
>>> abs(-5)
5
```

`math.factorial(x)`

The factorial of `x`. This is often written as $x!$, but not in Python!

```
>>> math.factorial(3)
6
>>> math.factorial(5)
120
```

`math.pow(x, y)`

`x` raised to the power `y` as a float.

```
>>> math.pow(5, 2)
25.0
```

`math.sqrt(x)`

The square root of `x` as a float.

```
>>> math.sqrt(25)
5.0
```

The `math` module also contains two constants: `math.pi` and `math.e`, for Pi and *e* as used in the [natural logarithm](#).^{[16](#)}

Additional math Methods

The `math` library offers many additional methods, including:

- Logarithmic methods (e.g., `math.log(x)`)
- Trigonometric methods (e.g., `math.sin(x)`)
- Angular conversion methods (e.g., `math.degrees(x)`)
- Hyperbolic methods (e.g., `math.sinh(x)`)

The random Module

The `random` module is also built into Python and includes methods for creating and selecting random values.

To access any of these methods, you must first import `random`:

```
>>> import random
```

Common Methods of the random Module^{[17](#)}

`random.random()`

Random float between 0 and 1.

```
>>> random.random()  
0.5715141345521301
```

`random.randint(a, b)`

Random integer between (and including) `a` and `b`.

```
>>> random.randint(1, 10)  
7 # integer between 1 and 10
```

`random.randrange(b)`

Random integer between 0 and `b-1`.

```
>>> random.randrange(10)
3 # integer between 0 and 9
```

random.randrange(a, b)

Random integer between a and b-1.

```
>>> random.randrange(1, 10)
5 # integer between 1 and 9
```

random.randrange(a, b, step)

Random integer at step between (and including) a and b-1.

```
>>> random.randrange(1, 10, 2)
3 # one of 1, 3, 5, 7, 9
```

random.uniform(a, b)

Random float between (and including) a and b.

```
>>> random.uniform(1, 10)
8.028088082797572 # a float between 1 and 10
```

random.choice(seq) and random.shuffle(seq)

random.choice(seq) returns a random element in the sequence seq.

random.shuffle(seq) shuffles the sequence seq in place.

[Sequences are covered in an upcoming lesson.](#)

Seeding

random.seed(a) is used to initialize the random number generator. The value of a will determine how random numbers are selected. The following code illustrates this:

```
>>> import random
>>> random.seed(1)
>>> random.randint(1, 100)
18
>>> random.randint(1, 100)
73
>>> random.randint(1, 100)
98
# reseed
>>> random.seed(1)
>>> random.randint(1, 100)
18
>>> random.randint(1, 100)
73
>>> random.randint(1, 100)
98
```

Notice that the random numbers generated depend on the seed. If you run this same code locally, you should get the same random integers. This can be useful for testing.

By default, `random.seed()` uses the current system time to ensure that `seed()` is seeded with a different number every time it runs, so that the random numbers generated will be different each time.

Exercise 9: How Many Pizzas Do We Need?

15 to 25 minutes.

In this exercise, you will write a program from scratch. Your program should prompt the user to input the information required:

- The number of people.
- The number of slices each person will eat.
- The number of slices in each pizza pie.

Using that information, your program must calculate how many pizzas

are needed to feed everyone. It should work like this:

How many people are eating? **5**
How many slices per person? **2.5**
How many slices per pie? **8**
You need 2 pizzas to feed 5 people.
There will be 3.5 leftover slices.

How many people are eating? **25**
How many slices per person? **2**
How many slices per pie? **8**
You need 7 pizzas to feed 25 people.
There will be 6.0 leftover slices.

Solution: math/Solutions/pizza_slices.py

```
import math

def main():
    people = int(input("How many people are eating? "))
    slices_per_person = float(input("How many slices per person? "))
    slices = slices_per_person * people

    slices_per_pie = int(input("How many slices per pie? "))
    pizzas = math.ceil(slices / slices_per_pie)

    print("You need", pizzas, "pizzas to feed", people, "people.")

    total_slices = slices_per_pie * pizzas
    slices_left = total_slices - slices
    print("There will be", slices_left, "leftover slices.")

main()
```

Exercise 10: Dice Rolling

15 to 25 minutes.

In this exercise, you will write a dice-rolling program from scratch. The program should include two functions: `main()` and `roll_die()`. The `roll_die()` function should take one parameter: `sides`, and return a random roll between 1 and `sides`.

The `main()` function should call the `roll_die()` function three times and keep a tally of the total. After each roll, it should print out the value of the roll, the number of rolls, and the total. At the end, it should output the average of the three rolls. The output will be similar to the following:

```
You rolled a 3
Total after first roll: 3
```

You rolled a 5

Total after 2 rolls: 8

You rolled a 3

Total after 3 rolls: 11

Your average roll was 3.67

Thanks for playing.

Solution: math/Solutions/dice.py

```
import random

def roll_die(sides=6):
    num_rolled = random.randint(1, sides)
    return num_rolled

def main():
    sides = 6
    total = 0

    num_rolls = 1
    roll = roll_die(sides)
    print("You rolled a", roll)
    total += roll
    print("Total after first roll:", total)

    num_rolls += 1
    roll = roll_die(sides)
    print("You rolled a", roll)
    total += roll
    print("Total after", num_rolls, "rolls:", total)

    num_rolls += 1
    roll = roll_die(sides)
    print("You rolled a", roll)
    total += roll
    print("Total after", num_rolls, "rolls:", total)

    average = round(total / num_rolls, 2)
    print("Your average roll was", average)

    print("Thanks for playing.")

main()
```

Conclusion

In this lesson, you have learned to do basic math in Python and to use the `math` and `random` modules for extended math functionality.

LESSON 5

Python Strings

Topics Covered

- String basics.
- Special characters.
- Multi-line strings.
- Indexing and slicing strings.
- Common string operators and methods.
- Formatting strings.
- Built-in string functions.

‘Yes,’ they say to one another, these so kind ladies, ‘he is a stupid old fellow, he will see not what we do, he will never observe that his sock heels go not in holes any more, he will think his buttons grow out new when they fall, and believe that strings make theirselves.’

– Little Women, Louisa May Alcott

According to the [Python documentation](#)¹⁸, “Strings are immutable sequences of Unicode code points.” Less technically speaking, strings are sequences of characters.¹⁹ The term *sequence* in Python refers to an ordered set. Other common sequence types are lists, tuples, and ranges, all of which we will cover.

Quotation Marks and Special Characters

Strings can be created with single quotes or double quotes. There is no difference between the two.

Escaping Characters

To create a string that contains a single quote (e.g., where'd you get the coconuts?), enclose the string in double quotes:

```
phrase = "Where'd you get the coconuts?"
```

Likewise, to create a string that contains a double quote (e.g., The soldier asked, "Are you suggesting coconuts migrate?"), enclose the string in single quotes:

```
phrase = 'The soldier asked, "Are you suggesting coconuts migrate?"'
```

Sometimes you will want to output single quotes within a string denoted by single quotes or double quotes within a string denoted by double quotes. In such cases, you will need to *escape* the quotation marks using a backslash (\), like this:

```
>>> phrase = "The soldier said, \"You've got two empty halves of  
>>> print(phrase)  
The soldier said, "You've got two empty halves of a coconut."
```

```
# or
```

```
>>> phrase = 'The soldier said, "You\'ve got two empty halves of  
>>> print(phrase)  
The soldier said, "You've got two empty halves of a coconut."
```

Notice that the printed output does not contain the backslashes.

Special Characters

The backslash can also be used to escape characters that have special meaning, such as the backslash itself:

```
>>> phrase = "Use an extra backslash to output a backslash: \\  
>>> print(phrase)  
Use an extra backslash to output a backslash: \
```

Two other common special characters are the newline (\n) and

horizontal tab (\t):

```
>>> print('Equation\tSolution\n 55 x 11\t 605\n 132 / 6\t 22')
Equation      Solution
 55 x 11      605
 132 / 6      22
```

Escape Sequences

Escape Sequence	Meaning
\'	Single quote
\"	Double quote
\\	Backslash
\n	Newline
\t	Horizontal tab

Raw Strings

Sometimes, a string might have a lot of backslashes in it that are just meant to be plain old backslashes. The most common example is a file path. For example:

```
'C:\news\today.txt'
```

Watch what happens when that string is assigned to a variable:

```
>>> my_path = 'C:\news\today.txt'
>>> print(my_path)
C:
ews      oday.txt
```

When we print `my_path`, the `\n` and `\t` characters get printed as a newline and a tab.

Using the “r” (for raw data) prefix on the string, we ensure that all backslashes are escaped:

```
>>> my_path = r'C:\news\today.txt'
```

```
>>> print(my_path)
C:\news\today.txt
```

If you examine the variable directly without printing it, you see that each backslash is escaped with another backslash:

```
>>> my_path
'C:\\news\\today.txt'
```

Note that backslashes will always escape single and double quotation marks, even in a raw string, so you cannot end a raw string with a single backslash:

Bad

```
my_path = r'C:\my\new\'
```

Good

```
my_path = r'C:\my\new\\'
```

Later, when we learn about [regular expressions](#), you'll see how raw strings can come in handy.

Triple Quotes

Triple quotes are used to create multi-line strings. You generally use three double quotes^{[20](#)} as shown in the following example:

Demo 16: strings/Demos/triple_quotes.py

```
print("""-----  
LUMBERJACK SONG  
  
I'm a lumberjack  
And I'm O.K.  
I sleep all night  
And I work all day.  
-----""")
```

Code Explanation

The preceding code will render the following:

```
-----  
LUMBERJACK SONG  
  
I'm a lumberjack  
And I'm O.K.  
I sleep all night  
And I work all day.  
-----
```

Note that quotation marks can be included within triple-quoted strings without being escaped with a backslash.

In some cases when using triple quotes, you may want to break up your code with a newline without having that newline show up in your output. You can escape the actual newline with a backslash as shown in the following demo:

Demo 17: strings/Demos/triple_quotes_newline_escaped.py

```
print("""We're knights of the Round Table, \
we dance whene'er we're able.
We do routines and chorus scenes \
with footwork impeccable,
We dine well here in Camelot, \
we eat ham and jam and Spam a lot.""")
```

Code Explanation

The preceding code will render the following:

```
We're knights of the Round Table, we dance whene'er we're able.
We do routines and chorus scenes with footwork impeccable,
We dine well here in Camelot, we eat ham and jam and Spam a lot.
```

Notice that the backslashes at the end of lines 1, 3, and 5 prevent line breaks in the output.

Spacing in ebooks

Because ebooks do not have a set font size, it is impossible for authors to control spacing and line breaks. So, the code above (and other code samples throughout this book) may wrap in places where it would not actually wrap when programming. We're sorry about that! The best way to see how it will really look is to run these code samples on your computer.

String Indexing

Indexing is the process of finding a specific element within a sequence of elements through the element's position. Remember that strings are basically sequences of characters. We can use indexing to find a specific character within the string.

If we consider the string from left to right, the first character (the left-

most) is at position 0. If we consider the string from right to left, the first character (the right-most) is at position -1. The following table illustrates this for the phrase “MONTY PYTHON”.

	M	O	N	T	Y		P	Y	T	H	O	N
Left to Right:	0	1	2	3	4	5	6	7	8	9	10	11
Right to Left:	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

The following demonstration shows how to find characters by position using indexing.

Demo 18: strings/Demos/string_indexing.py

```
phrase = "Monty Python"

first_letter = phrase[0] # [M]onty Python
print(first_letter)

last_letter = phrase[-1] # Monty Pytho[n]
print(last_letter)

fifth_letter = phrase[4] # Mont[y] Python
print(fifth_letter)

third_letter_from_end = phrase[-3] # Monty Pyt[h]on
print(third_letter_from_end)
```

Code Explanation

The expected output for each `print` statement is shown in square brackets in the comment. Running the file should result in:

```
M
n
y
h
```

Exercise 11: Indexing Strings

10 to 20 minutes.

In this exercise, you will write a program that gets a specific character from a phrase entered by the user.

1. Open [strings/Exercises/indexing.py](#).
2. Modify the `main()` function so that it:

- A. Prompts the user to enter a phrase.
- B. Tells the user what phrase they entered (e.g., Your phrase is 'Hello, world!').
- C. Prompts the user for a number.
- D. Tells the user what character is at that position in the user's phrase (e.g., Character number 4 is o).

3. Here is the program completed by the user:

```
Choose a phrase: Hello, world!  
Your phrase is 'Hello, world!'  
Which character? [Enter number] 4  
Character 4 is o
```

Challenge

As a Python programmer, you understand that the “o” in “Hello” is at position 4, because Python starts counting with 0. However, regular people will think that the character at position 4 is “l” and they will think your program is wrong. Fix your program so that it responds as the user expects. Also, to make it a little prettier, output the character in single quotes.

The program should work like this:

```
Choose a phrase: Hello, world!  
Your phrase is 'Hello, world!'  
Which character? [Enter number] 4  
Character 4 is 'l'
```

Solution: strings/Solutions/indexing.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos = int(input("Which character? [Enter number] "))
    print("Character number", pos, "is", phrase[pos])

main()
```

Challenge Solution: strings/Solutions/indexing_challenge.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos = int(input("Which character? [Enter number] "))-1
    print("Character ", pos+1, " is '", phrase[pos], "'", sep="")

main()
```

Slicing Strings

Often, you will want to get a sequence of characters from a string (i.e., a *substring*). In Python, you do this by getting a slice of the string using the following syntax:

```
substring = orig_string[first_pos:last_pos]
```

This returns a slice that starts with the character at `first_pos` and includes all the characters up to *but not including* the character at `last_pos`.

If `first_pos` is left out, then it is assumed to be 0. So `'hello'[:3]` would return `"hel"`.

If `last_pos` is left out, then it is assumed to be the length of the string, or in other words, one more than the last position of the string. So `'hello'[3:]` would return `"lo"`.

The following demonstration shows how to get substrings using slicing.

Demo 19: strings/Demos/string_slicing.py

```
phrase = "Monty Python"

first_5_letters = phrase[0:5] # [Monty] Python
print(first_5_letters)

letters_2_thru_4 = phrase[1:4] # M[ont]y Python
print(letters_2_thru_4)

letter_5_to_end = phrase[4:] # Mont[y Python]
print(letter_5_to_end)

last_3_letters = phrase[-3:] # Monty Pyt[hon]
print(last_3_letters)

first_3_letters = phrase[:3] # [Mon]ty Python
print(first_3_letters)

three_letters_before_last = phrase[-4:-1] # Monty Py[tho]n
print(three_letters_before_last)

copy_of_string = phrase[:] # [Monty Python]
print(copy_of_string)
```

Code Explanation

The expected output for each `print` statement is shown in square brackets in the comment. Running the file should result in:

```
Monty
ont
y Python
hon
Mon
tho
Monty Python
```

Exercise 12: Slicing Strings

10 to 20 minutes.

In this exercise, you will write a program that gets a substring (or slice) from a phrase entered by the user.

1. Open `strings/Exercises/slicing.py`.
2. Modify the `main()` function so that it:
 - A. Prompts the user to enter a phrase.
 - B. Tells the user what phrase they entered (e.g., Your phrase is 'Hello, world!').
 - C. Prompts the user for a start number.
 - D. Prompts the user for an end number.
 - E. Tells the user the substring (within single quotes) that starts with the start number and ends with the end number.
3. Here is the output of the program:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Character to start with? [Enter number] 4
Character to end with? [Enter number] 9
Your substring is 'o, wor'
```

Challenge

As with the last exercise, make your program respond as users would expect.

The new program should work like this:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Character to start with? [Enter number] 4
Character to end with? [Enter number] 9
```

Your substring is 'lo, wo'

Solution: strings/Solutions/slicing.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos1 = int(input("Character to start with? [Enter number] ")
    pos2 = int(input("Character to end with? [Enter number] "))
    print("Your substring is '", phrase[pos1:pos2], "'", sep="")

main()
```

Challenge Solution: strings/Solutions/slicing_challenge.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos1 = int(input("Character to start with? [Enter number] ")
    pos2 = int(input("Character to end with? [Enter number] "))
    print("Your substring is '", phrase[pos1:pos2], "'", sep="")

main()
```

Concatenation and Repetition

Concatenation

Concatenation is a fancy word for stringing strings together. In Python, concatenation is done with the + operator. It is often used to combine variables with literals as in the following example:

Demo 20: strings/Demos/concatenation.py

```
user_name = input("What is your name? ")
greeting = "Hello, " + user_name + "!"
print(greeting)
```

Code Explanation

The preceding code will render the following:

```
What is your name? Nat
Hello, Nat!
```

Repetition

Repetition is the process of repeating a string some number of times. In Python, repetition is done with the `*` operator.

Demo 21: strings/Demos/repetition.py

```
one_knight_says = "nee"  
many_knights_say = one_knight_says * 20  
print(many_knights_say)
```

Code Explanation

The preceding code will render the following:

```
neeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneene
```

Exercise 13: Repetition

5 to 10 minutes.

Remember our `insert_separator()` function from the “Hello, You!” programs. It looked like this:

```
def insert_separator(s==""):  
    print(s, s, s, sep="")
```

Using repetition, we can improve `insert_separator()` so that the number of times the separating character shows up is passed into the function.

1. Open `strings/Exercises/hello_you.py` in your editor.
2. Modify the `insert_separator()` function so that the number of times the separating character shows up is passed in as a parameter. It should default to show up 30 times.
3. Modify the calls to `insert_separator()` so that they pass in an argument to the new parameter.

Solution: strings/Solutions/hello_you.py

```
def say_hello(name):
    print('Hello, ', name, '!', sep='')

def insert_separator(s="=", repeat=30):
    print(s * repeat)

def recite_poem():
    print("How about a Monty Python poem?")
    insert_separator("-", 20)
    print("Much to his Mum and Dad's dismay")
    print("Horace ate himself one day.")
    print("He didn't stop to say his grace,")
    print("He just sat down and ate his face.")

def say_goodbye(name):
    print('Goodbye, ', name, '!', sep='')

def main():
    your_name = input('What is your name? ')
    insert_separator("-", 20)
    say_hello(your_name)
    insert_separator()
    recite_poem()
    insert_separator()
    say_goodbye(your_name)

main()
```

Combining Concatenation and Repetition

Concatenation and repetition can be combined. Repetition takes precedence, meaning it occurs first. Consider the following:

```
>>> 'a' + 'b' * 3 + 'c'
'abbbc'
```

Notice the output is “abbbc”. In other words, “b” will be repeated three

times before it is concatenated with “a” and “c”.

We can force the concatenation to take place first by using parentheses:

```
>>> ('a' + 'b') * 3 + 'c'  
'abababc'
```

Notice the output is “abababc”. In other words, “a” will be concatenated with “b”, then “ab” will be repeated three times, and finally “ababab” will be concatenated with “c”.

The following demo shows an example of combining concatenation with repetition:

Demo 22: strings/Demos/combining_concatenation_and_repetition.py

```
flower = input("What is your favorite flower? ")
reply = "A " + flower + (" is a " + flower) * 2 + "."
print(reply)
```

Code Explanation

The preceding code will render the following:

```
What is your favorite flower? dandelion
A dandelion is a dandelion is a dandelion.
```

Python Strings are Immutable

Python strings are immutable, meaning that they cannot be changed. However, it is easy to make a copy of a string and then assign the copy to the same variable as the original string.

To illustrate, we will use Python's built-in `id()` function, which returns the identity of an object:

```
>>> name = 'Nat'
>>> id(name)
1670060967728
>>> name += 'haniel'
>>> name
'Nathaniel'
>>> id(name)
1670060968240
```

Notice that the `id` of `name` changes when we modify the string in the variable.

Because strings are immutable, methods that operate on strings (i.e., string methods) cannot modify the string in place. Instead, they return a value. Sometimes that value is a modified version of the string, but it

is important to understand that the original string is unchanged. Consider, for example, the `upper()` method, which returns a string in all uppercase letters:

```
>>> name = 'Nat'
>>> name.upper() # Returns uppercase copy of 'Nat'
'NAT'
>>> name # Original variable is unchanged
'Nat'
```

If you want to change the original variable, you must assign the returned value back to the variable:

```
>>> name = 'Nat'
>>> name = name.upper()
>>> name
'NAT'
```

Common String Methods

String Methods that Return a Copy of the String

Methods that Change Case

- `str.capitalize()` – Returns a string with only the first letter capitalized.
- `str.lower()` – Returns an all lowercase string.
- `str.upper()` – Returns an all uppercase string.
- `str.title()` – Returns a string with each word beginning with a capital letter followed by all lowercase letters.
- `str.swapcase()` – Returns a string with the case of each letter swapped.

```
>>> 'hELLo'.capitalize()
'Hello'
>>> 'hELLo'.lower()
'hello'
```

```
'hello'  
>>> 'hELLO'.upper()  
'HELLO'  
>>> 'hello world'.title()  
'Hello World'  
>>> 'hELLO'.swapcase()  
'Hello'
```

`str.replace(old, new[, count])`

A string with `old` replaced by `new` `count` times.

```
>>> 'mommy'.replace('m', 'b')  
'bobby'  
>>> 'mommy'.replace('m', 'b', 2)  
'bobmy'
```

Square Brackets in Code Notation

As we mentioned in the [Math lesson](#), square brackets in code notation indicate that the contained portion is optional. To illustrate, consider the `str.replace()` method:

```
str.replace(old, new[, count])
```

This means that the `count` parameter, which indicates the maximum number of replacements to be made, is optional.

The syntax allows for nesting optional parameters within optional parameters. For example:

```
str.find(sub[, start[, end]])
```

This syntax indicates that `end` cannot be specified unless `start` is also specified. The outside square brackets in `[, start[, end]]` indicate that the whole section is optional. The inside square brackets indicate that `end` is optional even after `start` is specified. If it were written as `[start, end]`, it would indicate that `start` and `end`

are optional, but if one is included, the other must be included as well.

Methods that Strip Characters

- `str.strip([chars])` – Returns a string with leading and trailing chars removed.
- `str.lstrip([chars])` – Returns a string with leading chars removed.
- `str.rstrip([chars])` – Returns a string with trailing chars removed.

`chars` defaults to whitespace.

```
>>> ' hello '.strip()
'hello'
>>> 'hello'.lstrip('h')
'ello'
>>> 'hello'.rstrip('o')
'hell'
```

String Methods that Return a Boolean

- `str.isalnum()` – Returns `True` if all characters are letters or numbers.
- `str.isalpha()` – Returns `True` if all characters are letters.
- `str.islower()` – Returns `True` if string is all lowercase.
- `str.isupper()` – Returns `True` if string is all uppercase.
- `str.istitle()` – Returns `True` if string is title case.

```
>>> 'Hello World!'.isalnum()
False
>>> 'Hello World!'.isalpha()
False
```

```
>>> 'hello'.islower()
True
>>> 'HELLO'.isupper()
True
>>> 'Hello World!'.istitle()
True
```

`str.isspace()`

True if string is made up of only whitespace.

```
>>> ' '.isspace()
True
>>> ' hi '.isspace()
False
```

`str.isdigit()`, `str.isdecimal()`, and `str.isnumeric()`

The `str.isdigit()`, `str.isdecimal()`, and `str.isnumeric()` all check to see if a string has only numeric characters.

1. All three will return `True` if the string contains only Arabic digits (i.e., 0 through 9):

```
'42'.isdigit() # True
'42'.isdecimal() # True
'42'.isnumeric() # True
```

2. All three will return `False` if any character is non-numeric:

```
'4.2'.isdigit() # False
'4.2'.isdecimal() # False
'4.2'.isnumeric() # False
```

Beyond that, the difference is mostly academic for most Python developers:

1. `'2³'.isnumeric()` and `'2³'.isdigit()` return `True`, but `'2³'.isdecimal()` returns `False`.
2. `'¼'.isnumeric()` returns `True`, but `'¼'.isdigit()` and `'¼'.isdecimal()` return `False`.

So, which should you use? It doesn't make much difference really, but `isdigit()` is the most popular, perhaps because the name most closely matches the intention of the function.

If you're really curious about it, run [strings/Demos/numbers.py](#), which will create a [numbers.txt](#) file in the same folder showing tabular results, like this (but with much more data):

Char	isdigit	isdecimal	isnumeric
0	TRUE	TRUE	TRUE
1	TRUE	TRUE	TRUE
2	TRUE	TRUE	TRUE
3	TRUE	TRUE	TRUE
²	TRUE	FALSE	TRUE
¼	FALSE	FALSE	TRUE

`str.startswith()` and `str.endswith()`

- `str.startswith(prefix[, start[, end]])` – Returns `True` if string **starts with** prefix.
- `str.endswith(prefix[, start[, end]])` – Returns `True` if string **ends with** suffix.

Both of these methods start looking at `start` index and end looking at `end` index if `start` and `end` are specified.

```
>>> 'hello'.startswith('h')
True
>>> 'hello'.endswith('o')
True
```

is... Methods

All the `is...` methods shown in the table above return `False` for *empty* strings, meaning strings with zero length.

String Methods that Return a Number

String Methods that Return a Position (Index) of a Substring

- `str.find(sub[, start[, end]])` – Returns the lowest index where `sub` is found. Returns `-1` if `sub` isn't found.
- `str.rfind(sub[, start[, end]])` – Returns the highest index where `sub` is found. Returns `-1` if `sub` isn't found.
- `str.index(sub[, start[, end]])` – Same as `find()`, but errors when `sub` is not found.
- `str.rindex(sub[, start[, end]])` – Same as `rfind()`, but errors when `sub` is not found.

All of these methods start looking at `start` index and end looking at `end` index if `start` and `end` are specified.

```
>>> 'Hello World!'.find('l')
2
>>> 'Hello World!'.rfind('l')
9
>>> 'Hello World!'.index('l')
2
>>> 'Hello World!'.rindex('l')
9
```

`str.count(sub[, start[, end]])`

Returns the number of times `sub` is found. Start looking at `start` index and end looking at `end` index.

```
>>>"Hello World!".count('l')
3
```

String Formatting

Python includes powerful options for formatting strings.

The `format()` Method

One common way to format strings is to use the `format()` method combined with the [Format Specification Mini-Language](#)²¹.

Let's start with a simple example and then we'll explain the mini-language in detail:

```
>>> '{0} is an {1} movie!'.format('Monty Python', 'awesome')
'Monty Python is an awesome movie!'
```

The curly braces are used to indicate a replacement field, which takes position arguments specified by index (as in the preceding example) or by name (as in the following example):

```
>>> '{movie} is an {adjective} movie!'.format(movie='Monty Python',
...                                           adjective='awesome')
'Monty Python is an awesome movie!'
```

The field names (position arguments) can be omitted:

```
'{} is an {} movie!'.format('Monty Python', 'awesome')
```

When the field names are omitted, the first replacement field is at index 0, the second at index 1, and so on.

These examples really just show another form of concatenation and could be rewritten like this:

```
'Monty Python' + ' is an ' + awesome + ' movie!'
```

```
# or

movie = 'Monty Python'
adjective = 'awesome'
movie + ' is an ' + adjective + ' movie!'
```

When doing a lot of concatenation, using the `format()` method can be cleaner. However, as the name implies, the `format()` method does more than just concatenation; it also can be used to *format* the replacement strings. It is mostly used for formatting numbers.

Format Specification

The format specification is separated from the field name or position by a colon (:), like this:

```
{field_name:format_spec}
```

Because the field name is often left out, it is commonly written like this:

```
{:format_spec}
```

The format specification^{[22](#)} is:

```
[[fill]align][sign][width][,][.precision][type]
```

That looks a little scary, so let's break it down from right to left.

Type

```
[[fill]align][sign][width][,][.precision][type]
```

Type is specified by a one-letter specifier, like this:

```
'{:s} is an {:s} movie!'.format('Monty Python', 'awesome')
```

The `s` indicates that the replacement field should be formatted as a string. There are many different types, but, unless you are a mathematician or scientist²³, the most common types you'll be working with are strings, integers, and floats.

The default formatting for strings and integers are string format (`s`) and decimal integer (`d`), which are generally what you will want, so you can leave the one-letter type specifier off. Consider the following:

Demo 23: strings/Demos/formatting_types.py

Full formatting strings.

```
sentence = 'On a scale of {0:d} to {1:d}, I give {2:s} a {3:d}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)
```

Simplify by removing field names (indexes).

```
sentence = 'On a scale of {:d} to {:d}, I give {:s} a {:d}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)
```

Further simplify by removing default type specifiers.

```
sentence = 'On a scale of {:} to {:}, I give {:} a {:}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)
```

And with the field name and type specifier gone, we can # remove the colon separator.

```
sentence = 'On a scale of {} to {}, I give {} a {}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)
```

Code Explanation

The final line of code has the advantage of being brief, but the disadvantage of being obscure. As a rule, we prefer clarity over brevity. We can make it clearer using field names:

```
>>> 'On a scale of {low} to {high}, {movie} is a {rating}.'.format(
    1, 5, 'Monty Python', 6)
```

Floats

You will typically format floats as fixed point numbers using `f` as the one-letter specifier, which has a default precision of 6. If neither type nor precision is specified, floats will be as precise as they need to be

to accurately represent the value.

Fixed point type specified:

```
>>> import math
>>> 'pi equals {:.f}'.format(math.pi)
'pi equals 3.141593'
```

No type specified:

```
>>> import math
>>> 'pi equals {}'.format(math.pi)
'pi equals 3.141592653589793'
```

Another formatting option for floats is percentage (%). We will cover that shortly.

Precision

```
[[fill]align][sign][width][,][.precision][type]
```

The precision is specified before the type and is preceded by a decimal point, like this:

```
>>> import math
>>> 'pi equals {:.2f}'.format(math.pi)
'pi equals 3.14'
```

Separating the Thousands

```
[[fill]align][sign][width][,][.precision][type]
```

Insert a comma before the precision value to separate the thousands with commas, like this:

```
>>> '{:,.2f}'.format(1000000)
'1,000,000.00'
```

Width

```
[[fill]align][sign][width][,][.precision][type]
```

The width is an integer indicating the minimum number of characters of the resulting string. If the passed-in string has fewer characters than the specified width, *padding* will be added. By default, padding is added *after* strings and *before* numbers, so that strings are aligned to the left and numbers are aligned to the right.

Here are some examples:

```
>>> '{:5}'.format('abc')
'abc  '
>>> '{:5}'.format(123)
'  123'
>>> '{:5.2f}'.format(123)
'123.00'
```

In all three cases, the width of the formatted string is set to 5. Notice the padding on the first two examples: after the string and before the number.

In the final example, we format the number 123, but the format type has been specified as fixed point (f) with a precision of 2. So, the resulting string ('123.00') is six characters long – longer than the specified width, so it just returns the full string without padding.

Sign

```
[[fill]align][sign][width][,][.precision][type]
```

By default, negative numbers are preceded by a negative sign, but positive numbers are not preceded by a positive sign. To force the sign to show up, add a plus sign (+) before the precision, like this:

```
>>> 'pi equals {:.2f}'.format(math.pi)
```

```
'pi equals +3.14'
```

Alignment

```
[[fill]align][sign][width][,][.precision][type]
```

You can change the default alignment by preceding the width (and sign if there is one) with one of the following options:

Alignment

Options	Meaning
<	Left aligned (default for strings).
>	Right aligned (default for numbers).
=	Padding added between sign and digits. Only valid for numbers.
^	Centered.

Some examples:

```
>>> '{:>5}'.format('abc')
'   abc'
```

```
>>> '{:<5}'.format(123)
'123   '
```

```
>>> '{:^5}'.format(123)
'  123  '
```

```
>>> '{:=+5}'.format(123)
'+ 123'
```

Fill

```
[[fill]align][sign][width][,][.precision][type]
```

By default, spaces are used for padding, but this can be changed by

inserting a fill character before the alignment option, like this (note the period after the colon):

```
>>> '{: ^10.2f}'.format(math.pi)
'...3.14...'
```

And now with a dash:

```
>>> '{: ^10.2f}'.format(math.pi)
'---3.14---'
```

Percentage Type

As mentioned earlier, another option for type is percentage (%):

```
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> grade
0.72
>>> '{:.2f}'.format(grade)
'0.72'
>>> '{:.2%}'.format(grade)
'72.00%'
>>> '{:.0%}'.format(grade)
'72%'
```

Long Lines of Code

The [Python Style Guide](#)²⁴ suggests that lines of code should be limited to 79 characters. This can be difficult as each line of code is considered a new statement; however, it can usually be accomplished through some combination of:

1. Breaking method arguments across multiple lines.
2. Concatenation.

3. Triple-quoted multi-line strings.

All three methods are shown in the following file:

Demo 24: strings/Demos/long_code_lines.py

EXAMPLE 1: Breaking method arguments across multiple lines

```
phrase = ("On a scale of {} to {}, I give {} a {}".format(1, 5,
                                                         "Monty",
                                                         "Monty"))
print(phrase)
```

```
location = "ponds"
items = "swords"
beings = "masses"
adjective = "farcical"
```

EXAMPLE 2: Concatenation

```
quote = ("Listen, strange women lyin' in {} " +
         "distributin' {} is no basis for a system of " +
         "government. Supreme executive power derives from " +
         "a mandate from the {}, not from some {} " +
         "aquatic ceremony.").format(location, items,
                                     beings, adjective)

print(quote)
```

EXAMPLE 3: Triple quotes

```
quote = """Listen, strange women lyin' in {} \
distributin' {} is no basis for a system of \
government. Supreme executive power derives from \
a mandate from the {}, not from some {} \
aquatic ceremony.""".format(location, items,
                             beings, adjective)

print(quote)
```

Code Explanation

Remember that the backwards slashes at the end of each line escape the newline character.

Also, notice that the arguments passed to `format()` are broken across

lines and vertically aligned to make it clear that they are related.

Run the file to see the resulting strings.

Exercise 14: Playing with Formatting

10 to 20 minutes.

In this exercise, you will practice formatting strings. Here are two options for practicing:

Option 1

1. Run `'{}'.format('')` at the Python shell.
2. Try formatting different values in different ways. For example, try running:

```
>>> '{:.0%}'.format(.87)
```

Option 2

1. From [strings/Demos](#) run `python formatter.py`
2. Try different format specifications with different numbers, like this:

```
Format to try: {:f}
Entry to format: math.pi
Result: 3.141593
Enter for another or 'q' to quit.
Format to try: {}
Entry to format: math.pi
Result: 3.141592653589793
Enter for another or 'q' to quit.
Format to try: {:.2f}
Entry to format: math.pi
Result: 3.14
Enter for another or 'q' to quit.
Format to try: {:.2f}
```

```
Entry to format: 1000000
Result: 1000000.00
Enter for another or 'q' to quit.
Format to try: {:, .2f}
Entry to format: 1000000
Result: 1,000,000.00
Enter for another or 'q' to quit.
Format to try: {:>20}
Entry to format: 'abc'
Result: abc
```

Formatted String Literals (f-strings)

Formatted string literals or *f-strings*^{[25](#)} use a syntax that implicitly embeds the `format()` function within the string itself. The coding tends to be less verbose.

The following demonstration compares string concatenation and string formatting with the f-string syntax.

Demo 25: strings/Demos/f_strings.py

```
import math
user_name = input("What is your name? ")

# Concatenation:
greeting = "Hello, " + user_name + "!"

# The format() method:
greeting = "Hello, {}".format(user_name)

# f-string:
greeting = f"Hello, {user_name}!"
print(greeting)

# format specification is also available:
pi_statement = f"pi is {math.pi:.4f}"
print(pi_statement)
```

Code Explanation

The curly braces within the f-string contain the variable name and optionally a format specification. The string literal is prepended with an f.

Everything you learned earlier about formatting can be applied to the f-string because the same formatting function is called. Practice with f-strings by running the following lines of code:

```
>>> import math
>>> movie = 'Monty Python'
>>> adjective = 'awesome'
>>> f'{movie} is an {adjective} movie!'
'Monty Python is an awesome movie!'
>>> low = 1
>>> high = 5
>>> rating = 6
```

```

>>> f'On a scale of {low} to {high}, {movie} is a {rating}.'
'On a scale of 1 to 5, Monty Python is a 6.'
>>> f'pi equals {math.pi:f}'
'pi equals 3.141593'
>>> f'pi equals {math.pi}'
'pi equals 3.141592653589793'
>>> f'pi equals {math.pi:.2f}'
'pi equals 3.14'
>>> f'{1000000:,.2f}'
'1,000,000.00'
>>> f"{'abc':20}"
'abc'
>>> f'{123:20}'
'123'
>>> f'{123:5.2f}'
'123.00'
>>> f'pi equals {math.pi:+.2f}'
'pi equals +3.14'
>>> f"{'abc':>20}"
'abc'
>>> f'{123:<20}'
'123'
>>> f'{123:=+20}'
'+123'
>>> f'pi equals {math.pi:.^10.2f}'
'pi equals ...3.14...'
>>> f'pi equals {math.pi:-^10.2f}'
'pi equals ---3.14---'
>>>
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> f'{grade:.2f}'
'0.72'
>>> f'{grade:.2%}'
'72.00%'
>>> f'{grade:.2f}'
'0.72'

```

Built-in String Functions

str(object)

Converts object to a string.

```
>>> str('foo') # 'foo' is already a string
'foo'
>>> str(999)
'999'
>>> import math
>>> str(math.pi)
'3.141592653589793'
```

len(string)

Returns the number of characters in the string. [26](#)

```
>>> len('foo')
3
```

min() and max() [27](#)

The `min(args)` function returns the smallest value of the passed-in arguments.

The `max(args)` function returns the largest value of the passed-in arguments.

```
>>> min('w', 'e', 'b')
'b'
>>> min('a', 'B', 'c')
'B'
>>> max('w', 'e', 'b')
'w'
>>> max('a', 'B', 'c')
'c'
```

Note that all uppercase letters come before lowercase letters (e.g., `min('Z', 'a')` returns `'Z'`).

min() and max() with Iterables

The `min()` and `max()` functions can also take an *iterable* containing values to compare. We will cover this in the [Iterables lesson](#).

Exercise 15: Outputting Tab-delimited Text

25 to 40 minutes.

In this exercise, you will write a program that repeatedly prompts the user for a Company name, Revenue, and Expenses and then outputs all the information as tab-delimited text. Here is the program after it has run:

```
Company: Pepperpots
Revenue: 1200000
Expenses: 999002
Again? Press ENTER to add a row or Q to quit.
Company: Ni Knights
Revenue: 19
Expenses: 24
Again? Press ENTER to add a row or Q to quit.
Company: Round Knights
Revenue: 777383
Expenses: 777382
Again? Press ENTER to add a row or Q to quit. q
```

Company	Revenue	Expenses	Profit
Pepperpots	\$1,200,000.00	\$999,002.00	\$200,998.00
Ni Knights	\$ 19.00	\$ 24.00	\$ -5.00
Round Knights	\$777,383.00	\$777,382.00	\$ 1.00

In this exercise, you can use the `format()` method or f-strings or any combination of the two.

1. Open `strings/Exercises/tab_delimited_text.py`.

2. Modify the `addheaders()` function so that it creates a header row and appends it to `_output`. The four headers should each take up 10 spaces, be aligned to the center, and be separated by tabs, like this:

```
' Company \t Revenue \t Expenses \t Profit \n'
```

Don't just copy that string. Use the `format()` method.

3. Modify the `addrows()` function so that it adds a row to `_output` by prompting the user for values for company, revenue, and expenses, and then calculating profit.
 - A. All "columns" should be 10 spaces wide.
 - B. The company name should be a left-aligned string.
 - C. The other three columns should be formatted in U.S. dollars (e.g., \$1,200,000.00) and right-aligned.
4. Save and run the file. Try entering data for at least three companies.

Exercise Code: strings/Exercises/tab_delimited_text.py

```
_output = ""

def addheaders():
    # Write your code here
    pass

def addrow():
    # Write your code here

    # The rest of the function prompts the user to add another row
    # or quit. On quitting, it prints _output. Leave it as is.

    again = input("Again? Press ENTER to add a row or Q to quit. ")
    if again.lower() != "q":
        addrow()
    else:
        print(_output)

def main():
    # Call addheaders() and addrow()
    addheaders()
    addrow()

main()
```

Solution: strings/Solutions/tab_delimited_text.py

```
_output = ""

def add_headers():
    global _output
    c_header = "{:^10}".format("Company")
    r_header = "{:^10}".format("Revenue")
    e_header = "{:^10}".format("Expenses")
    p_header = "{:^10}".format("Profit")
    _output += "{}\t{}\t{}\t{}\n".format(c_header, r_header,
                                         e_header, p_header)

def add_row():
    global _output

    c = input("Company: ")
    r = float(input("Revenue: "))
    e = float(input("Expenses: "))
    p = r - e # profit

    c_str = "{:<10}".format(c)
    r_str = "${:>10, .2f}".format(r)
    e_str = "${:>10, .2f}".format(e)
    p_str = "${:>10, .2f}".format(p)

    new_row = "{}\t{}\t{}\t{}\n".format(c_str, r_str, e_str, p_str)

    _output += new_row
-----Lines Omitted-----
```

Solution: strings/Solutions/tab_delimited_text_f_string.py

```
_output = ""

def add_headers():
    global _output
    c_header = f"{'Company':^10}"
    r_header = f"{'Revenue':^10}"
    e_header = f"{'Expenses':^10}"
    p_header = f"{'Profit':^10}"
    _output += f"{c_header}\t{r_header}\t{e_header}\t{p_header}\n"

def add_row():
    global _output

    c = input("Company: ")
    r = float(input("Revenue: "))
    e = float(input("Expenses: "))
    p = r - e # profit

    c_str = f"{c:<10}"
    r_str = f"${r:>10, .2f}"
    e_str = f"${e:>10, .2f}"
    p_str = f"${p:>10, .2f}"

    new_row = f"{c_str}\t{r_str}\t{e_str}\t{p_str}\n"

    _output += new_row
-----Lines Omitted-----
```

Conclusion

In this lesson, you have learned to manipulate and format strings.

LESSON 6

Iterables: Sequences, Dictionaries, and Sets

Topics Covered

- Types of iterables available in Python.
- Lists.
- Tuples.
- Ranges.
- Dictionaries.
- Sets.
- The `*args` and `**kwargs` parameters.

I did not like this iteration of one idea--this strange recurrence of one image, and I grew nervous as bedtime approached and the hour of the vision drew near.

– Jane Eyre, Charlotte Bronte

Iterables are objects that can return their members one at a time. The iterables we will cover in this lesson are lists, tuples, ranges, dictionaries, and sets.

Definitions

Here are some quick definitions to provide an overview of the different types of objects we will be covering in this lesson. Don't worry if the meanings aren't entirely clear now. They will be when you finish the lesson.

1. *Sequences* are iterables that can return members based on their position within the iterable. Examples of sequences are strings, lists, tuples, and ranges.

2. *Lists* are *mutable* (changeable) sequences similar to arrays in other programming languages.
3. *Tuples* are *immutable* sequences.
4. *Ranges* are *immutable* sequences of numbers often used in *for loops*.
5. *Dictionaries* are mappings that use arbitrary keys to map to values. Dictionaries are like associative arrays in other programming languages.
6. *Sets* are *mutable* unordered collections of distinct *immutable* objects. So, while the set itself can be modified, it cannot be populated with objects that can be modified.

Sequences

Sequences are iterables that can return members based on their position within the iterable. You have already learned about one type of sequence: *strings*. Remember [string indexing](#):

```
>>> 'Hello, world!'[1]
'e'
```

That's one way of getting at one member of a sequence.

The sequences we cover in this lesson are:

1. Lists
2. Tuples
3. Ranges

Lists

Python's lists are similar to arrays in other languages. Lists are created using square brackets, like this:

```
colors = ["red", "blue", "green", "orange"]
```

List Methods

Some of the most common list methods are shown in the following list:

- `mylist.append(x)` – Appends `x` to `mylist`.
- `mylist.remove(x)` – Removes first element with value of `x` from `mylist`. Errors if no such element is found.
- `mylist.insert(i, x)` – Inserts `x` at position `i`.
- `mylist.count(x)` – Returns the number of times that `x` appears in `mylist`.
- `mylist.index(x)` – Returns the index position of the first element in `mylist` whose value is `x` or a `ValueError` if no such element exists.
- `mylist.sort()` – Sorts `mylist`.
- `mylist.reverse()` – Reverses the order of `mylist`.
- `mylist.pop(n)` – Removes and returns the element at position `n` in `mylist`. If `n` is not passed in, the last element in the list is popped (removed and returned).
- `mylist.clear()` – Removes all elements from `mylist`.
- `mylist.copy()` – Returns a copy of `mylist`.
- `mylist.extend(anotherlist)` – Appends `anotherlist` onto `mylist`.

The following code illustrates how these methods are used. Try this out at the Python shell:

```
>>> colors = ["red", "blue", "green", "orange"]
>>> colors
['red', 'blue', 'green', 'orange']
>>> colors.append("purple") # Append purple to colors
>>> colors
['red', 'blue', 'green', 'orange', 'purple']
>>> colors.remove("green") # Remove green from colors
>>> colors
['red', 'blue', 'orange', 'purple']
>>> colors.insert(2, "yellow") # Insert yellow in position 2
```

```

>>> colors
['red', 'blue', 'yellow', 'orange', 'purple']
>>> colors.index("orange") # Get position of orange
3
>>> colors.sort() # Sort colors in place
>>> colors
['blue', 'orange', 'purple', 'red', 'yellow']
>>> colors.reverse() # Reverse order of colors
>>> colors
['yellow', 'red', 'purple', 'orange', 'blue']
>>> colors.pop() # Remove and return last element
'blue'
>>> colors.pop(1) # Remove and return element at position 1
'red'
>>> colors # Notice blue and red have been removed
['yellow', 'purple', 'orange']
>>> colors_copy = colors.copy() # Create a copy of colors
>>> colors_copy
['yellow', 'purple', 'orange']
>>> colors.extend(colors_copy) # Append colors_copy to colors
>>> colors
['yellow', 'purple', 'orange', 'yellow', 'purple', 'orange']
>>> colors_copy.clear() # Delete all elements from colors_copy
>>> colors_copy # Notice colors_copy is now empty
[]
>>> del colors_copy # Delete colors_copy
>>> colors_copy # It's gone
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'colors_copy' is not defined

```

Copying a List

Note again how we made a copy of the `colors` list:

```

>>> colors = ["red", "blue", "green", "orange"]
>>> colors_copy = colors.copy()
>>> colors_copy

```

```
['red', 'blue', 'green', 'orange']
>>> colors_copy.sort()
>>> colors_copy
['blue', 'green', 'orange', 'red']
>>> colors
['red', 'blue', 'green', 'orange'] # colors remains unsorted
```

You can see that sorting `colors_copy` has no effect on the `colors` list. They are two distinct objects. Compare that to the following, which continues from the code above:

```
>>> colors_copy2 = colors
>>> colors_copy2
['red', 'blue', 'green', 'orange']
>>> colors_copy2.sort()
>>> colors_copy2
['blue', 'green', 'orange', 'red']
>>> colors
['blue', 'green', 'orange', 'red'] # Wait! colors is sorted too!
>>> id(colors)
2147162345152
>>> id(colors_copy)
2147163702400 # different id than colors
>>> id(colors_copy2)
2147162345152 # the same id as colors
```

Do you see what's going on here? When you assign one variable to another, instead of making a copy, it just creates a pointer to that object. If you modify one object, it affects both. The `copy()` method creates a brand new identical object, so modifying the new object has no effect on the original.

Deleting List Elements

The `del` statement can be used to delete elements or slices of elements from a list, like this:

Demo 26: iterables/Demos/del_list.py

```
colors = ['red', 'blue', 'green', 'orange', 'black']
```

```
del colors[0] # deletes first element
```

```
print(colors) # ['blue', 'green', 'orange', 'black']
```

```
del colors[1:3] # deletes 2nd and 3rd elements
```

```
print(colors) # ['blue', 'black']
```

Sequences and Random

In the **Math** lesson, when we learned about the `random` [module](#), we mentioned two methods that we were not yet ready to explore. Now, we are:

`random.choice(seq)`

Returns random element in seq.

```
>>> import random
>>> colors = ["red", "blue", "green", "orange"]
>>> random.choice(colors)
'orange'
>>> random.choice(colors)
'green'
```

`random.shuffle(seq)`

Shuffles seq in place.

```
>>> import random
>>> colors = ["red", "blue", "green", "orange"]
>>> random.shuffle(colors)
>>> colors
['green', 'red', 'blue', 'orange']
```

Exercise 16: Remove and Return Random Element

In this exercise, you will write a `remove_random()` function that removes a random element from a list and returns it.

1. Open `iterables/Exercises/remove_random.py` in your editor.
2. Write the code for the `remove_random()` function so that it removes and returns a random element from the passed-in list: `the_list`.
3. Modify the `main()` function so that it uses `remove_random()` to remove a random element from the `colors` list and then prints something like the following:

```
The removed color was green.  
The remaining colors are ['red', 'blue', 'orange'].
```

Exercise Code: iterables/Exercises/remove_random.py

```
import random

def remove_random(the_list):
    pass # replace this with your code

def main():
    colors = ['red', 'blue', 'green', 'orange']
    # Your code here

main()
```

Solution: iterables/Solutions/remove_random.py

```
import random

def remove_random(the_list):
    x = random.choice(the_list)
    the_list.remove(x)
    return x

def main():
    colors = ['red', 'blue', 'green', 'orange']
    removed_color = remove_random(colors)
    print(f'The removed color was {removed_color}.')
    print(f'The remaining colors are {colors}.')

main()
```

Tuples

Tuples are like lists, but they are *immutable*: once created, they cannot be changed.

Get ready for a lie: Tuples are created using parentheses, like this:

```
MAGENTA = (255, 0, 255)
```

Wait, what??! Why are you lying to me?

OK, the truth is that tuples are created *with commas* AND don't *require* parentheses. You *can* create a tuple like this:

```
MAGENTA = 255, 0, 255 # Avoid this
```

But just because you *can* doesn't mean you *should*. It's a better idea to get used to including the parentheses, because sometimes you do need them. To illustrate, take a look at the following code:

Demo 27: iterables/Demos/tuples.py

```
def show_type(obj):
    print(type(obj))

# tuple created w/o parens (works but bad practice)
MAGENTA = 255, 0, 255
show_type(MAGENTA)

# When passing a tuple to a function, you need parens:
show_type((255, 0, 255))

# Passing the tuple w/o parens to a function will error
show_type(255, 0, 255)
```

Code Explanation

The preceding code will render the following:

```
<class 'tuple'>
<class 'tuple'>
Traceback (most recent call last):
  File "c:/Webucator/Python/iterables/Demos/tuples.py", line 12,
    show_type( 255, 0, 255 )
TypeError: show_type() takes 1 positional argument but 3 were gi
```

1. On line 5, the `MAGENTA` tuple is created without using parentheses.
2. By passing `MAGENTA` to the `show_type()` function, we see that `MAGENTA` is indeed a tuple.
3. On line 9, the tuple `(255, 0, 255)` (constructed with parentheses) is passed to the `show_type()` function. This works fine.
4. On line 12, the tuple (*well, not really*) `255, 0, 255` (constructed without parentheses) is passed to the `show_type()` function. In this case, Python passes the values to the `show_type()` function as three separate arguments. As the function only expects one argument, this results in an error:

```
TypeError: show_type() takes 1 positional argument but 3 were
```

The takeaway here is: *Always use parentheses when creating tuples.*

Remember Constants

The `MAGENTA` variable above makes a good [constant](#). The values represent the amounts of red, green, and blue in the color magenta.

When to Use a Tuple

While lists are used for holding collections of like data, tuples are meant for holding *heterogeneous collections of data*. In tuples, the position of the element is meaningful. To illustrate, let's consider our `MAGENTA` constant again:

```
MAGENTA = (255, 0, 255)
```

The values in the tuple correspond to RGB (red, green, blue) color values. Magenta is created by mixing full red (255) with full blue (255) and no green (0).

Other use cases for tuples include:

1. X-Y Coordinates (e.g, (55, -23))
2. Latitude-Longitude Coordinates (e.g., (43.0298, -76.0044))
3. Geometric Shapes (notice that these are tuples of tuples):

```
line = ((-40, 10), (-80, 170))
triangle = ((140, 200), (180, 270), (335, 180))
rectangle = ((40, 100), (80, 170), (235, 80), (195, 10))
```

Turtle Graphics

For a little fun, open and run the [iterables/Demos/shapes.py](#) file. It

uses [turtle](#), a built-in graphics library, to draw shapes from tuples of coordinates.

Empty and Single-element Tuples

You're unlikely to have to create empty or single-element tuples very often, but in case you do...

Empty Tuple

```
t_empty = ()
```

Single Element Tuple

To create a single-element tuple, follow the element with a comma, like this:

```
t_single = ("a",)
```

If you do not include the comma, you just get a string as illustrated in the following code:

```
>>> t1 = ("a",)
>>> type(t1)
<class 'tuple'>
>>> t2 = ("a")
>>> type(t2)
<class 'str'>
```

Ranges

A range is an immutable sequence of numbers often used in for loops, which will be covered in the [Flow Control lesson](#). Ranges are created using `range()`, which can take one, two, or three arguments:

```
range(stop)
```

```
range(start, stop)
range(start, stop, step)
```

The following examples show the options for creating a range:

```
range(10) # range starting at 0 and ending at 9
range(5, 11) # range starting at 5 and ending at 10
range(0, 13, 3) # range starting at 0, ending at 12, in steps of 3
range(4, -4, -1) # range starting at 4, ending at -3, in steps of -1
```

Note that the `stop` number is not included in the range. You should read it as "from start up to *but not including* stop."

Converting Sequences to Lists

The easiest way to see how ranges work in the Python shell is to convert them into lists first as shown in the following code:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 11))
[5, 6, 7, 8, 9, 10]
>>> list(range(0, 13, 3))
[0, 3, 6, 9, 12]
>>> list(range(-4, 4))
[-4, -3, -2, -1, 0, 1, 2, 3]
```

We will revisit ranges when we discuss for loops.

Converting a Sequence to a List

You can convert any type of sequence to a list with the `list()` function:

```
>>> coords = (55, -23)
>>> list(coords)
[55, -23]
>>> list("Hello, world!")
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Indexing

In the **Strings** lesson, you learned how to find specific characters within a string using [indexing and slicing](#). All sequences can be indexed and sliced in this way.

Again, indexing is the process of finding a specific element within a sequence of elements through the element's position. If we consider a sequence from left to right, the first element (the left-most) is at position 0. If we consider a sequence from right to left, the first element (the right-most) is at position -1.

The following code shows how to use indexing with a list, but the same can be done with any sequence type. Try this out in the Python shell:

```
>>> fruit = ['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
>>> fruit[0]
'apple'
>>> fruit[-1]
'watermelon'
>>> fruit[4]
'lemon'
>>> fruit[-3]
'pear'
```

Exercise 17: Simple Rock, Paper, Scissors Game

15 to 20 minutes.

1. Open [iterables/Exercises/roshambo.py](#) in your editor.
2. Write the `main()` function so that it:
 - A. Creates a sequence with three elements: “Rock”, “Paper”, and “Scissors”.
 - B. Makes a random choice for the computer and stores it in a

variable.

- C. Prompts the user with 1 for Rock, 2 for Paper, 3 for Scissors:
- D. Prints out the computer's choice and then the user's choice.

The program should run like this:

```
1 for Rock, 2 for Paper, 3 for Scissors: 1
Computer: Scissors
User: Rock
```

Exercise Code: iterables/Exercises/roshambo.py

```
import random

def main():
    pass # replace this with your code

main()
```

Solution: iterables/Solutions/roshambo.py

```
import random

def main():
    roshambo = ["Rock", "Paper", "Scissors"]

    computer_choice = random.choice(roshambo)

    num = input("1 for Rock, 2 for Paper, 3 for Scissors: ")
    num = int(num) - 1
    user_choice = roshambo[num]

    print("Computer:", computer_choice)
    print("User:", user_choice)

main()
```

Slicing

Slicing is the process of getting a slice or segment of a sequence as a new sequence. The syntax is as follows:

```
sub_sequence = orig_sequence[first_pos:last_pos]
```

This returns a slice that starts with the element at `first_pos` and includes all the elements up to *but not including* the element at `last_pos`.

If `first_pos` is left out, then it is assumed to be 0. For example:

```
>>> ["a", "b", "c", "d", "e"][:3]
['a', 'b', 'c']
```

If `last_pos` is left out, then it is assumed to be the length of the sequence, or in other words, one more than the last position of the sequence. For example:

```
>>> ["a", "b", "c", "d", "e"][3:]
```

```
['d', 'e']
```

The following code shows how to slice a list, but the same can be done with any sequence type. Try this out in the Python shell:

```
>>> fruit = ["apple", "orange", "banana", "pear", "lemon", "watermelon"]
>>> fruit[0:5]
['apple', 'orange', 'banana', 'pear', 'lemon']
>>> fruit[1:4]
['orange', 'banana', 'pear']
>>> fruit[4:]
['lemon', 'watermelon']
>>> fruit[-3:]
['pear', 'lemon', 'watermelon']
>>> fruit[:3]
['apple', 'orange', 'banana']
>>> fruit[-4:-1]
['banana', 'pear', 'lemon']
>>> fruit[:]
['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
```

Exercise 18: Slicing Sequences

10 to 20 minutes.

1. Open [iterables/Exercises/slicing.py](#) in your editor.
2. Write the `split_list()` function so that it returns a list that contains two lists: the first and second half of the original list. For example, when passed `[1, 2, 3, 4]`, `split_list()` will return `[[1, 2], [3, 4]]`.
3. If the original list has an odd number of elements, the function should put the extra element in the first list. For example, when passed `[1, 2, 3, 4, 5]`, `split_list()` will return `[[1, 2, 3], [4, 5]]`.

When you run the program, it should output:

```
["red", "blue", "green"]  
['orange', 'purple']
```

Exercise Code: iterables/Exercises/slicing.py

```
import math

def split_list(orig_list):
    pass # replace this with your code

def main():
    colors = ["red", "blue", "green", "orange", "purple"]
    colors_split = split_list(colors)
    print(colors_split[0])
    print(colors_split[1])

main()
```

Solution: iterables/Solutions/slicing.py

```
import math

def split_list(orig_list):
    list_len = len(orig_list)
    mid_pos = math.ceil(list_len/2)
    list1 = orig_list[:mid_pos]
    list2 = orig_list[mid_pos:]
    return [list1, list2]
-----Lines Omitted-----
```

min(), max(), and sum()

min(iter) and max(iter)

The `min(iter)` function returns the smallest value of the passed-in iterable.

The `max(iter)` function returns the largest value of the passed-in iterable.

```
>>> colors = ["red", "blue", "green", "orange", "purple"]
>>> min(colors)
'blue'
>>> max(colors)
'red'
>>> ages = [27, 4, 15, 99, 33, 25]
>>> min(ages)
4
>>> max(ages)
99
```

Note that, for strings, uppercase letters are “smaller” than lowercase letters:

```
>>> min("NatDunn")
```

```
'D'
```

min() and max() with Multiple Arguments

The `min()` and `max()` functions can also take multiple arguments to compare. This is covered in the [Math](#) and [Strings](#) lessons.

`sum(iter[, start])`

The `sum()` function takes an iterable and adds up all of its elements and then adds the result to `start` (if it is passed in). For example:

```
>>> nums = range(1, 6)
>>> sum(nums) # 1 + 2 + 3 + 4 + 5
15
>>> sum(nums, 10)
25
```

Converting Sequences to Strings with `str.join(seq)`

The `join()` method of a string joins the elements of a sequence of strings on the given string. For example:

```
>>> colors = ["red", "blue", "green", "orange"]
>>> ','.join(colors)
'red,blue,green,orange'
>>> ', '.join(colors) # space after comma
'red, blue, green, orange'
>>> ':'.join(colors)
'red:blue:green:orange'
>>> ' '.join(colors)
'red blue green orange'
```

Note, the `join()` method will error if any of the elements in the sequence is not a string.

Splitting Strings into Lists

The `split()` method of a string splits the string into substrings. By default it splits on whitespace. For example:

```
>>> sentence = 'We are no longer the Knights Who Say "Ni!'"
>>> list_of_words = sentence.split()
>>> list_of_words
['We', 'are', 'no', 'longer', 'the', 'Knights', 'Who', 'Say', '']
```

`split()` takes an optional `sep` parameter to specify the separator:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(",")
['apple', ' banana', ' pear', ' melon']
```

Notice the extra space before “banana”, “pear”, and “melon”. To get rid of that space, you can specify a multi-character separator, like this:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(", ")
['apple', 'banana', 'pear', 'melon']
```

`split()` takes a second optional parameter, `maxsplit`, to indicate the maximum number of times to split the string. For example:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(", ", 2)
['apple', 'banana', 'pear, melon']
```

Notice `pear` and `melon` are part of the same string element. That’s because the string stopped splitting after two splits.

The `splitlines()` Method

The `splitlines()` method of a string splits a string into a list on line boundaries (i.e., line feeds (`\n`) and carriage returns (`\r`)). For example:

```
>>> fruit = """apple
```

```
    banana
    pear
    melon"""
>>> fruit.splitlines()
['apple', 'banana', 'pear', 'melon']
```

The `splitlines()` method is really useful when reading files. Consider the following text file:

Demo 28: iterables/data/states.txt

```
Alabama AL
Alaska  AK
Arizona AZ
Arkansas      AR
California    CA
-----Lines Omitted-----
```

Code Explanation

The file is a listing of states in the United States. Each line lists the name of the state and its abbreviation, separated by a newline.

The following code can be used to read this file into a list:

Demo 29: iterables/Demos/states.py

```
def main():
    with open('../data/states.txt') as f:
        states = f.read().splitlines()

    print(f'The file contains {len(states)} states.')

main()
```

Code Explanation

This will read the states from the file into a list. Run the file:

```
PS ...\\iterables\\Demos> python states.py
The file contains 50 states.
```

As you will soon see, having the data in a list makes it much easier to work with.

Unpacking Sequences

We learned earlier about [simultaneous assignment](#), which allows you to assign values to multiple variables at once, like this:

```
>>> first_name, last_name, company = "Nat", "Dunn", "Webucator"
>>> first_name
'Nat'
>>> last_name
'Dunn'
>>> company
'Webucator'
```

You can use the same concept to *unpack* a sequence into multiple variables, like this:

```
>>> about_me = ("Nat", "Dunn", "Webucator")
```

```
>>> first_name, last_name, company = about_me
>>> first_name
'Nat'
>>> last_name
'Dunn'
>>> company
'Webucator'
```

Dictionaries

Dictionaries are mappings that use arbitrary keys to map to values. They are like associative arrays in other programming languages. Dictionaries are created with curly braces and comma-delimited key-value pairs, like this:

```
>>> dict = {
    'key1': 'value 1',
    'key2': 'value 2',
    'key3': 'value 3'
}
>>> dict['key2'] = 'new value 2' # assign new value to existing key
>>> dict['key4'] = 'value 4' # assign value to new key
>>> print(dict['key1']) # print value of key
value 1
```

And here is a file showing a dictionary with meaningful content:

Demo 30: iterables/Demos/dict.py

```
grades = {
    "English": 97,
    "Math": 93,
    "Global Studies": 85,
    "Art": 74,
    "Music": 86
}

grades["Global Studies"] = 87 # assign new value to existing key
grades["Gym"] = 100 # assign value to new key

print(grades["Math"]) # print value of key
```

Common Dictionary Methods

The code samples for the methods that follow assume this dictionary:

```
grades = {
    "English": 97,
    "Math": 93,
    "Art": 74,
    "Music": 86
}
```

mydict.get(key[, default])

Returns the value for key if it is in mydict. Otherwise, it returns default if passed in or None if it is not.

```
>>> grades.get('English')
97
>>> grades.get('French') # returns None
>>> grades.get('French', 0)
0
```

mydict.pop(key[, default])

Removes and returns the value of `key` if it is in `mydict`. Otherwise, it returns `default` if passed in or a `KeyError` if it is not.

```
>>> grades.pop('English')
97
>>> grades # Notice 'English' has been removed
{'Math': 93, 'Art': 74, 'Music': 86}
>>> grades.pop('English')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'English'
>>> grades.pop('English', 'Not found')
'Not found'
```

mydict.popitem()

Removes and returns the last key/value pair entered as a tuple. [28](#)

```
>>> grades.popitem()
('Music', 86)
>>> grades # Notice 'Music' has been removed
{'Math': 93, 'Art': 74}
```

mydict.copy()

Returns a copy of `mydict`.

This works just like the `copy()` [method of lists](#):

```
>>> grades_copy = grades.copy()
>>> grades['English'] = 94 # Add 'English' back to grades
>>> grades['Music'] = 100 # Add 'Music' back to grades
>>> grades # Notice they're back
{'Math': 93, 'Art': 74, 'English': 94, 'Music': 100}
>>> grades_copy # But grades_copy still has the old data
{'Math': 93, 'Art': 74}
```

```
mydict.clear()
```

Removes all elements form mydict.

```
>>> grades.clear()
>>> grades # Dictionary is now empty
{}
```

```
update()
```

The `update()` method is used to add new key/value pairs to a dictionary or to overwrite the values of existing keys or both. It can take several types of arguments. Consider the following dictionary:

```
grades = {
    "English": 97,
    "Math": 93,
    "Art": 74,
    "Music": 86
}
```

All three of the following statements would modify the value of the "Math" key and add a new "Gym" key:

```
grades.update({"Math": 97, "Gym": 93}) # argument is dict with k
grades.update(Math=97, Gym=93) # individual arguments
grades.update([('Math', 97), ('Gym', 93)]) # argument is list of
```

The update() Method

Note that when updating a single key in a dictionary, it is possible to use the `update()` method, but it is generally preferable to use the subscript syntax:

```
grades['Math'] = 97
```

```
setdefault()
```

The `setdefault(key, default)` method works like this:

- If `key` does not exist in the dictionary, `key` is added with a value of `default`.
- If `key` exists in the dictionary, the value for `key` is left unchanged.

The following example illustrates how `setdefault()` works:

Demo 31: iterables/Demos/setdefault.py

```
grades = {
    "English": 97,
    "Math": 93,
    "Art": 74,
    "Music": 86
}

grades.setdefault("Art", 87) # Art key exists. No change.
print("Art grade:", grades["Art"])

grades.setdefault("Gym", 97) # Gym key is new. Added and set.
print("Gym grade:", grades["Gym"])
```

Code Explanation

The preceding code will render the following:

```
Art grade: 74
Gym grade: 97
```

Notice that the value of the Art key did not change. `setdefault()` really means `add_key_unless_key_already_exists()`.

When to Use `setdefault()`

Imagine you are creating a dictionary of grades from data in a file or a database. You cannot be sure what data that source contains, but you need grades for four specific subjects. You can populate your dictionary using the external data, and then use `setdefault()` to make sure you have data for all keys:

```
grades = get_data() # Imaginary function that returns dictionary
grades.setdefault('English') = 0
grades.setdefault('Math') = 0
grades.setdefault('Art') = 0
```



```
grades.setdefault('Music') = 0
```

The grade for any one of the keys in the `setdefault()` calls will only be 0 if the dictionary returned by `get_data()` doesn't include that key.

Dictionary View Objects

The following three methods return dictionary view objects:

- `mydict.keys()`
- `mydict.values()`
- `mydict.items()`

Try this out in Python interactive mode:

```
>>> grades = {
    "English": 97,
    "Math": 93,
    "Art": 75
}
>>> grades.keys()
dict_keys(['English', 'Math', 'Art'])
>>> grades.values()
dict_values([97, 93, 75])
>>> grades.items()
dict_items([('English', 97), ('Math', 93), ('Art', 75)])
```

As you can see, `dict_keys` and `dict_values` look like simple lists and `dict_items` looks like a list of tuples. But while they look like lists, they differ in two important ways:

1. Dictionary views do not support indexing or slicing as they have no set order.
2. Dictionary views cannot be modified. They provide dynamic views into a dictionary. When the dictionary changes, the views will change.

Consider the following:

Demo 32: iterables/Demos/dict_views.py

```
grades = {
    "English": 97,
    "Math": 93,
    "Global Studies": 85,
    "Art": 74,
    "Music": 86
}

grade_points = grades.values()
print("Grade points:", grade_points)

grades["Art"] = 87
print("Grade points:", grade_points)
```

Code Explanation

The output:

```
Grade points: dict_values([97, 93, 85, 74, 86])
Grade points: dict_values([97, 93, 85, 87, 86])
```

Notice that the Art grade changes in the output (from 74 to 87). There was no need to reassign `grade.values()` to `grade_points` as `grade_points` provides a dynamic view into the `grades` dictionary.

To get a list from a dictionary view, use the `list()` method:

```
>>> grades = {
    "English": 97,
    "Math": 93,
    "Art": 75
}
>>> list(grades.keys())
['English', 'Math', 'Art']
>>> list(grades.values())
```

```
[97, 93, 75]  
>>> list(grades.items())  
[('English', 97), ('Math', 93), ('Art', 75)]
```

Deleting Dictionary Keys

The `del` statement can be used to delete a specific key from a dictionary, like this:

```
grades = {  
    "English": 97,  
    "Math": 93,  
    "Global Studies": 85,  
    "Art": 74,  
    "Music": 86  
}
```

```
del grades["Math"] # deletes Math key  
print(grades)
```

The len() Function

The `len()` function can be used to determine the number of characters in a string or the number of objects in a list, tuple, dictionary, or set:

```
>> len("hello")  
5  
>>> len( ["a", "b", "c"] )  
3  
>>> len( (255, 0, 255) )  
3  
>>> len({"Math": 97, "Music": 86, "Global Studies": 85})  
3
```

Exercise 19: Creating a Dictionary from User Input

15 to 25 minutes.

1. Open [iterables/Exercises/gradepoints.py](#) in your editor.
2. Write the `main()` function so that it:
 - A. Creates a `grades` dictionary and populates it with grades entered by the user in English, Math, Global Studies, Art, and Music.

- B. Determines the average grade and prints it out. Note, to do this you will need to convert the user input to integers.

The program should run as follows:

```
English grade: 98
Math grade: 89
Global Studies grade: 79
Art grade: 91
Music grade: 84
Your average is 88.2
```

Challenge

After printing the average, ask the user to change the grade in one subject and then get the new average and print it out. **Hint:** you will have to prompt the user twice, once for the subject and once for the grade.

Solution: iterables/Solutions/gradepoints.py

```
def main():
    grades = {}
    grades["English"] = int(input("English grade: "))
    grades["Math"] = int(input("Math grade: "))
    grades["Global Studies"] = int(input("Global Studies grade: "))
    grades["Art"] = int(input("Art grade: "))
    grades["Music"] = int(input("Music grade: "))

    gradepoints = grades.values()

    average = sum(gradepoints)/len(gradepoints)

    print("Your average is", average)

main()
```

Challenge Solution: iterables/Solutions/gradepts-challenge.py

```
def avg(gradepts):
    average = sum(gradepts)/len(gradepts)
    return average

def main():
    grades = {}
    grades["English"] = int(input("English grade: "))
    grades["Math"] = int(input("Math grade: "))
    grades["Global Studies"] = int(input("Global Studies grade: "))
    grades["Art"] = int(input("Art grade: "))
    grades["Music"] = int(input("Music grade: "))

    gradepts = grades.values()

    average = avg(gradepts)

    print("Your average is", average)

    subject = input("Choose a subject to change your grade: ")
    new_grade = input("What is your new " + subject + " grade? ")
    grades[subject] = int(new_grade)
    average = avg(gradepts)

    print("Your new average is", average)

main()
```

Code Explanation

Note that, as our program now requires calculating the average more than one time, we have moved that functionality into a new `avg()` function.

Sets

Sets are *mutable* unordered collections of distinct *immutable* objects. So, while the set itself can be modified, it cannot be populated with objects that can be modified. You can also think of sets as dictionaries in which the keys have no values. In fact, they can be created with curly braces, just like dictionaries:

```
>>> classes = {"English", "Math", "Global Studies",  
               "Art", "Music"}  
>>> type(classes)  
<class 'set'>
```

Sets are less commonly used than the other iterables we've looked at in this lesson, but one great use for sets is to remove duplicates from a list as shown in the following example:

```
>>> veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "  
>>> v_set = set(veggies) # converts to set and remove duplicates  
>>> v_set  
{'pepper', 'tomato', 'spinach', 'pea'}  
>>> veggies = list(v_set) # converts back to list  
>>> veggies  
['pepper', 'tomato', 'spinach', 'pea']
```

This is often done in a single step, like this:

```
>>> veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "  
>>> veggies = list(set(veggies)) # remove duplicates
```

You could also create a `remove_dups()` function:

```
def remove_dups(the_list):  
    return list(set(the_list))  
  
veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"  
veggies = remove_dups(veggies)
```

veggies will now contain:

```
['pepper', 'tomato', 'spinach', 'pea']
```

***args and **kwargs**

When defining a function, you can include two special parameters to accept an arbitrary number of arguments:

- `*args` – A parameter that begins with a single asterisk will accept an arbitrary number of non-keyworded arguments and store them in a tuple. Often the variable is named `*args`, but you can call it whatever you want (e.g., `*people` or `*colors`). Note that any parameters that come after `*args` in the function signature are keyword-only parameters.
- `**kwargs` – A parameter that begins with two asterisks will accept an arbitrary number of keyworded arguments and store them in a dictionary. Often the variable is named `**kwargs`, but, as with `*args`, you can call it whatever you want (e.g., `**people` or `**colors`). When included, `**kwargs` must be the last parameter in the function signature.

The parameters in a function definition must appear in the following order:

1. Non-keyword-only parameters that are required (i.e., have no defaults).
2. Non-keyword-only parameters that have defaults.
3. `*args`
4. Keyword-only parameters (with or without defaults).
5. `**kwargs`

Using *args

Earlier, we saw a function that looked like this:

```
def add_nums(num1, num2, num3=0, num4=0, num5=0):  
    sum = num1 + num2 + num3 + num4 + num5  
    print(num1, "+", num2, "+", num3, "+",  
          num4, "+", num5, " = ", sum)
```

This works fine if you know that there will be between two and five numbers passed into `add_nums()`, but you can use `*args` to allow the function to accept an arbitrary number of numbers:

Demo 34: iterables/Demos/add_nums.py

```
def add_nums(num, *nums):
    total = sum(nums, num)
    print(f"The sum of {nums} and {num} is {total}.")

def main():
    add_nums(1, 2)
    add_nums(1, 2, 3, 4, 5)
    add_nums(11, 12, 13, 14)
    add_nums(101, 201, 301)

main()
```

Code Explanation

This will output:

```
The sum of (2,) and 1 is 3.
The sum of (2, 3, 4, 5) and 1 is 15.
The sum of (12, 13, 14) and 11 is 50.
The sum of (201, 301) and 101 is 603.
```

1. The `add_nums()` function requires one argument: `num`. It can also take zero or more subsequent arguments, which will all be stored in a single tuple, `nums`.
 2. We use the built-in `sum()` [function](#) to get the sum of `nums` and add it to `num`.
 3. The output is not perfect. We will improve it later on.
-

Using ****kwargs**

The `**kwargs` parameter is most commonly used when you need to pass an unknown number of keyword arguments from one function to another. While this can be very useful (e.g., in decorators), it's beyond

the scope of this lesson.

Conclusion

In this lesson, you have learned about lists, tuples, ranges, dictionaries, and sets. You also learned about the `*args` and `**kwargs` parameters. Soon, you'll learn to search these iterables for values and to loop through them performing operations on each element they contain one by one.

LESSON 7

Virtual Environments, Packages, and pip

Topics Covered

- Creating virtual environments.
- Activating and deactivating virtual environments.
- Installing packages with pip.
- Sharing project requirements so others can create a matching virtual environment.
- Deleting a virtual environment.

But, you see, the Land of Oz has never been civilized, for we are cut off from all the rest of the world. Therefore we still have witches and wizards amongst us.

– The Wonderful Wizard of Oz, L. Frank Baum

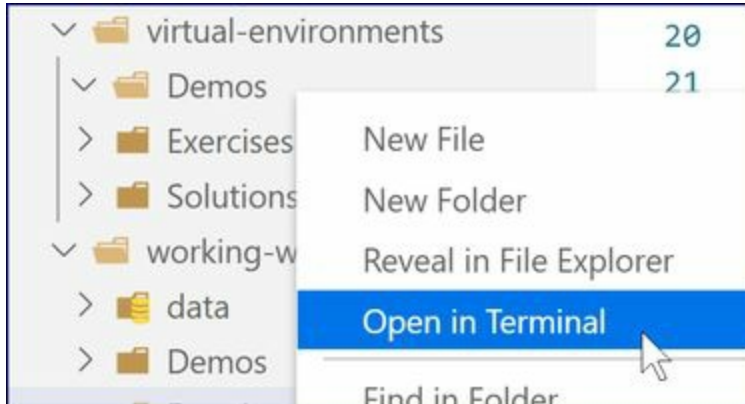
A virtual environment provides a self-contained directory tree with its own Python installation and additional packages necessary for the project(s) being done in that environment. As such, scripts can be run in a virtual environment that have dependencies that are different from those in other development projects that may be running in the standard environment or in separate virtual environments.

Exercise 20: Creating, Activating, Deactivating, and Deleting a Virtual Environment

20 to 30 minutes.

To create a virtual environment, you will use Python's built-in `venv` module.

1. Open a terminal at virtual-environments/Demos:

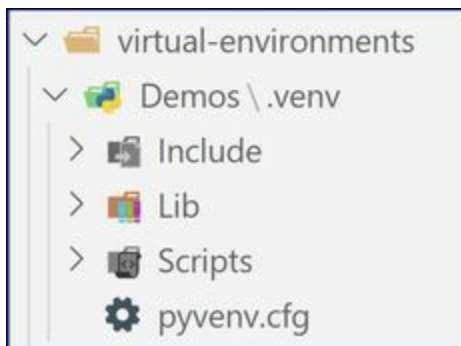


Then run the following command:

```
PS ...\virtual-environments\Demos> python -m venv .venv
```

This will create and populate a new .venv directory.^{[29](#)}

2. Take a look at the directory contents:



The contents will differ by operating system. Included in this directory is a Scripts (Windows) or bin (Mac) folder that contains the python executable file and scripts for activating the virtual environment.

3. To work within your virtual environment, you must first activate it. The command for activating a virtual environment varies by operating system. With virtual-environments/Demos still open at the terminal, run one of the following:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

4. The prompt text varies by operating system, terminal type, and settings. However, when the virtual environment is activated, its name will always appear enclosed in parentheses before the prompt. Here's what it looks like in PowerShell:

```
(.venv) PS ...\virtual-environments\Demos>
```

If you don't see the virtual environment name in parentheses before the prompt, you are not in the virtual environment.

5. You can now invoke the Python interpreter and/or install additional packages (using `pip`) within the virtual environment. Because the `$PATH` environment variable is modified when a virtual environment is activated, the executable files are resolved within the virtual environment. Let's take a look at the value of this environment variable. Run the following command for your environment:

Windows: PowerShell

```
$Env:PATH
```

Windows: Command Prompt

```
echo %PATH%
```

Mac / Linux

```
echo $PATH
```

You should see the path of the Scripts (Windows) or bin (Mac) directory prepended to the list of paths:

```
(.venv) PS ...\virtual-environments\Demos> $Env:PATH  
C:\Webucator\Python\virtual-environments\Demos\.venv\Scripts;
```


Because it is at the beginning of \$PATH, the Scripts or bin directory will be scanned first to resolve references to executable files. Once you deactivate the virtual environment, the Scripts and bin directory will be removed from \$PATH.

6. To deactivate (exit) the virtual environment, run:

```
(.venv) PS ..\virtual-environments\Demos> deactivate
```

Now, you are back in the original prompt.

7. When you no longer need a virtual environment, you can delete it by using operating system commands to delete the folder (e.g., .venv) that was automatically built when you created the virtual environment. Note that all your work will be lost, so if you want to keep specific files, you should copy and paste them into another directory before deleting.

Packages with pip

A Python package is a collection of related modules in a folder. The **Python Package Index** (PyPI) is a collection of freely available Python packages, which any developer can install using `pip`. For example, the playsound package³⁰ can be installed like this:

```
pip install playsound
```

To uninstall a package, use `pip uninstall`:

```
pip uninstall playsound
```

Packages and Virtual Environments

Often, it makes more sense to install packages within a virtual environment to keep your standard Python environment as clean as possible. If you create a new virtual environment for each project and install the necessary packages for each project within its virtual environment, then you know exactly what packages that project

requires (in addition to the packages you have installed in the standard environment). You can see a list of the packages installed using:

```
pip list
```

Run `pip list` at the prompt in your standard environment. You should see something like this (your packages may be different):

```
PS ...\\virtual-environments\\Demos\\> pip list
```

Package	Version
-----	-----
astroid	2.3.3
colorama	0.4.3
isort	4.3.21
lxml	4.5.0
mccabe	0.6.1
pip	20.0.2
pylint	2.4.4
setuptools	45.2.0
six	1.13.0
wrapt	1.11.2

In the next exercise, you will learn how to write these requirements to a requirements.txt file and then use that file to set up a new virtual environment. That way, when you share project code with other developers, you can include the requirements.txt file so they can easily set up a virtual environment for the project on their computer.

Exercise 21: Working with a Virtual Environment

15 to 25 minutes.

In this exercise you will:

1. Create and activate a virtual environment.
2. Install a package.

3. Create a requirements.txt file.
4. Deactivate and delete the virtual environment.
5. Recreate the virtual environment using the requirements.txt file.

Instructions

1. Open virtual-environments/Exercises in the terminal.
2. Create a virtual environment named `.venv`:

```
(.venv) PS ..\virtual-environments\Exercises> python -m venv .
```

3. Activate the virtual environment:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

4. Run `pip freeze`. This will output a list of the packages installed in the virtual environment. At this point, it won't return anything as we haven't installed any packages yet.
5. Install the `playsound` package:

```
(.venv) PS ..\virtual-environments\Exercises> pip install play
```

6. Now, when you run `pip freeze`, you will see the `playsound` package and its version:

```
(.venv) PS ..\virtual-environments\Exercises> pip freeze  
playsound==1.2.2
```

`pip freeze` outputs the installed packages in the format `pip` needs to install them. Write the output to a file in the Exercises directory

by running:

```
(.venv) PS ..\virtual-environments\Exercises> pip freeze > req
```

7. Open the new requirements.txt file in Visual Studio Code. You will see that it contains just one line showing the `playsound` package and its version.
8. In the Exercises directory, we have included some sound files and a demo.py file that makes use of `playsound` to play the sounds. At the terminal, run:

```
(.venv) PS ..\virtual-environments\Exercises> python demo.py
```

You should hear several sounds played.

9. Deactivate the virtual environment by running:

```
(.venv) PS ..\virtual-environments\Exercises> deactivate
```

0. Now, run this again:

```
(.venv) PS ..\virtual-environments\Exercises> python demo.py
```

Because the `playsound` library is only installed in the virtual environment, you should get an error like this one:

```
ModuleNotFoundError: No module named 'playsound'
```

1. Delete the virtual environment by deleting the .venv folder at the command line, in VS Code's Explorer, or using the file system.
2. Now, recreate the virtual environment and activate it. Run:

```
(.venv) PS ..\virtual-environments\Exercises> python -m venv .
```

Then activate it:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

3. Now, run:

```
(.venv) PS ..\virtual-environments\Exercises> python demo.py
```

Because you haven't installed the requirements, you should get an error like this one:

```
ModuleNotFoundError: No module named 'playsound'
```

4. Now, install the requirements:

```
(.venv) PS ..\virtual-environments\Exercises> pip install -r r
```

5. Run this again:

```
(.venv) PS ..\virtual-environments\Exercises> python demo.py
```

This time, it should run without errors.

In this exercise, we have only installed one package, but you can imagine a project that has many packages. Creating and sharing the requirements.txt file makes it easy for you to let other developers quickly create an identical virtual environment.

Conclusion

In this lesson, you have learned to create and use virtual environments, and you have learned to install packages with `pip`.

LESSON 8

Flow Control

Topics Covered

- `if` conditions in Python.
- Loops in Python.
- Generator functions.
- List comprehensions.

But the path began nowhere and ended nowhere, and it remained mystery, as the man who made it and the reason he made it remained mystery.

– The Call of the Wild, Jack London

By default, a program flows line by line in sequential order. We have seen already that we can change this flow by calling functions. The flow can also be changed using conditional statements and loops.

Conditional Statements

Conditional statements (`if-elif-else` conditions) allow programs to output different code based on specific conditions. The syntax is:

```
if some_conditions:
    do_this_1()
    do_this_2()
do_this_after()
```

Notice that the `do_this_1()` and `do_this_2()` function calls are both indented indicating that they are part of the `if` block. They will only run if `some_conditions` evaluates to `True`. The `do_this_after()` function is not indented. That indicates that the `if` block has ended and it will run

regardless of the value of `some_conditions`.

`elif` and `else`

- `elif` (for **else if**) conditions are only evaluated when the `if` condition is `False`. They are evaluated in order up until one evaluates to `True`, at which point, that `elif` block is executed and the rest of the `elif` conditions are skipped. An `if` statement can have zero or more `elif` conditions.
- The `else` block is only executed if the `if` condition and all the `elif` conditions evaluate to `False`. An `if` statement can have zero or one `else` conditions.

The following syntax blocks show an `if-else` and an `if-elif-else` statement:

```
if some_conditions:
    do_this_1()
    do_this_2()
else:
    do_this_3()
do_this_after()
```

```
if some_conditions:
    do_this_1()
    do_this_2()
elif other_conditions:
    do_this_3()
else:
    do_this_4()
    do_this_5()
do_this_after()
```

Values that Evaluate to False

The following values are considered `False`:

1. None
2. 0 (or 0.0)
3. Empty containers such as strings, lists, tuples, etc.

You can use the `bool()` function to demonstrate this:

```
>>> bool(None)
False
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
```

The following table shows Python's comparison operators:

Comparison Operators

Operator	Description
<code>==</code>	Equals
<code>!=</code>	Doesn't equal
<code>></code>	Is greater than
<code><</code>	Is less than
<code>>=</code>	Is greater than or equal to
<code><=</code>	Is less than or equal to
<code>is</code>	Is the same object
<code>is not</code>	Is not the same object

Python also includes these *membership* operators:

`in`

True if value is found in sequence.


```
'a' in ['a', 'b', 'c'] # True  
'a' in 'abc' # True
```

not in

True if value is **not** found in sequence.

```
'a' not in ['a', 'b', 'c'] # False  
'd' not in 'abc' # True
```

The following example demonstrates an `if-elif-else` statement.

Demo 35: flow-control/Demos/if.py

```
def main():
    age = int(input('How old are you? '))

    if age >= 21:
        print('You can vote and drink.')
    elif age >= 18:
        print('You can vote, but can\'t drink.')
    else:
        print('You cannot vote or drink.')

main()
```

Code Explanation

You can see the different results by running this file and entering different ages at the prompt:

```
PS ...\\flow-control\\Demos> python if.py
How old are you? 17
You cannot vote or drink.
PS ...\\flow-control\\Demos> python if.py
How old are you? 19
You can vote, but can't drink.
PS ...\\flow-control\\Demos> python if.py
How old are you? 21
You can vote and drink.
```

Compound Conditions

More complex if statements often require that several conditions be checked. The following table shows `and` and `or` operators for checking multiple conditions and the `not` operator for negating a boolean value (i.e., turning `True` to `False` or vice versa).

Logical Operators

--	--	--

Operator	Name	Example
and	AND	if a and b:
or	OR	if a or b:
not	NOT	if not a:

The following example shows these logical operators in practice:

Demo 36: flow-control/Demos/if2.py

```
def main():
    age = int(input('How old are you? '))

    is_citizen = (input('Are you a citizen? Y or N ').lower() == 'y')

    if age >= 21 and is_citizen:
        print('You can vote and drink.')
    elif age >= 18:
        print('You can drink, but can\'t vote.')
    elif age >= 18 and is_citizen:
        print('You can vote, but can\'t drink.')
    else:
        print('You cannot vote or drink.')

main()
```

Code Explanation

Notice that `is_citizen` is assigned to an comparison expression:

```
is_citizen = (input('Are you a citizen? Y or N ').lower() == 'y')
```

If the user enters “Y” or “y”, `is_citizen` will be `True`. Otherwise, it will be `False`.

The `is` and `is not` Operators

The `is` and `is not` operators differ from the `==` and `!=` operators. The former check whether two objects are the same object. The latter check whether two objects have the same value. The following code illustrates this:

```
>>> a = [1, 2, 3, 4]
>>> b = [1, 2, 3, 4]
>>> c = a
```

```

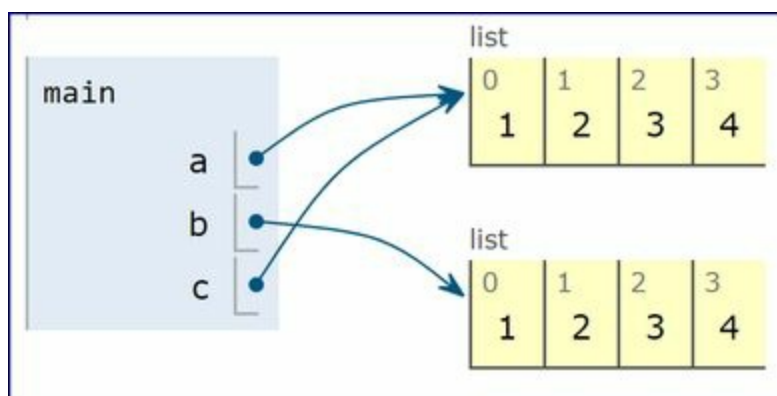
>>> a == b
True
>>> a is b
False
>>> a == c
True
>>> a is c
True
>>> id(a), id(b), id(c)
(1847278372096, 1847279744512, 1847278372096)
>>> a.append(5)
>>> c
[1, 2, 3, 4, 5]

```

Two identical lists are assigned to the `a` and `b` variables, but, because they are assigned separately, they are two separate objects. So, while `a == b` is `True`, `a is b` is `False`.

But `c` is assigned `a`, meaning `c` points to the same location in memory as `a` does, which is why the two variables have the same `id` and why, when we append 5 to `a` and then print `c`, we get the list containing 5.

The following diagram, which was generated at <http://www.pythontutor.com>, shows the status after all the variables have been assigned:



As you can see, `a` and `c` point to the same object:

`all()` and `any()`

The built-in `all()` and `any()` functions are used to loop through an iterable to check the boolean value of its elements.

- The `all()` function returns `True` if *all* the elements of the iterable evaluate to `True`.
- The `any()` function returns `True` if *any* of the elements of the iterable evaluate to `True`.

Ternary Operator

Many programming languages, including Python, have a ternary conditional operator, which is most often used for conditional variable assignment. To illustrate, consider this code:

```
if season == 'summer':  
    pant_color = 'white'  
else:  
    pant_color = 'black'
```

That's very clear, but it takes four lines of code to assign a value to `pant_color`.

The syntax for the ternary operator in Python is:

```
[on_true] if [expression] else [on_false]
```

Here is how we could use this syntax to assign our pants' color:

```
pant_color = 'white' if season == 'summer' else 'black'
```

It's still pretty clear, but much shorter. Note that the expression could be any type of expression, including a function call, that returns a value that evaluates to `True` or `False`. For example, if you had an `is_summer()` function, you could do this:

```
pant_color = 'white' if is_summer() else 'black'
```

In Between

To check if a value is between two values, use the following syntax:

```
if low_value < your_value < high_value:
```

For example:

```
age = 15
if 12 < age < 20:
    print("You are a teenager.")

if 13 <= age <= 19:
    print("You are a teenager.")
```

This same syntax works with other types of objects (e.g., strings and dates) as well.

Loops in Python

As the name implies, loops are used to loop (or iterate) over code blocks. The following section shows examples of the two different types of loops in Python: `while` loops and `for` loops.

You can find the loop examples used in this section in [flow-control/Demos/loops.py](#)

`while` Loops

`while` loops are used to execute a block of code repeatedly while one or more conditions evaluate to `True`.

```
num=0
while num < 6:
    print(num)
    num += 1
```

The preceding code will return:

```
0
1
2
3
4
5
```

You can combine `while` loops with user input to continually prompt the user until he or she enters an acceptable value. The following example illustrates this:

Demo 37: flow-control/Demos/while_input.py

```
def is_valid_age(s):
    return s.isdigit() and 1 <= int(s) <= 113

def main():
    age = input('How old are you? ')
    while not is_valid_age(age):
        age = input('Please enter a real age as a number: ')

    age = int(age)
    if age >= 21:
        print('You can vote and drink.')
    elif age >= 18:
        print('You can vote, but can\'t drink.')
    else:
        print('You cannot vote or drink.')

main()
```

Code Explanation

Run the file at the terminal and try entering some non-integer values:

```
PS ...\\flow-control\\Demos> python while_input.py
How old are you? twenty
Please enter a real age as a number: I'm 20
Please enter a real age as a number: 20!
Please enter a real age as a number: 20
You can vote, but can't drink.
```

The following demo shows how to combine `if` conditions and a `while` loop to create a simple game in which the user has to guess a number between 1 and 100. Review the code and the comments, which provide explanations.

Demo 38: flow-control/Demos/guess_the_number.py

```
import random

def is_valid_num(s):
    # Make sure the number is a digit between 1 and 100.
    return s.isdigit() and 1 <= int(s) <= 100

def main():
    # Get a random number between 1 and 100
    number = random.randint(1, 100)

    # Set guessed_number to False to start.
    # We'll set it to True when the user guesses the number
    guessed_number = False

    # Get the user's guess
    guess = input("Guess a number between 1 and 100: ")

    # Set num_guesses to 0.
    # We'll increment it by 1 after each guess
    num_guesses = 0

    # We'll keep looping until the user guesses the number
    while not guessed_number:
        # Make sure the number is valid
        if not is_valid_num(guess):
            print("I won't count that one.")
            guess = input("A number between 1 and 100 please: ")
        else: # Number is valid
            num_guesses += 1 # Increment num_guesses
            guess = int(guess) # Convert user's guess to an int

            # If guess is wrong, provide info and ask for another
            if guess < number:
                guess = input("Too low. Guess again: ")
            elif guess > number:
                guess = input("Too high. Guess again: ")
            else:
```

```
        print("You got it in", num_guesses, "guesses!")
        guessed_number = True # This will break us out of the loop

    print("Thanks for playing.")

main()
```

for Loops

A for loop is used to loop through an iterator (e.g., a range, list, tuple, dictionary, etc.). The syntax is as follows:

```
for item in sequence_name:
    do_something(item)
```

Looping through a range

```
for num in range(6):
    print(num)

for num in range(0, 6):
    print(num)
```

Both loops above will return:

```
0
1
2
3
4
5
```

Remember that a range goes up to but does not include the *stop* value.

Skipping Steps

```
for num in range(1, 11, 2): # 3rd argument is the step
    print(num)
```

This loop will return:

```
1
3
5
7
9
```

Stepping Backwards

```
for num in range(10, 0, -1):
    print(num)
```

This loop will return:

```
10
9
8
7
6
5
4
3
2
1
```

Looping through a list and a tuple

```
nums = [0, 1, 2, 3, 4, 5]
for num in nums:
    print(num)
```

```
nums = (0, 1, 2, 3, 4, 5)
for num in nums:
```

```
print(num)
```

Both loops above will return:

```
0  
1  
2  
3  
4  
5
```

Looping through a dict

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}  
  
for course in grades:  
    print(course)  
  
for course in grades.keys():  
    print(course)
```

Both of the loops above will return the courses (the keys):

```
English  
Math  
Art  
Music
```

Looping through the values of dict

```
for grade in grades.values():  
    print(grade)
```

This loop will return the grades (the values)

```
97  
93
```

```
74  
86
```

Looping through the items of dict

```
for course, grade in grades.items():  
    print(f'{course}: {grade}')
```

This loop will return the courses (the keys) and grades (the values):

```
English: 97  
Math: 93  
Art: 74  
Music: 86
```

Notice the unpacking of the variables:

```
for course, grade in grades.items():
```

Remember that the `items()` method returns a `dict_items` object, which is a pseudo-list of tuples:

```
dict_items([('English', 97), ('Math', 93), ('Art', 74), ('Music', 86)])
```

The first item in each of the tuples above is the course and the second item is the grade.

Another way of writing the loop shown above would be:

```
for item in grades.items():  
    course = item[0]  
    grade = item[1]  
    print(f'{course}: {grade}')
```

But, the original way, which unpacks the tuples into the `course` and `grade` variables with each iteration, is more *pythonic*.

Exercise 22: All True and Any True

In this exercise, you will use loops to write `all_true()` and `any_true()` functions that behave the same way as the built-in `all()` and `any()` functions.

1. Open `flow-control/Exercises/all_and_any.py` in your editor.
2. In the `main()` function, there are calls to `all_true()` and `any_true()`, but those functions have yet to be written. Complete those functions:
 - `all_true()` – should return `True` if (and only if) all elements in the passed-in iterable evaluate to `True`.
 - `any_true()` – should return `True` if at least one element in the passed-in iterable evaluates to `True`.
3. Keep in mind that you want your function to return a value as soon as it can. For example, if you pass a list of 1,000,000 values to `all_true()` and the first value is `False`, the function does not need to continue looping through the list to determine that not all values are `True`.

Exercise Code: flow-control/Exercises/all_and_any.py

```
def all_true(iterable):  
    # write function  
    pass
```

```
def any_true(iterable):  
    # write function  
    pass
```

```
def main():  
    a = all_true([1, 0, 1, 1, 1])  
    b = all_true([1, 1, 1, 1, 1])  
    c = any_true([0, 0, 0, 1, 1])  
    d = any_true([0, 0, 0, 0, 0])  
  
    print(a, b, c, d) # Should be: False True True False  
  
main()
```

Solution: flow-control/Solutions/all_and_any.py

```
def all_true(iterable):
    # Return False if any item is False
    for item in iterable:
        if not item:
            return False
    # If we made it through the loop without returning False,
    # then all items are True
    return True

def any_true(iterable):
    # Return True if any item is True
    for item in iterable:
        if item:
            return True
    # If we made it through the loop without returning True,
    # then all items are False
    return False

def main():
    a = all_true([1, 0, 1, 1, 1])
    b = all_true([1, 1, 1, 1, 1])
    c = any_true([0, 0, 0, 1, 1])
    d = any_true([0, 0, 0, 0, 0])

    print(a, b, c, d) # Should be: False True True False

main()
```

break and continue

To break out of a loop, insert a break statement:

```
for num in range(11, 20):
    print(num)
    if num % 5 == 0:
        break
```

Because `15 % 5` is equal to `0` (i.e., `15 / 5` has a remainder of `0`), the loop will stop after printing `15`:

```
11
12
13
14
15
```

To jump to the next iteration of a loop without executing the remaining statements in the block, insert a `continue` statement:

```
for num in range(1, 12):
    if num % 5 == 0:
        continue
    print(num)
```

This will skip to the next iteration when `num` is divisible by `5`, resulting in:

```
1
2
3
4
6
7
8
9
11
```

Notice `5` and `10` are not printed.

Here's our Guess-the-Number game modified to use `continue` instead of an `else` block:

```
import random

def is_valid_num(s):
    return s.isdigit() and 1 <= int(s) <= 100

def main():
    number = random.randint(1, 100)
    guessed_number = False
    guess = input("Guess a number between 1 and 100: ")
    num_guesses = 0

    while not guessed_number:
        if not is_valid_num(guess):
            print("I won't count that one.")
            guess = input("A number between 1 and 100 please: ")
            continue # Skip to next iteration through loop

        num_guesses += 1
        guess = int(guess)

        if guess < number:
            guess = input("Too low. Guess again: ")
        elif guess > number:
            guess = input("Too high. Guess again: ")
        else:
            print("You got it in", num_guesses, "guesses!")
            guessed_number = True

    print("Thanks for playing.")

main()
```

Looping through Lines in a File

[In an earlier lesson](#), we showed how to use the `splitlines()` method of a string to split the lines of text read from a file into a list. Let's look again at the [states.txt](#) file we saw in that lesson:

Demo 40: flow-control/data/states.txt

```
Alabama AL
Alaska  AK
Arizona AZ
Arkansas      AR
California    CA
-----Lines Omitted-----
```

Code Explanation

Again, this file is a listing of states in the United States. Each line lists the name of the state and its abbreviation, separated by a tab.

Let's see how we can use the data in this file to write a program that allows the user to enter a state abbreviation and get the state name in return:

Demo 41: flow-control/Demos/get_state.py

```
def get_state(abbreviation):
    # Read the data from the file into a list
    with open('../data/states.txt') as f:
        states = f.read().splitlines()

    # Loop through the list
    for state_row in states:
        # Split each row on the tab character into a
        # two-element list
        state = state_row.split('\t')

        # If the 2nd element matches the passed-in abbreviation
        # return the first element
        if state[1] == abbreviation.upper():
            return state[0]

    # If no state matched the abbreviation, return None
    return None

def main():
    # Loop until break statement
    while True:
        abbr = input('State abbreviation (q to quit): ').upper()

        # Allow user to break loop by entering "Q"
        if abbr == 'Q':
            print('Goodbye!')
            break

        # Get state name from abbreviation
        state = get_state(abbr)

        # Inform the user what state, if any, maps to the abbrev:
        if state:
            print(f'"{abbr}" is the abbreviation for {state}.')
        else:
            print(f'No state has "{abbr}" as an abbreviation.')
```

```
main()
```

Code Explanation

Read through this file carefully, paying particular attention to the comments. Then run the file. Here is a sample result:

```
PS ...\\flow-control\\Demos> python get_state.py
State abbreviation (q to quit): ny
"NY" is the abbreviation for New York.
State abbreviation (q to quit): ca
"CA" is the abbreviation for California.
State abbreviation (q to quit): as
No state has "AS" as an abbreviation.
State abbreviation (q to quit): tx
"TX" is the abbreviation for Texas.
State abbreviation (q to quit): q
Goodbye!
```

Exercise 23: Word Guessing Game

45 to 120 minutes.

In this exercise, you will create a word guessing game similar to hangman, but without a limit on the number of guesses.

You may find this exercise quite challenging. Just do your best and work your way through it one step at a time. Start by carefully reviewing the code. You may also find it helpful to play the game a couple of times by running `python guess_word.py` at the terminal from the [flow-control/Solutions](#) folder. A completed game would look like this:

```
PS ...\\flow-control\\Solutions> python guess_word.py
The word contains 9 letters.
```

```
Guess a letter or a 9-letter word: A
Sorry. The word has no "A"s.
*****
Guess a letter or a 9-letter word: O
The word has 3 "O"s.
*OO***O**
Guess a letter or a 9-letter word: S
The word has one "S".
*OO*S*O**
Guess a letter or a 9-letter word: A
You already guessed "A".
Guess a letter or a 9-letter word: CK
Invalid entry.
Guess a letter or a 9-letter word: Woodstock
WOODSTOCK is it! It took 4 tries.
```

You should spend as much time as it takes to get this working or nearly working. Refrain from checking the solution until you have really given it your best shot. Learning to work through problems patiently and methodically, to read Python's error messages, and to debug your code are essential programming skills that are only attainable through practice.

The starting code is shown below:

Exercise Code: flow-control/Exercises/guess_word.py

```
import random

def get_word():
    """Returns random word."""
    words = ['Charlie', 'Woodstock', 'Snoopy', 'Lucy', 'Linus',
            'Schroeder', 'Patty', 'Sally', 'Marcie']
    return random.choice(words).upper()

def check(word, guesses):
    """Creates and returns string representation of word
    displaying asterisks for letters not yet guessed."""
    status = '' # Current status of guess
    last_guess = guesses[-1]
    matches = 0 # Number of occurrences of last_guess in word

    # Loop through word checking if each letter is in guesses
    # If it is, append the letter to status
    # If it is not, append an asterisk (*) to status
    # Also, each time a letter in word matches the last guess,
    # increment matches by 1.

    # Write a condition that outputs one of the following when
    # the user's last guess was "A":
    # 'The word has 2 "A"s.' (where 2 is the number of matches
    # 'The word has one "A".'
    # 'Sorry. The word has no "A"s.'

    return status

def main():
    word = get_word() # The random word
    n = len(word) # The number of letters in the random word
    guesses = [] # The list of guesses made so far
    guessed = False
    print('The word contains {} letters.'.format(n))

    while not guessed:
```



```

    guess = input('Guess a letter or a {}-letter word: '.format(n))
    guess = guess.upper()
    # Write an if condition to complete this loop.
    # You must set guessed to True if the word is guessed.
    # Things to be looking for:
    # - Did the user already guess this guess?
    # - Is the user guessing the whole word?
    #   - If so, is it correct?
    # - Is the user guessing a single letter?
    #   - If so, you'll need your check() function.
    # - Is the user's guess invalid (the wrong length)?
    #
    # Also, don't forget to keep track of the valid guesses.

    print('{} is it! It took {} tries.'.format(word, len(guesses)))

main()

```

1. Open [flow-control/Exercises/guess_word.py](#) in your editor. The program consists of three functions:
 - A. `get_word()` – returns a random secret word to guess. This function is complete.
 - B. `check()` – described below. You must complete this function.
 - C. `main()` – our main program. You must complete this function.
2. The program selects a secret word and prints: “The word contains *n* letters.” For example:


```
The word contains 6 letters.
```
3. The program then continuously prompts the user to choose a letter or guess the word until the word is guessed: “Guess a letter or a *n*-letter word: ”. For example:


```
Guess a letter or a 6-letter word:
```
4. The program keeps all previous guesses in a `guesses` list.

5. For each guess:

- A. **If the user has already guessed that letter or word**, the program prints “You already guessed *guess*.” For example:

```
You already guessed "A".
```

It then prompts for another guess.

- B. **Otherwise, if the user enters a word of the same length as the secret word**, the program records the guess in `guesses` and then checks to see if it is correct.

- i. If the user’s guess is correct, the program prints “*GUESS* is it! It took *n* tries.” For example:

```
SNOOPY is it! It took 8 tries.
```

The game then ends.

- ii. If the user’s guess is incorrect, the program prints “Sorry, that is incorrect.” It then prompts for another guess.

- C. **Otherwise, if the user enters a single character**, the program records the guess in `guesses` and then checks to see if the letter is in the word. It should do this by calling the `check()` function and passing it the secret word and `guesses`:

```
result = check(word, guesses)
```

- i. The `check()` function must do the following:
- a. Print one of the following based on the number of times the last guess in `guesses` shows up in the secret word:
1. **Multiple times:** “The word has *n* *guess*’s.” For example:

```
The word has 2 "O"s.
```
 2. **Exactly one time:** “The word has one *guess*.” For

example:

```
The word has one "S".
```

3. **Zero times:** “Sorry. The word has no *guess*’s.” For example:

```
Sorry. The word has no "E"s.
```

- b. It should then return a string representation of the word displaying asterisks for letters not yet guessed. For example, if the word is SNOOPY and o and Y have been guessed, it would return **oo*Y.
- ii. Back in the `main()` function, the value returned from the `check()` function should be compared to the secret word:
- a. **If the two values are the same**, the program prints “*GUESS* is it! It took *n* tries.” For example:

```
SNOOPY is it! It took 8 tries.
```

The game then ends.

- b. **If the two values are different**, the program prints the value returned from `check()` (e.g., **oo*Y).
- D. **Otherwise**, the program prints “Invalid Entry” and prompts the user for another guess. We only reach this block if the user enters a multi-character string that is not the same length as the secret word.

Challenge

Create your own list of words in a separate file in the `flow-control/data` folder (or use the Harry Potter spells in the `words.txt` file that’s already there) and use the words from that file in the `get_word()` function.

Solution: flow-control/Solutions/guess_word.py

-----Lines Omitted-----

```
def check(word, guesses):
    """Creates and returns string representation of word
    displaying asterisks for letters not yet guessed."""
    status = '' # Current status of guess
    last_guess = guesses[-1]
    matches = 0 # Number of occurrences of last_guess in word

    for letter in word:
        status += letter if letter in guesses else '*'

        if letter == last_guess:
            matches += 1

    if matches > 1:
        print('The word has {} "{}s.".format(matches, last_guess))
    elif matches == 1:
        print('The word has one "{}.".format(last_guess))
    else:
        print('Sorry. The word has no "{}s.".format(last_guess))

    return status

def main():
    word = get_word() # The random word
    n = len(word) # The number of letters in the random word
    guesses = [] # The list of guesses made so far
    guessed = False
    print('The word contains {} letters.'.format(n))

    while not guessed:
        guess = input('Guess a letter or a {}-letter word: '.format(n))
        guess = guess.upper()
        if guess in guesses:
            print('You already guessed "{}.".format(guess))
        elif len(guess) == n: # Guessing whole word
            guesses.append(guess)
```

```
        if guess == word:
            guessed = True
        else:
            print('Sorry, that is incorrect.')
    elif len(guess) == 1: # Guessing letter
        guesses.append(guess)
        result = check(word, guesses)
        if result == word:
            guessed = True
        else:
            print(result)
    else: # guess had wrong number of characters
        print('Invalid entry.')
```

```
    print('{} is it! It took {} tries.'.format(word, len(guesses)))
```

```
main()
```

Challenge Solution: `flow-control/Solutions/guess_word_challenge.py`

```
import random

def get_word():
    """Returns random word."""
    with open('../data/words.txt') as f:
        words = f.read().splitlines()
    return random.choice(words).upper()
-----Lines Omitted-----
```

If you were not able to get your program to work, study the solutions and then go back and try to fix your code. If you're not able to fix it after studying it once, try again. That iterative process of attacking a problem is excellent practice for real-world development.

The `else` Clause of Loops

In Python, `for` and `while` loops have an optional `else` clause, which is executed after the loop has successfully completed iterating (i.e., without a `break`). The following demo shows how it works:

Demo 42: flow-control/Demos/loop_else.py

```
def main():
    print('Example 1: for loop')
    num = int(input('Enter a number: '))
    for i in range(5):
        if i == num:
            break
        print(i)
    else:
        print(f'Completed iterating without reaching {num}.')

    print('\nExample 2: while loop')
    i = 0
    num = int(input('Enter a number: '))
    while i <= 5:
        if i == num:
            break
        print(i)
        i += 1
    else:
        print(f'Completed iterating without reaching {num}.')

main()
```

Code Explanation

The preceding code will render something like the following:

First Run

```
PS ...\\flow-control\\Demos> python loop_else.py
Example 1: for loop
Enter a number: 4
0
1
2
3
```

Example 2: while loop

Enter a number: 3

0

1

2

Second Run

```
PS ...\\flow-control\\Demos> python loop_else.py
```

Example 1: for loop

Enter a number: 7

0

1

2

3

4

Completed iterating without reaching 7.

Example 2: while loop

Enter a number: 6

0

1

2

3

4

5

Completed iterating without reaching 6.

The first run through, we entered numbers lower than 5, so the loops were broken out of before completing. The second run through, we entered numbers higher than 5, so the loops completed all iterations, which led to the `else` clause also being executed.

So, when would you use this? You might use it to check user-entered text for black-listed words. The following code shows a common way to do this in other programming languages:


```
sentence = input('Input a sentence: ')
found_bad_word = False
for word in sentence.split():
    if word in BLACK_LIST:
        found_bad_word = True
        break

if found_bad_word:
    print('You used a naughty word.')
else:
    print('Sentence passes cleanliness test.')
```

The code in the following file shows how you can accomplish the same thing in a cleaner way using the `else` clause:

Demo 43: flow-control/Demos/loop_else_use_case.py

```
BLACK_LIST = [
    'shitake',
    'sugar',
    'fudge',
    'gosh',
    'darn',
    'dang',
    'heck'
]

def main():
    sentence = input('Input a sentence: ')

    for word in sentence.split():
        if word in BLACK_LIST:
            print('You used a naughty word.')
            break
    else:
        print('Sentence passes cleanliness test.')

main()
```

Code Explanation

Here are the results of running this file, first with a “clean” sentence and then with a “dirty” one:

```
PS ...\\flow-control\\Demos> python loop_else_use_case.py
Input a sentence: I like cream in my coffee.
Sentence passes cleanliness test.
PS ...\\flow-control\\Demos> python loop_else_use_case.py
Input a sentence: I like sugar in my coffee.
You used a naughty word.
```

While using the `else` clause is not functionally better than the other

method shown, it is cleaner, and it saves us from having to create and check the `found_bad_word` flag.

Exercise 24: `for...else`

10 to 20 minutes.

In this exercise, you will rewrite a function to use the `else` clause of a loop.

1. Open `flow-control/Exercises/states.py` in your editor and review the code. Then run the script to see how it works.

A. First, try entering only valid states:

```
PS ...\\flow-control\\Exercises> python states.py
Name as many state abbreviations do you know?
Separate them with spaces:
AZ CA NY
You named 3 states.
```

B. Now, try including an invalid state:

```
PS ...\\flow-control\\Exercises> python states.py
Name as many state abbreviations do you know?
Separate them with spaces:
AZ CA CC NY
CC is not a state.
```

2. Modify the `main()` function so that it uses the `for` loop's `else` clause.
3. Run the script again. It should work in exactly the same way.

Exercise Code: flow-control/Exercises/states.py

```
def is_state(state):
    with open('../data/states.txt') as f:
        states = f.read().splitlines()

    state_abbreviations = []
    for state_row in states:
        state_abbreviation = state_row.split('\t')[1]
        state_abbreviations.append(state_abbreviation)

    return state in state_abbreviations

def main():
    print('Name as many state abbreviations do you know?')
    print('Separate them with spaces:')
    states = input('').split()
    bad_state = False
    for state in states:
        state = state.upper()
        if not is_state(state):
            print(f'{state} is not a state.')
            bad_state = True
            break

    if not bad_state:
        print(f'You named {len(states)} states.')

main()
```

Solution: flow-control/Solutions/states.py

-----Lines Omitted-----

```
def main():
    print('Name as many state abbreviations do you know?')
    print('Separate them with spaces:')
    states = input('').split()
    for state in states:
        state = state.upper()
        if not is_state(state):
            print(f'{state} is not a state.')
            break
    else:
        print(f'You named {len(states)} states.')

main()
```

The enumerate() Function

It is common in other programming languages to write code like this:

Demo 44: flow-control/Demos/without_enum.py

```
i = 1
for item in ['a', 'b', 'c']:
    print(i, item, sep='. ')
    i += 1
```

Code Explanation

This code will output an enumerated list:

```
1. a
2. b
3. c
```

The more Pythonic way of accomplishing the same thing is to use the `enumerate()` function, like this:

Demo 45: flow-control/Demos/with_enum.py

```
for i, item in enumerate(['a', 'b', 'c'], 1):  
    print(i, item, sep='. ')
```

This saves us from creating a new variable to hold the count.

Note that `enumerate()` takes two arguments:

1. The iterable to enumerate.
2. The number at which to start counting (defaults to 0).

It returns an iterable of two-element tuples of the format (count, value).

Here is a more practical example, in which we output a list of all the state names in [states.txt](#):

Demo 46: flow-control/Demos/state_list.py

```
def main():
    with open('../data/states.txt') as f:
        states = f.read().splitlines()

    for i, state in enumerate(states, 1):
        state_name = state.split('\t')[0]
        print(f'{i}. {state_name}')

main()
```

Generators

Generators are special iterators that can only be iterated through one time. Generators are created with special generator functions. Before looking at one, let's first consider how a standard iterator (e.g., a list) works:

Demo 47: flow-control/Demos/list_loop.py

```
import random

def get_rand_nums(low, high, num):
    numbers = []
    for number in range(num):
        numbers.append(random.randint(low, high))
    return numbers

numbers = get_rand_nums(1, 100, 5)

print('First time through:')
for num in numbers:
    print(num)

print('Second time through:')
for num in numbers:
    print(num)
```

Code Explanation

This code will output something like:

```
First time through:
13
63
50
10
16
Second time through:
13
63
50
10
16
```

There is nothing new in this code. The `get_rand_nums()` function

returns a list of `num` random numbers between `low` and `high`. We assign that list to `numbers` and then loop through it twice. We can loop through it any number of times.

Now, consider this code, which uses a generator:

Demo 48: flow-control/Demos/generator_loop.py

```
import random

def get_rand_nums(low, high, num):
    for number in range(num):
        yield random.randint(low, high)

numbers = get_rand_nums(1, 100, 5)

print('First time through:')
for num in numbers:
    print(num)

print('Second time through:')
for num in numbers:
    print(num)
```

Code Explanation

This code will output something like:

```
First time through:
69
32
65
76
87
Second time through:
```

Here are the two functions again:

List Loop

```
def get_rand_nums(low, high, num):
    numbers = []
    for number in range(num):
        numbers.append(random.randint(low, high))
```

```
    return numbers
```

Generator

```
def get_rand_nums(low, high, num):  
    for number in range(num):  
        yield random.randint(low, high)
```

The two functions work in essentially the same way, but the second function is much cleaner as there is no need for the local `numbers` variable. Rather than creating a full list and then returning it, the generator function yields each result one by one as it works its way through the `for` loop. The only functional difference, as the example illustrates, is that you can only iterate through the generator one time. However, if you want to iterate through multiple times, you can simply call the generator function again as shown in the next example:

Demo 49: flow-control/Demos/generator_loop2.py

```
import random

def get_rand_nums(low, high, num):
    for number in range(num):
        yield random.randint(low, high)

print('First time through:')
for num in get_rand_nums(1, 100, 5):
    print(num)

print('Second time through:')
for num in get_rand_nums(1, 100, 5):
    print(num)
```

Code Explanation

This will return something like:

```
First time through:
56
19
99
33
8
Second time through:
36
88
76
54
80
```

As the function is returning a sequence of random numbers, you will get different numbers each time you call the generator function.

If you want a sequence of random numbers that you can count on being the same each time you iterate through it, then you should

create a function that returns a list.

When to Use Generators

Use generators whenever you want to be able to get the next item in a sequence without creating the whole sequence ahead of time. Some examples:

Infinite Sequences

Demo 50: flow-control/Demos/odd_numbers.py

```
def odd_numbers():
    i = -1
    while True:
        i += 2
        yield i

def main():
    for i in odd_numbers():
        print(i)
        if input('Enter for next or q to quit: ') == 'q':
            print('Goodbye!')
            break

main()
```

Code Explanation

It's impossible to store the infinite number of odd numbers in a list. This generator function just yields the next one every time it is called:

```
PS ...\\flow-control\\Demos> python odd_numbers.py
1
Enter for next or q to quit:
3
Enter for next or q to quit:
5
Enter for next or q to quit: q
Goodbye!
```

The Fibonacci Sequence

The Fibonacci sequence is a sequence in mathematics in which each number is the sum of the two preceding numbers.

Demo 51: flow-control/Demos/fibonacci.py

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def main():
    for i in fibonacci():
        print(i)
        if input('Enter for next or q to quit: ') == 'q':
            print('Goodbye!')
            break

main()
```

Code Explanation

See https://en.wikipedia.org/wiki/Fibonacci_number for information on the Fibonacci Sequence.

A Sequence of Unknown Length

In Bingo, the caller keeps picking out numbers until someone wins. It's impossible to know ahead of time how many numbers must be drawn.


```
import random

def bingo():
    balls = {
        'B': list(range(1, 16)),
        'I': list(range(16, 31)),
        'N': list(range(31, 46)),
        'G': list(range(46, 61)),
        'O': list(range(61, 76))
    }

    while balls:
        letter = random.choice(list(balls.keys()))
        number = random.choice(balls[letter])
        balls[letter].remove(number) # Remove number from letter
        if not balls[letter]: # Letter has no more numbers
            print('Removing', letter)
            del balls[letter] # Delete letter from dictionary
        yield (letter, number)

def main():
    for ball in bingo():
        print(ball)
        if input('Enter for next ball or q to quit') == 'q':
            print('Goodbye!')
            break
    else:
        print('All out of balls.')

main()
```

Code Explanation

This generator function keeps spitting out Bingo balls until it runs out of balls.

Here's a Bingo card, in case you want to try your luck:

B I N G O				
14	26	37	51	66
15	27	42	57	74
8	28	*	47	64
4	19	39	59	72
3	17	45	50	71

The `next()` Function

To be an iterator, an object must have a special `__next__()` method (that's two underscores before and two underscores after "next"), which returns the next item of the iterator. Python's built-in `next(iter)` function calls the `__next__()` method of `iter`. The following example shows how to use the `next()` function with a generator to play Rock, Paper, Scissors (RoShamBo) against your computer.

Demo 53: flow-control/Demos/roshambo.py

```
import random

def roshambo(weapons):
    while True:
        yield random.choice(weapons)

def play(weapons, choice, python_weapons):
    your_weapon = weapons[int(choice)-1]
    python_weapon = next(python_weapons)

    if your_weapon == python_weapon:
        print('Tie: You both chose', your_weapon)
    elif ((your_weapon == 'Scissors' and python_weapon == 'Paper'
          or (your_weapon == 'Paper' and python_weapon == 'Rock'
              or (your_weapon == 'Rock' and python_weapon == 'Scissors'))):
        print('You win:', your_weapon, 'beats', python_weapon)
    else:
        print('You lose:', python_weapon, 'beats', your_weapon)
    print('-----')

def make_choice():
    choice = input("""Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
""")
    return choice

def main():
    weapons = ['Rock', 'Paper', 'Scissors']
    python_weapons = roshambo(weapons)
    choice = make_choice()
    while choice in ['1', '2', '3']:
        play(weapons, choice, python_weapons)
        choice = make_choice()
```

```
print('Goodbye!')
```

```
main()
```

Code Explanation

One benefit of assigning a generator to `python_weapons` instead of a list is that you don't need to know how many iterations there will be. The generator just spits out a new result each time its `__next__()` method is called.

Here is a possible output of the game:

```
PS ...\\flow-control\\Demos> python roshambo.py
Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
1
You win: Rock beats Scissors
-----
Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
2
You lose: Scissors beats Paper
-----
Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
q
Goodbye!
```

Note that the `roshambo()` generator is overly simplistic. On line 9, we could have just used `random.choice(weapons)` to get the value of `python_weapon`. But imagine that the generator did something more than that. For example, you could give it weapon preferences like this:

```
def roshambo(weapons):
    while True:
        num = random.random()
        if num < .5:
            yield weapons[0]
        elif num < .8:
            yield weapons[1]
        else:
            yield weapons[2]
```

List Comprehensions

List comprehensions are used to create new lists from existing sequences by taking a subset of that sequence and/or modifying its members. To understand the purpose, let's first look at creating a new list from a list using a `for` loop:

```
squares = []
for i in range(10):
    squares.append(i*i)

print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This same thing can be done much more tersely with a list comprehension. The syntax is:

```
new_list = [modify(item) for item in items]
```

Where `modify(item)` represents any change made to `item`.

Using a list comprehension, the code above could be rewritten like this:

```
squares = [i*i for i in range(10)]
```

Here's another example, in which we use a list comprehension to create a list of people's initials from a list of their names:

Demo 54: flow-control/Demos/list_from_list.py

```
def initials(name):
    fullname = name.split(' ')
    inits = (fullname[0][0], fullname[1][0])
    return inits

def main():
    names = ['Graham Chapman', 'John Cleese', 'Eric Idle',
            'Terry Gilliam', 'Terry Jones', 'Michael Palin']
    inits = [initials(name) for name in names]
    for i in inits:
        print(f'{i[0]}.{i[1]}.')

main()
```

Code Explanation

The list comprehensions creates a new list from `names` by passing each of its elements to the `initials()` function. This will return:

```
G.C.
J.C.
E.I.
T.G.
T.J.
M.P.
```

Remember our `add_nums()` function from the **Iterables** lesson:

Demo 55: iterables/Demos/add_nums.py

```
def add_nums(num, *nums):
    total = sum(nums, num)
    print(f"The sum of {nums} and {num} is {total}.")

def main():
    add_nums(1, 2)
    add_nums(1, 2, 3, 4, 5)
    add_nums(11, 12, 13, 14)
    add_nums(101, 201, 301)

main()
```

When you call that function like this:

```
add_nums(1, 2, 3, 4, 5)
```

The result is:

```
The sum of (2, 3, 4, 5) and 1 is 15.
```

That's not too pretty. We can improve it using a list comprehension, like this:

Demo 56: flow-control/Demos/add_nums.py

```
def add_nums(num, *nums):
    total = sum(nums, num)

    nums_joined = ', '.join([str(n) for n in nums])
    print(f"The sum of {nums_joined} and {num} is {total}.")

def main():
    add_nums(1, 2)
    add_nums(1, 2, 3, 4, 5)
    add_nums(11, 12, 13, 14)
    add_nums(101, 201, 301)

main()
```

Code Explanation

This uses a list comprehension to convert the list of numbers into a list of strings and then joins that list on ', ' in one step. The output will be:

```
The sum of 2 and 1 is 3.
The sum of 2, 3, 4, 5 and 1 is 15.
The sum of 12, 13, 14 and 11 is 50.
The sum of 201, 301 and 101 is 603.
```

Much prettier, right?

Creating Sublists with List Comprehensions

List comprehensions have an optional `if` clause that can be used to create a sublist from a list:

```
new_list = [item for item in items if some_condition]
```

Assume that you want to get all the three-letter words from a list of words. First, let's do it without a list comprehension:

```
three_letter_words = []  
for word in words:  
    if len(word) == 3:  
        three_letter_words.append(word)
```

In the following file, we do the same thing with a list comprehension:

Demo 57: flow-control/Demos/sublist_from_list.py

```
def main():
    words = ['Woodstock', 'Gary', 'Tucker', 'Gopher', 'Spike', 'I
             'Faline', 'Willy', 'Rex', 'Rhino', 'Roo', 'Littlefo
             'Bagheera', 'Remy', 'Pongo', 'Kaa', 'Rudolph', 'Ban
             'Courage', 'Nemo', 'Nala', 'Alvin', 'Sebastian', 'I
    three_letter_words = [w for w in words if len(w) == 3]
    print(three_letter_words)
```

```
main()
```

Code Explanation

This code combines an `if` condition within a `for` loop into a single line. It will return:

```
['Rex', 'Roo', 'Kaa']
```

Here's another example. Consider the following file:

Demo 58: flow-control/Demos/states_with_spaces.py

```
def main():
    with open('../data/states.txt') as f:
        lines = f.read().splitlines()

    states_with_spaces = []

    for line in lines:
        # Split the line into a 2-item list containing the
        # state name and abbreviation
        state = line.split('\t')

        # If the state name has a space, add it to states_with_spaces
        if ' ' in state[0]:
            states_with_spaces.append(state[0])

    # Print the states_with_spaces list
    for i, state_name in enumerate(states_with_spaces, 1):
        print(f'{i}. {state_name}')

main()
```

Code Explanation

This reads in the content of the states.txt file and then splits it into a list of lines. It then loops through that list to create a new list of just the states that have spaces in their names. It will output:

1. New Hampshire
2. New Jersey
3. New Mexico
4. New York
5. North Carolina
6. North Dakota
7. Rhode Island
8. South Carolina
9. South Dakota

10. West Virginia

You could replace the for loop with a couple of list comprehensions, like this:

Demo 59: flow-control/Demos/states_with_spaces2.py

```
-----Lines Omitted-----
```

```
    states = [line.split('\t')[0] for line in lines]
```

```
    states_with_spaces = [state for state in states if ' ' in state]
```

```
-----Lines Omitted-----
```

Code Explanation

1. The first list comprehension gets all the state names from the lines in the file.
 2. The second list comprehension then uses that list to get just the state names that have spaces in them.
-

This could also be done in a single step like this:

Demo 60: flow-control/Demos/states_with_spaces3.py

```
-----Lines Omitted-----
    states_with_spaces = [
        line.split('\t')[0] # Get state
        for line in lines # For each line
        if ' ' in line.split('\t')[0] # Where there is a space in
    ]
-----Lines Omitted-----
```

Code Explanation

While that's a bit harder to read, the code is more efficient, because it doesn't have to loop through `lines` and then loop again through `states`. It just does all the work the first time through the loop.

Conclusion

In this lesson, you have learned to write `if-elif-else` conditions and to loop through sequences. You also learned about the `enumerate()` function, generators, and list comprehensions.

LESSON 9

Exception Handling

Topics Covered

- Handling exceptions in Python.

Once married, it would be infinitely easier to ask her father's forgiveness, than to beg his permission beforehand.

– A Professional Rider, Mrs. Edward Kennard

By this time, you've probably run into some exceptions (errors) in your Python scripts. In this lesson, we will learn how to anticipate and handle exceptions gracefully.

Exception Basics

You may remember from school that you cannot divide by zero. Two or three people can share a pizza. One person can eat a whole pizza alone. But zero people cannot eat a pizza. The pizza will never get eaten. In Python, if you try to divide by zero, you will get a `ZeroDivisionError` exception:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In some cases, you may be fine with allowing the Python interpreter to report these exceptions as it finds them. But in other cases, you will want to anticipate and catch these exceptions and handle them in some special way. You do this in Python with `try/except` blocks. Here's a very simple example:

Demo 61: exception-handling/Demos/simple.py

```
try:
    1/0
except:
    print('You cannot divide by zero!')
```

Multiple `except` Clauses

You can have multiple `except` clauses with each clause specifying the type of exception to catch:

Demo 62: exception-handling/Demos/specific.py

```
def divide():
    try:
        numerator = int(input('Enter a numerator: '))
        denominator = int(input('Enter a denominator: '))
        result = numerator / denominator
        print(numerator, 'over', denominator, 'equals', result)
    except ValueError:
        print('Integers only please. Try again.')
        divide()
    except ZeroDivisionError:
        print('You cannot divide by zero! Try again.')
        divide()
    except KeyboardInterrupt:
        print('Quitter!')
    except:
        print('I have no idea what went wrong!')

def main():
    divide()

main()
```

Code Explanation

1. If the user enters anything other than an integer for the numerator or denominator, the `int()` function will fail and a `ValueError` exception will be thrown.
2. If the user enters `0` for the denominator, a `ZeroDivisionError` exception will be thrown.
3. If the user quits the program by pressing **CTRL+C**, a `KeyboardInterrupt` exception will be thrown.
4. The last `except` clause, which is optional, serves as a wildcard to catch any unspecified exception types.

Run the program and try entering different values to see how it works.

```
PS ...\\exception-handling\\Demos> python specific.py
Integers only please. Try again.
Enter a numerator: 10
Enter a denominator: 0
You cannot divide by zero! Try again.
Enter a numerator: 10
Enter a denominator: 2
10 over 2 equals 5.0
PS ...\\exception-handling\\Demos> python specific.py
Enter a numerator: 5
Enter a denominator: Quitter!
```

At the end, the user pressed **Ctrl+C** to end the program.

Wildcard except Clauses

Including a wildcard `except` clause can be dangerous because it can hide a real error. If you do use it, you may want to print a friendly error message and then reraise the error, like this:

Demo 63: exception-handling/Demos/reraise.py

```
try:
    1/0
except:
    print('Something really bad just happened! Oh no, oh no, oh no!')
    raise
```

Code Explanation

This first prints the friendly error (if you want it to be friendly) and then outputs the interpreter's traceback:

```
Something really bad just happened! Oh no, oh no, oh no!
Traceback (most recent call last):
  File ".\reraise.py", line 2, in <module>
    1/0
ZeroDivisionError: division by zero
```

Getting Information on Exceptions

To access the exception object, assign it to a variable using the `as` keyword, like this:

Demo 64: exception-handling/Demos/details.py

```
try:
    1/0
except Exception as e:
    print(type(e)) # prints exception type
    print(e) # prints friendly message
```

Code Explanation

This will output the following:

```
<class 'ZeroDivisionError'>
division by zero
```

Note that `e` is an arbitrary variable name. You can name the variable whatever you like.

Exercise 25: Raising Exceptions

15 to 25 minutes.

In this exercise, you will intentionally try to create different types of exceptions using the following template:

```
try:
    # code that creates exception
except Exception as e:
    print(type(e))
    print(e, '\n')
```

1. Open [exception-handling/Exercises/exception_details.py](#) in your editor.
2. Finish the `try/except` blocks to raise the following types of errors:
 - A. `ZeroDivisionError` – This one is done for you.

- B. ValueError
 - C. NameError
 - D. FileNotFoundError
 - E. ModuleNotFoundError
 - F. TypeError
 - G. AttributeError
 - H. StopIteration
 - I. KeyError
3. Use the documentation at <https://docs.python.org/3/library/exceptions.html#base-classes> as necessary.

Solution: exception-handling/Solutions/exception_details.py

```
# ZeroDivisionError
try:
    1/0
except Exception as e:
    print(type(e))
    print(e, '\n')

# ValueError
try:
    int('a')
except Exception as e:
    print(type(e))
    print(e, '\n')

# NameError
try:
    print(foo)
except Exception as e:
    print(type(e))
    print(e, '\n')

# FileNotFoundError
try:
    open('non-existing-file.txt', 'r')
except Exception as e:
    print(type(e))
    print(e, '\n')

# ImportError
try:
    import non_existing_module
except Exception as e:
    print(type(e))
    print(e, '\n')

# TypeError
try:
```

```

    nums = [1, 2] # Lists are iterables, not iterators
    next(nums)
except Exception as e:
    print(type(e))
    print(e, '\n')

# AttributeError
try:
    greeting = 'Hello'
    greeting.print() # strings don't have a print() method
except Exception as e:
    print(type(e))
    print(e, '\n')

# StopIteration
try:
    nums = [1, 2]
    iter_nums = iter(nums)
    print(next(iter_nums))
    print(next(iter_nums))
    print(next(iter_nums))
except Exception as e:
    print(type(e))
    print(e, '\n')

# KeyError
try:
    grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
    print(grades['Science'])
except Exception as e:
    print(type(e))
    print(e, '\n')

```

Code Explanation

The preceding code will render the following:

```
<class 'ZeroDivisionError'>
```


division by zero

```
<class 'ValueError'>
```

```
invalid literal for int() with base 10: 'a'
```

```
<class 'NameError'>
```

```
name 'foo' is not defined
```

```
<class 'FileNotFoundError'>
```

```
[Errno 2] No such file or directory: 'non-existing-file.txt'
```

```
<class 'ModuleNotFoundError'>
```

```
No module named 'non_existing_module'
```

```
<class 'TypeError'>
```

```
'list' object is not an iterator
```

```
<class 'AttributeError'>
```

```
'str' object has no attribute 'print'
```

```
1
```

```
2
```

```
<class 'StopIteration'>
```

```
<class 'KeyError'>
```

```
'Science'
```

The else Clause

You should limit your `try` clause to the code that might cause an exception. After the final `except` clause, you can include an `else` clause that contains code that only runs if no exception was raised in the `try` clause. Take a look at the following example:

Demo 65: exception-handling/Demos/else.py

```
def divide():
    try:
        numerator = int(input('Enter a numerator: '))
        denominator = int(input('Enter a denominator: '))
        result = numerator / denominator
    except ValueError:
        print('Integers only please. Try again.')
        divide()
    except ZeroDivisionError:
        print('You cannot divide by zero! Try again.')
        divide()
    except KeyboardInterrupt:
        print('Quitter!')
        raise
    except:
        print('I have no idea what went wrong!')
    else:
        print(numerator, 'over', denominator, 'equals', result)

def main():
    divide()

main()
```

Code Explanation

As we're fairly confident that the final `print()` function call will not raise a `ValueError` or a `ZeroDivisionError`, it makes sense to include it within the `else` clause.

The finally Clause

The `finally` clause is for doing any necessary cleanup (e.g., closing connections, releasing resources, etc.). You would generally only use it in a nested `try/except` block. Here is a pseudo-example:

```

try:
    resource = Resource() # Some generic resource
    resource.open() # Open resource

    # If the resource fails to open, execution will
    # jump to the outer except block, skipping the
    # whole try block below
    try:
        # Do something with resource
        resource.do_something()
    except:
        # Re-raise error to outer try block
        raise
    else:
        do_something_safe()
    finally:
        # Close resource. This will happen even if an
        # exception occurred in the inner except block above
        resource.close()
except Exception as e:
    # resource failed to open, so we don't need to close it
    print('Something bad happened:', e)
else:
    do_more_cool_stuff()

```

The crux of this code is that the resource should only be closed (and *always* be closed) if it was successfully opened, whether or not some exception occurred after it was opened. If the resource fails to open, then we should not try to close it, so `resource.close()` cannot go in a hypothetical `finally` block in the outer `try`. Note that this is an advanced scenario, so don't worry too much if you don't fully understand it. Generally, you won't have to use the `finally` block in exception handling.

Using Exceptions for Flow Control

In Python, it's not uncommon to use exception handling as a method of flow control. For example, rather than using a condition to check to

make sure everything is in order before proceeding, your code can simply try to proceed and then respond appropriately if an exception is raised. Compare the following `square_num()` function, which uses exception handling, with the `cube_num()` function, which uses an if condition to accomplish the same thing:

Demo 66: exception-handling/Demos/try_else.py

```
def square_num():
    try:
        num = int(input('Input Integer: '))
    except ValueError:
        print('That is not an integer.')
    else:
        print(num, 'squared is', num**2)

def cube_num():
    num = input('Input Number: ')
    if num.isdigit():
        print(num, 'cubed is', int(num)**3)
    else:
        print('That is not an integer.')
```

Code Explanation

The difference between using `if/else` and `try/except` for flow control is the difference between asking for permission and asking for forgiveness.

Exercise 26: Running Sum

10 to 20 minutes.

In this exercise, you will change a function that uses an `if-else` construct to use a `try-except` construct instead.

1. Open `exception-handling/Exercises/running_sum.py` in your editor and study the code. The code repeatedly prompts the user for a number, adding each valid entry to a total. If the user enters “q”, the program quits. If they enter any other non-integer value, the function prompts them for integers only. Run the script to see how it works.

2. Now, replace the `if-else` block with a `try-except` block. The script should continue to work in exactly the same way.

Exercise Code: exception-handling/Exercises/running_sum.py

```
def main():
    total = 0
    while True:
        num = input('Enter a number (q to quit): ')

        if num.lower() == 'q':
            print('Goodbye!')
            break

        if not num.isdigit():
            print('Integers only please. Try again.')
        else:
            total += int(num)
            print('The current total is:', total)

main()
```

Solution: exception-handling/Solutions/running_sum.py

```
def main():
    total = 0
    while True:
        try:
            num = input('Enter a number: ')
            if num.lower() == 'q':
                print('Goodbye!')
                break
            num = int(num) # This might cause an error
        except ValueError:
            print('Integers only please. Try again.')
        else:
            total += num
            print('The current total is:', total)

main()
```

Raising Your Own Exceptions

Sometimes it can be useful to raise your own exceptions, which you do with the `raise` statement. To illustrate, consider the following function, which creates a bi-directional dictionary from a dictionary:

```
def bidict(d):
    d2 = d.copy()
    for k,v in d.items():
        d2[v] = k
    return d2
```

If you pass `{'hola':'hi', 'adios':'bye'}` to this function, it will return this dictionary:

```
{'hola':'hi', 'adios':'bye', 'hi': 'hola', 'bye': 'adios'}
```

You can use this new dictionary to look up values in either direction. But what if you pass `bidict()` a dictionary that has two keys with the

same values, like this:

```
{'hola':'hi', 'adios':'bye', 'ciao':'bye'}
```

It won't error, but it won't return the mapping you want, because the `bye` key can only have one value:

```
{'hola':'hi', 'adios':'bye', 'ciao':'bye', 'hi': 'hola', 'bye':
```

The value of the `bye` key is `'adios'`, but there is no key with the value of `'ciao'`.

Consider the loop in the `bidict()` function. Each time it assigns a value to a key. But if that key already exists, it overwrites the existing value.

One way to handle this is to raise an exception, like this:

```
def bidict(d):
    d2 = d.copy()
    for k,v in d.items():
        if v in d2.keys():
            raise KeyError('Cannot create bidirectional dict ' +
                           'with duplicate keys.')
    d2[v] = k
    return d2
```

Now, when a programmer tries to pass a `dict` that won't work with the function, it will raise an exception. Here is the full code:

Demo 67: exception-handling/Demos/bidict_with_exception.py

```
def bidict(d):
    d2 = d.copy()
    for k, v in d.items():
        if v in d2.keys():
            raise KeyError('Cannot create bidirectional dict ' +
                           'with duplicate keys.')

    d2[v] = k
    return d2

translator = bidict({'hola':'hi', 'adios':'bye'})
print(translator['hola'])
print(translator['hi'])

translator2 = bidict({'hola':'hi', 'adios':'bye', 'ciao':'bye'})
```

Code Explanation

Run this and you will see that the call creating translator works, but the call creating translator2 does not:

```
hi
hola
Traceback (most recent call last):
File ".\bidict_with_exception.py", line 14, in <module>
translator2 = bidict({'hola':'hi', 'adios':'bye', 'ciao':'bye'})
File ".\bidict_with_exception.py", line 5, in bidict
raise KeyError('Cannot create bidirectional dict ' +
KeyError: 'Cannot create bidirectional dict with duplicate keys.'
```

Conclusion

In this lesson, you have learned to handle Python exceptions. For further study, see <https://docs.python.org/3/tutorial/errors.html>

LESSON 10

Python Dates and Times

Topics Covered

- The `time` module.
- The `datetime` module.

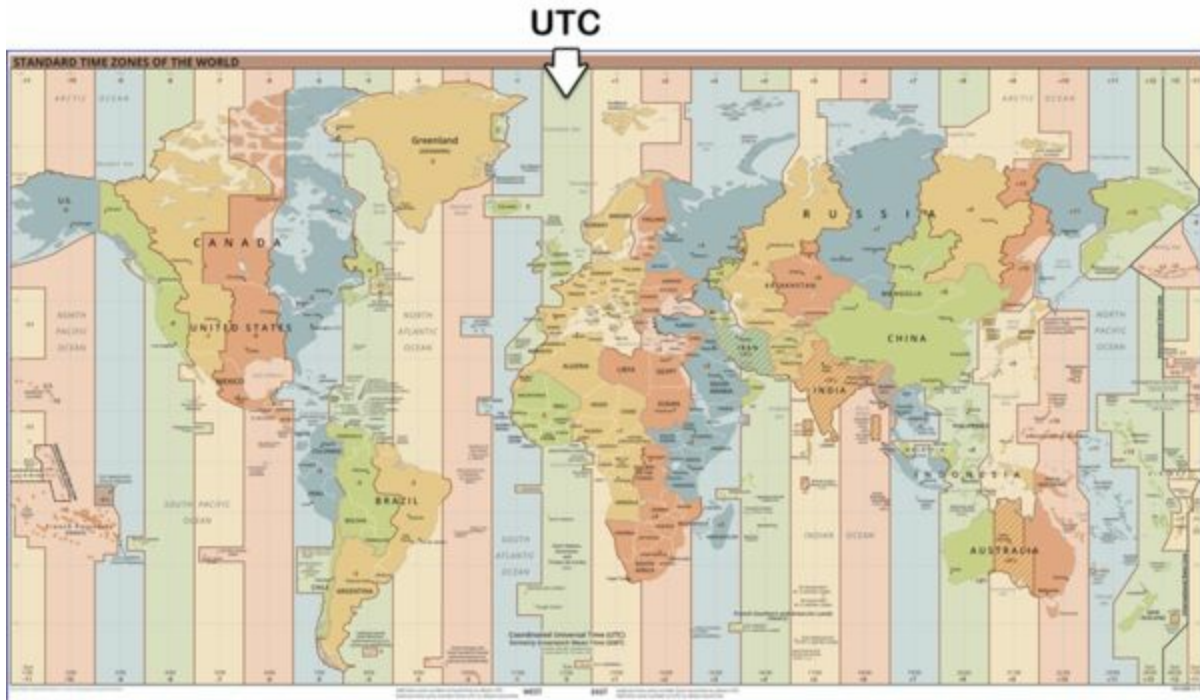
It was the best of times, it was the worst of times, ... it was the epoch of belief, it was the epoch of incredulity...

– A Tale of Two Cities, Charles Dickens

Working with dates and times can be tricky in programming. Python has some great built-in modules for helping with this. You'll learn about these modules in this lesson.

Understanding Time

Before we get into the Python code, it's important to understand a little about how computer languages, including Python, understand time. In particular, time is not the same across geography. The time in Moscow is different from the time in New York City. These differences wouldn't cause too much of a problem if they were consistent, but they are not, in large part because Daylight Saving Time (DST) is practiced in some parts of the world, but not in others, and it goes into effect on different dates and times. Even that wouldn't be too bad if there were some scientific way of determining where and when times were changed. But, alas, there is not. The decision to practice Daylight Saving Time is a political one, not a scientific one. Because of this, we cannot rely on local times to make exact calculations. **Coordinated Universal Time** (UTC) to the rescue. UTC, the successor to GMT (Greenwich Mean Time), is the standard by which we measure time. The following image shows world time zones as offsets:



World Time Zones on May 7, 2019 at 12:48³¹

Whether or not you need to be concerned with UTC time and time offsets will depend on the data with which you are working and the type of problem that you are trying to solve.

The Epoch

The **epoch** is the moment that a computer or computer language considers time to have started. Python considers the epoch to be January 1, 1970 at midnight (1970-01-01 00:00:00).³² Times before the epoch are expressed internally as negative numbers.

Python and Time

The most useful built-in modules for working with dates and times in Python are `time` and `datetime`.

The `time` Module³³.

The `time` module is useful for comparing moments in time, especially

for testing the performance of your code. It can also be used for accessing and manipulating dates and times.

Clocks

Python's `time` module includes these different types of clocks:

1. `time.monotonic()`
2. `time.perf_counter()`
3. `time.process_time()`
4. `time.time()`

Absolute Time

Of the clocks, only `time.time()` measures the *absolute time*: the number of seconds since the epoch, and can be converted to an actual time of day.

Demo 68: date-time/Demos/what_time_is_it.py

```
import time

seconds_since_epoch = time.time()
minutes_since_epoch = seconds_since_epoch / 60
hours_since_epoch = minutes_since_epoch / 60
days_since_epoch = hours_since_epoch / 24
years_since_epoch = days_since_epoch / 365.25

print("""s: {:,}
m: {:,}
h: {:,}
d: {:,}
y: {:,}""".format(seconds_since_epoch,
                    minutes_since_epoch,
                    hours_since_epoch,
                    days_since_epoch,
                    years_since_epoch, sep='\n'))
```

Code Explanation

The preceding code will render the following (on March 06, 2020 at 08:28:21):

```
s: 1,583,501,301.986899
m: 26,391,688.366448317
h: 439,861.4727741386
d: 18,327.56136558911
y: 50.17812831099003
```

Relative Time

The other clocks return relative times from an indeterminate start time. They are useful in determining differences between moments of time. The most useful of these will be `time.perf_counter()` as it provides the most precise results. It is often used to measure how quickly a

piece of code runs.

To illustrate how you would use this, assume that you wanted to create a list of random integers between 1 and 100 and that you needed the list as a string.

The most straightforward way of doing this is to create a loop and concatenate a new random number onto your string with each iteration.

But string concatenation is much less efficient than appending to a list, so it's actually faster to create a list, append a random number with each iteration, and then when the loop is complete join the list into a string.

And for a slightly faster solution, use a list comprehension. Here's the code:

Demo 69: date-time/Demos/compare_times.py

```
import time
import random

iterations = int(input('Number of iterations: '))

# Concatenating strings
start_time = time.perf_counter()
numbers = ''
for i in range(iterations):
    num = random.randint(1, 100)
    numbers += ',' + str(num)
end_time = time.perf_counter()
td1 = end_time - start_time

# Appending to a list
start_time = time.perf_counter()
numbers = []
for i in range(iterations):
    num = random.randint(1, 100)
    numbers.append(str(num))
numbers = ', '.join(numbers)
end_time = time.perf_counter()
td2 = end_time - start_time

# Using a list comprehension
start_time = time.perf_counter()
numbers = [str(random.randint(1, 100)) for i in range(1, iterations+1)]
numbers = ', '.join(numbers)
end_time = time.perf_counter()
td3 = end_time - start_time

print(f"""Number of numbers: {iterations},
Time Delta 1: {td1}
Time Delta 2: {td2}
Time Delta 3: {td3}
td1 is {round(td1/td3, 2)}x slower than td3.\n""")
```

Code Explanation

And here are the results using different values for integers:

```
PS ...\\date-time\\Demos> python compare_times.py
Number of iterations: 10000
Number of numbers: 10,000
Time Delta 1: 0.011690399999999999
Time Delta 2: 0.0099835000000000145
Time Delta 3: 0.009279500000000033
td1 is 1.26x slower than td3.
```

```
PS ...\\date-time\\Demos> python compare_times.py
Number of iterations: 1000000
Number of numbers: 1,000,000
Time Delta 1: 1.8037353
Time Delta 2: 1.0743994
Time Delta 3: 0.93997080000000002
td1 is 1.92x slower than td3.
```

```
PS ...\\date-time\\Demos> python compare_times.py
Number of iterations: 10000000
Number of numbers: 10,000,000
Time Delta 1: 60.204990500000001
Time Delta 2: 11.402855500000001
Time Delta 3: 9.9097729000000007
td1 is 6.08x slower than td3.
```

Notice that the speed differential increases markedly when dealing with large amounts of data.

The timeit Module

Python includes a `timeit` [34](#) module specifically for testing performance. We will cover `timeit` in the [Testing and Debugging lesson](#).

Time Structures

In addition to expressing time in seconds since the epoch, as `time.time()` does, time can be expressed as a special `time.struct_time` object, which is a type of tuple. The `time.struct_time` object for the epoch looks like this:

```
time.struct_time(tm_year=1970,
                 tm_mon=1, # 1 = January
                 tm_mday=1,
                 tm_hour=0,
                 tm_min=0,
                 tm_sec=0,
                 tm_wday=3, # 3 = Thursday (0 = Monday)
                 tm_yday=1, # 1 = First day of year
                 tm_isdst=0) # 0 = No daylight savings time
```

The same time expressed as a string looks like this:

```
'Thu Jan  1 00:00:00 1970'
```

Methods that create `time.struct_time` objects include `time.gmtime()` and `time.localtime()`.

```
time.gmtime([secs])
```

`time.gmtime([secs])` converts a time expressed in seconds since the epoch to a `struct_time` in *Coordinated Universal Time (UTC)*. If the `secs` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch. The value of `tm_isdst` will always be 0, meaning no daylight savings time.

Epoch as struct_time

```
>>> epoch = time.gmtime(0)
>>> epoch
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1,
                 tm_hour=0, tm_min=0, tm_sec=0,
```

```
tm_wday=3, tm_yday=1, tm_isdst=0)
```

Current Time as struct_time

```
>>> now = time.gmtime()
>>> now
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=20, tm_min=11, tm_sec=52,
                  tm_wday=4, tm_yday=66, tm_isdst=0)
```

Yesterday as struct_time

```
>>> day_in_seconds = 60*60*24
>>> yesterday = time.gmtime(time.time() - day_in_seconds)
>>> yesterday
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=5,
                  tm_hour=20, tm_min=12, tm_sec=2,
                  tm_wday=3, tm_yday=65, tm_isdst=0)
```

Tomorrow as struct_time

```
>>> day_in_seconds = 60*60*24
>>> tomorrow = time.gmtime(time.time() + day_in_seconds)
>>> tomorrow
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=7,
                  tm_hour=20, tm_min=12, tm_sec=5,
                  tm_wday=5, tm_yday=67, tm_isdst=0)
```

```
time.localtime([secs])
```

`time.localtime([secs])` is similar to `time.gmtime([secs])`, but it converts a time expressed in seconds since the epoch to a `struct_time` *in the local time*. As with `time.gmtime([secs])`, if the `secs` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch. The value of `tm_isdst` will be 0 or 1 depending on whether daylight savings time applies.

Epoch as Local struct_time

```
>>> epoch_offset = time.localtime(0) # "offset," because it's local
>>> epoch_offset
time.struct_time(tm_year=1969, tm_mon=12, tm_mday=31,
                  tm_hour=19, tm_min=0, tm_sec=0,
                  tm_wday=2, tm_yday=365, tm_isdst=0)
```

Current Time as Local struct_time

```
>>> now = time.localtime()
>>> now
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=16, tm_min=20, tm_sec=36,
                  tm_wday=4, tm_yday=66, tm_isdst=0)
```

Yesterday as Local struct_time

```
>>> day_in_seconds = 60*60*24
>>> yesterday = time.localtime(time.time() - day_in_seconds)
>>> yesterday
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=16, tm_min=20, tm_sec=36,
                  tm_wday=4, tm_yday=66, tm_isdst=0)
```

Tomorrow as Local struct_time

```
>>> tomorrow = time.localtime(time.time() + day_in_seconds)
>>> tomorrow
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=7,
                  tm_hour=16, tm_min=20, tm_sec=36,
                  tm_wday=5, tm_yday=67, tm_isdst=0)
```

To find out if daylight savings time currently applies in your timezone, run this:

```
time.localtime().tm_isdst
```

```
time.mktime()
```

The inverse of `time.localtime()` is `time.mktime()`, which takes a `struct_time` and returns the number of seconds since the epoch:

```
>>> lt = time.localtime()
>>> lt
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=14, tm_min=33, tm_sec=32,
                  tm_wday=4, tm_yday=66, tm_isdst=0)

>>> seconds_since_epoch = time.mktime(lt)
>>> seconds_since_epoch
1583523234.0
```

Times as Strings

When comparing or doing any sort of calculations with dates and times, it is necessary to treat them as objects or numbers; however, when outputting dates in reports, it is more useful to see their human-readable string representations.

Methods that create times as strings include :

```
time.asctime([t])
```

`time.asctime([t])` converts a time expressed as a `struct_time` to a date represented as a string. If the `struct_time` argument is omitted, it defaults to `time.localtime()`, which returns the current local time.

String Representation of Epoch

```
>>> time.asctime(time.gmtime(0))
'Thu Jan  1 00:00:00 1970'
```

String Representation of Current Local Time

```
>>> time.asctime()
'Fri Mar  6 09:41:24 2020'
```

```
time.ctime([secs])
```

`time.ctime([secs])` is like `time.asctime([t])`, but it takes a time expressed in seconds since the epoch instead of a `struct_time`. It returns a date in local time represented as a string. If the `secs` `struct_time` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch.

```
>>> str_epoch_offset = time.ctime(0) # "offset," because it's local
>>> str_epoch_offset
'Wed Dec 31 19:00:00 1969'

>>> str_now = time.ctime()
>>> str_now
'Fri Mar  6 09:57:17 2020'
```

Time and Formatted Strings

```
time.strftime(format[, t])
```

The `time.strftime(format[, t])` takes a `struct_time` object and returns a formatted string. The “f” in “strftime” is for “format.”

```
time.strptime(string[, format])
```

The `time.strptime(string[, format])` takes a string representing a time and returns a `struct_time` object. The “p” in “strptime” is for parse.”

Formatting Directives

Some of the more common formatting directives^{[35](#)} are shown in the following list:

- %a – Abbreviated weekday name.
- %A – Full weekday name.

- %b – Abbreviated month name.
- %B – Full month name.
- %d – Day of the month as a decimal number. Possible values: 01 - 31.
- %H – Hour (24-hour clock) as a decimal number. Possible values: 00 - 23.
- %I – Hour (12-hour clock) as a decimal number. Possible values: 01 - 12.
- %j – Day of the year as a decimal number. Possible values: 001 - 366.
- %m – Month as a decimal number. Possible values: 01 - 12.
- %M – Minute as a decimal number. Possible values: 00 - 59.
- %p – Locale's equivalent of either AM or PM.
- %S – Second as a decimal number. Possible values: 00 - 59.
- %w – Weekday as a decimal number. Possible values: 0 (Sunday) – 6.
- %x – Locale's appropriate date representation.
- %X – Locale's appropriate time representation.
- %c – Locale's appropriate date and time representation.
- %y – Year without century as a decimal number. Possible values: 00 - 99.
- %Y – Year with century as a decimal number.
- %Z – Time zone name (no characters if no time zone exists).

The following demo provides some examples of `time.strftime()` and `time.strptime()`.

Demo 70: date-time/Demos/formatting_times.py

```
import time

epoch = time.gmtime(0)
print(time.strftime('%c', epoch)) # 01/01/70 00:00:00
print(time.strftime('%x', epoch)) # 01/01/70
print(time.strftime('%X', epoch)) # 00:00:00
print(time.strftime('%A, %B %d, %Y, %I:%M %p', epoch))
# Thursday, January 01, 1970, 12:00 AM

independence_day = time.strptime('07/04/1776', '%m/%d/%Y')
print(independence_day)
```

Code Explanation

This will output:

```
Thu Jan  1 00:00:00 1970
01/01/70
00:00:00
Thursday, January 01, 1970, 12:00 AM
time.struct_time(tm_year=1776, tm_mon=7, tm_mday=4,
                  tm_hour=0, tm_min=0, tm_sec=0,
                  tm_wday=3, tm_yday=186, tm_isdst=-1)
```

Pausing Execution with `time.sleep()`

The `time.sleep(secs)` method suspends execution for a given number of seconds. Run the following demo to see how it works:

Demo 71: date-time/Demos/sleep.py

```
import time

for i in range(10, 0, -1):
    print(i)
    time.sleep(.5)

print('Blast off!')
```

Code Explanation

This will output the following, printing a new line every half second:

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```

`time.sleep()` can be used to create a digital clock:

Demo 72: date-time/Demos/clock.py

```
import time

def start_clock():
    print("Starting Clock")
    try:
        while True:
            localtime = time.localtime()
            result = time.strftime("%I:%M:%S %p", localtime)
            print(result)
            time.sleep(1)
    except KeyboardInterrupt:
        print("Stopping Clock")

start_clock()
```

Code Explanation

This will output something like the following, printing a new time every second until the user presses **Ctrl+C**:

```
Starting Clock
10:45:02 AM
10:45:03 AM
10:45:04 AM
10:45:05 AM
10:45:06 AM
Stopping Clock
```

The datetime Module^{[36](#)}

The datetime module includes the following types:

1. `datetime.date` – a date with year, month, and day attributes.
2. `datetime.time` – a time with hour, minute, second, microsecond, and `tzinfo` attributes.

3. `datetime.datetime` – a combination of `datetime.date` and `datetime.time`.
4. `datetime.timedelta` – a duration expressing the difference between instances of two `datetime.date`, `datetime.time`, or `datetime.datetime` objects.

`datetime.date` **Objects**

There are a number of `datetime` methods for creating `datetime.date` objects:

1. `datetime.date(year, month, day)` – Creates a local date.

```
>>> datetime.date(1776, 7, 4)
datetime.date(1776, 7, 4)
```

2. `datetime.date.today()` – Returns the current local date.

```
>>> datetime.date.today()
datetime.date(2020, 1, 31)
```

3. `datetime.date.fromtimestamp(secs)` – Returns the local date from `secs` seconds since the epoch:

```
>>> week_in_seconds = 24*60*60*7
>>> datetime.date.fromtimestamp(week_in_seconds) # One week a
datetime.date(1970, 1, 7)
```

`datetime.date` **Attributes**

A `datetime.date` instance includes the following attributes:

- `year`
- `month`

- day

```
>>> i_day = datetime.date(1776, 7, 4)
>>> i_day.year, i_day.month, i_day.day
(1776, 7, 4)
```

`datetime.date` **Methods**

A `datetime.date` instance includes the following methods:

- `replace()` – Returns a new `datetime.date` instance based on date with the given replacements.

```
>>> i_day = datetime.date(1776, 7, 4)
>>> i_day.replace(year=1826)
datetime.date(1826, 7, 4)
```

- `timetuple()` – Returns a `struct_time`.

```
>>> i_day.timetuple()
time.struct_time(tm_year=1776, tm_mon=7, tm_mday=4,
                  tm_hour=0, tm_min=0, tm_sec=0,
                  tm_wday=3, tm_yday=186, tm_isdst=-1)
```

- `weekday()` – Returns an integer representing the day of the week.

```
>>> i_day.weekday()
3
```

- `ctime()` – Returns a formatted date string. Similar to `time.ctime()`.

```
>>> i_day.ctime()
'Thu Jul  4 00:00:00 1776'
```

- `strftime()` – Returns a formatted date string. Similar to

`time.strftime(format[, t])` with the same formatting directives.

```
>>> i_day.strftime('%A, %B %d, %Y, %I:%M %p')
'Thursday, July 04, 1776, 12:00 AM'
```

`datetime.time` Objects

`datetime.time` objects are created with the `datetime.time()` method, which takes the following arguments:

- `hour` – defaults to 0.
- `minute` – defaults to 0.
- `second` – defaults to 0.
- `microsecond` – defaults to 0.
- `tzinfo` – defaults to `None`, which makes the `datetime` object “naive,” meaning that it is unaware of timezones. It is common to set `tzinfo` to UTC time like this
`tzinfo=datetime.timezone.utc.`

`datetime.time` Attributes

A `datetime.time` instance includes the following attributes:

- `hour`
- `minute`
- `second`
- `microsecond`

```
>>> t = datetime.time(hour=14, minute=13, second=12,
                      microsecond=11, tzinfo=datetime.timezone.utc)
>>> t.hour, t.minute, t.second, t.microsecond, t.tzinfo
(14, 13, 12, 11, datetime.timezone.utc)
```

`datetime.time` **Methods**

A `datetime.time` instance includes the following methods:

- `replace()` – Returns a new `datetime.time` instance based on time with the given replacements.

```
>>> t = datetime.time(hour=14, minute=13, second=12,
    microsecond=11, tzinfo=datetime.timezone.utc)
>>> t.replace(hour=4)
datetime.time(4, 13, 12, 11, tzinfo=datetime.timezone.utc)
```

- `strftime()` – Returns a formatted date string. Similar to `time.strftime(format[, t])` with the same formatting directives.

```
>>> t.strftime('%I:%M %p')
'02:13 PM'
```

`datetime.datetime` **Objects**

A `datetime.datetime` object is a combination of a `datetime.date` object and a `datetime.time` object. There are a number of `datetime` methods for creating `datetime.datetime` objects:

1. `datetime(year, month, day, hour, minute, second, microsecond, tzinfo)` – creates new `datetime`
2. `datetime.today()` – Returns the current local date and time.
3. `datetime.now()` – like `datetime.today()` but allows for time zone to be set.
4. `datetime.utcnow()` – Returns the current UTC date and time.
5. `datetime.fromtimestamp(timestamp)` – Returns the local date and time corresponding to timestamp.
6. `datetime.utcfromtimestamp(timestamp)` – Returns the UTC date and time corresponding to timestamp.
7. `datetime.combine(datetime.date, datetime.time)` – Combines a

`datetime.date` object and `datetime.time` object into a single `datetime.datetime` object.

8. `datetime.strptime(date_string, format)` – Takes a string representing a date and time and returns a `datetime.datetime` object.

`datetime.datetime` **Attributes**

A `datetime.datetime` instance includes the following attributes:

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`
- `microsecond`

```
>>> moon_landing = datetime.datetime(year=1969, month=7, day=21,
                                     hour=2, minute=56, second=15,
                                     tzinfo=datetime.timezone.utc)
>>> moon_landing.year, moon_landing.month, moon_landing.day
(1969, 7, 21)

>>> moon_landing.hour, moon_landing.minute, moon_landing.second
(2, 56, 15)

>>> moon_landing.microsecond, moon_landing.tzinfo
(0, datetime.timezone.utc)
```

`datetime.datetime` **Methods**

A `datetime.datetime` instance includes the following methods:

- `datetime.replace(year, month, day, hour, minute, second, microsecond)` – Returns a new `datetime.datetime` instance based on `datetime` with the given replacements.

```
>>> moon_landing.replace(year=2019)
datetime.datetime(2019, 7, 21, 2, 56, 15,
                  tzinfo=datetime.timezone.utc)
```

- `datetime.date()` – Returns a `datetime.date` object with same year, month, and day.

```
>>> moon_landing.date()
datetime.date(1969, 7, 21)
```

- `datetime.time()` – Returns a `datetime.time` object with same hour, minute, second, and microsecond.

```
>>> moon_landing.time()
datetime.time(2, 56, 15)
```

- `datetime.timetuple()` – Returns a `struct_time` representing the local time.

```
>>> moon_landing.timetuple()
time.struct_time(tm_year=1969, tm_mon=7, tm_mday=21,
                 tm_hour=2, tm_min=56, tm_sec=15,
                 tm_wday=0, tm_yday=202, tm_isdst=-1)
```

- `datetime.utctimetuple()` – Returns a `struct_time` representing the UTC time.

```
>>> moon_landing.utctimetuple()
time.struct_time(tm_year=1969, tm_mon=7, tm_mday=21,
                 tm_hour=2, tm_min=56, tm_sec=15,
                 tm_wday=0, tm_yday=202, tm_isdst=0)
```

- `datetime.timestamp()` – Returns a timestamp corresponding to

the `datetime`.

```
>>> moon_landing.timestamp()  
-14159025.0
```

- `datetime.weekday()` – Returns an integer representing the day of the week.

```
>>> moon_landing.weekday()  
0
```

- `datetime.ctime()` – Returns a formatted date string. Similar to `time.ctime()`.

```
>>> moon_landing.ctime()  
'Mon Jul 21 02:56:15 1969'
```

- `datetime.strftime(format)` – Returns a formatted date string. Similar to `time.strftime(format[, t])` with the same formatting directives.

```
>>> moon_landing.strftime('%A, %B %d, %Y, %I:%M %p')  
'Monday, July 21, 1969, 02:56 AM'
```

Exercise 27: What Color Pants Should I Wear?

15 to 30 minutes.

1. Create a new file called `pant_color.py` in the `date-time/Exercises` folder.
2. Write an `is_summer()` function that takes one argument: a `datetime.datetime` object that defaults to the current time. The function should return `True` if the date is between June 20 and September 22 of the year, and `False` otherwise. You will need to construct dates marking the start of summer and the start of fall.

To do so, you should make use of the year of the passed-in `datetime.datetime` object. For example, if the passed in date is in 2025, you must check if the date is between June 20, 2025 and September 22, 2025.

3. In the `main()` function, make a call to `is_summer()` and then print “You should wear white pants.” if it is summer or “You should wear black pants.” if it isn’t summer.
4. There are many ways to do this. Our solution makes use of the [ternary operator](#) in the `main()` function.

Solution: date-time/Solutions/pant_color.py

```
from datetime import datetime

def is_summer(the_date=datetime.now()):
    # Get the year of the passed-in datetime.datetime object
    year = the_date.year

    # Create datetime.datetime objects for starts of seasons
    summer_start = datetime(year, 6, 20)
    fall_start = datetime(year, 9, 22)

    # Return true if passed-in date is between starts of seasons
    return (summer_start < the_date < fall_start)

def main():
    # Use ternary operator to assign pant color
    pant_color = 'white' if is_summer() else 'black'
    print(f'You should wear {pant_color} pants.')

main()
```

datetime.timedelta Objects

A `datetime.timedelta` object expresses a duration – the time between two date, time, or datetime objects.

`datetime.timedelta` objects can be...

- Added ($t_1 + t_2$). **Result:** a new `datetime.timedelta` object.
- Subtracted ($t_1 - t_2$). **Result:** a new `datetime.timedelta` object.
- Divided (t_1 / t_2). **Result:** a float.
- Multiplied by an integer or float ($t_1 * 2$). **Result:** a new `datetime.timedelta` object.
- Divided by an integer or float ($t_1 / 2$). **Result:** a new `datetime.timedelta` object.

`datetime.timedelta` **Attributes**

A `datetime.timedelta` instance includes the following attributes:

- `days`
- `seconds`
- `microseconds`

```
>>> now = datetime.datetime.today()
>>> starttime = now.replace(hour=8, minute=30, second=0, microsecond=0)
>>> endtime = now.replace(hour=8, minute=48, second=23, microsecond=0)
>>> racetime = endtime - starttime
>>> racetime
datetime.timedelta(seconds=1103)
>>> racetime.days, racetime.seconds, racetime.microseconds
(0, 1103, 0)
```

`datetime.timedelta.total_seconds()` **Method**

A `datetime.timedelta` instance includes only one method: `total_seconds()`, which returns the total number of seconds in the duration (with microsecond accuracy).

```
>>> racetime.total_seconds()
1103.0
```

The Time Delta Between Dates

To determine the time delta between two dates, simply subtract one from the other:

```
>>> start = datetime.datetime(year=1861, month=4, day=12)
>>> end = datetime.datetime(year=1865, month=4, day=9)
>>> delta = end - start
>>> delta.days
1458
```

Exercise 28: Report on Departure Times

45 to 90 minutes.

In this exercise, you will create a small report on departure times from July, 1980. All the data is in a text file ([date-time/data/departure-data.txt](#)). Some of the data is shown below:

Exercise Code: date-time/data/departure-data.txt

```
*Scheduled      Actual
07/01/1980 2:40 AM      07/01/1980 4:00 AM
07/01/1980 3:00 AM      07/01/1980 3:00 AM
07/01/1980 4:40 AM      07/01/1980 4:40 AM
07/01/1980 5:30 AM      07/01/1980 5:30 AM
07/01/1980 6:00 AM      07/01/1980 6:01 AM
-----Lines Omitted-----
07/25/1980 7:00 PM      07/25/1980 7:09 PM
07/25/1980 7:01 PM      07/25/1980 7:01 PM
07/25/1980 7:15 PM
07/25/1980 7:53 PM      07/25/1980 7:53 PM
07/25/1980 8:00 PM      07/25/1980 8:00 PM
-----Lines Omitted-----
07/31/1980 9:10 PM      07/31/1980 9:10 PM
07/31/1980 10:05 PM     07/31/1980 10:05 PM
07/31/1980 10:45 PM     07/31/1980 10:45 PM
07/31/1980 11:15 PM     07/31/1980 11:15 PM
```

Things to note:

1. The first line is a header.
2. Each subsequent line has a tab separating a planned departure date and an actual departure date.
3. Some lines have a planned date, but no actual departure date. That means the trip was cancelled.

You should spend as much time as it takes to get this working or nearly working. Refrain from checking the solution until you have really given it your best shot. Learning to work through problems patiently and methodically, to read Python's error messages, and to debug your code are essential programming skills that are only attainable through practice.

The file you will be working on has been started already:

Exercise Code: date-time/Exercises/departure_report.py

```
from datetime import datetime

def get_departures():
    departures = []
    with open('../data/departure-data.txt') as f:
        for line in f.read().splitlines():
            departure = get_departure(line)
            if departure:
                departures.append(departure)
    return departures

def get_departure(line):
    """Return a tuple containing two datetime objects."""

    # If the line begins with an asterisk (*), return None

    # Get the planned and actual departures as strings by
    # splitting the line on a tab character into a list
    # Assign the first item in the list to planned and the
    # second item to actual

    # Convert the planned departure time to a datetime and
    # assign the result to date_planned

    # For those lines that have an actual departure time,
    # convert the actual departure time to a datetime and
    # assign the result to date_actual.
    # For lines that don't have an actual departure date, assign
    # None to date_actual.

    # Return a tuple with date_planned and date_actual.
    return (date_planned, date_actual)

def left_ontime(departure):
    planned = departure[0]
    actual = departure[1]
    if not actual:
```

```

        return False
    return actual == planned

# Write the following four functions. They should
# all return a boolean value
def left_early(departure):
    pass

def left_late(departure):
    pass

def left_next_day(departure):
    pass

def did_not_run(departure):
    pass

def main():
    departures = get_departures()
    ontime_departures = [d for d in departures if left_ontime(d)]
    early_departures = [d for d in departures if left_early(d)]
    late_departures = [d for d in departures if left_late(d)]
    next_day_departures = [d for d in departures if left_next_day(d)]
    cancelled_trips = [d for d in departures if did_not_run(d)]

    print(f"""Total Departures: {len(departures)}
    Ontime Departures: {len(ontime_departures)}
    Early Departures: {len(early_departures)}
    Late Departures: {len(late_departures)}
    Next Day Departures: {len(next_day_departures)}
    Cancelled Trips: {len(cancelled_trips)}""")

main()

```

The `main()` and `get_departures()` functions are complete. Your task is to complete the `get_departure()` function.

1. Open [date-time/Exercises/departure_report.py](#) in your editor.

2. Review the `main()` and `get_departures()` functions.
3. Your first job is to write the `get_departure()` function:
 - A. The first line is a header and begins with an asterisk (*). Return `None` for that line.
 - B. Each passed-in line is formatted as follows:

```
07/12/1980 7:53 PM\t07/12/1980 7:57 PM
```

The string before the `\t` represents the planned departure time. The string after the `\t` represents the actual departure time. Not all records have an actual departure time, indicating that the trip was cancelled. Those lines are formatted like this:

```
07/12/1980 7:53 PM\t
```

- C. Split the line on the tab into a two-element list.
 - D. Convert the planned departure time to a `datetime` and assign the result to `date_planned`.
 - E. For those lines that have an actual departure time, convert the actual departure time to a `datetime` and assign the result to `date_actual`. For lines that don't have an actual departure date, assign `None` to `date_actual`.
 - F. Return a tuple with `date_planned` and `date_actual`. This line is already written.
4. Study the `main()` function. See how it makes use of the `left_ontime()`, `left_early()`, `left_late()`, `left_next_day()`, and `did_not_run()` functions. The `left_ontime()` function is already written. Write the other four functions. They should all return a boolean value.

Solution: date-time/Solutions/departure_report.py

-----Lines Omitted-----

```
def get_departure(line):
    """Return a tuple containing two datetime objects."""

    if line[0] == '*':
        return None

    departure = line.split('\t')
    planned = departure[0]
    actual = departure[1]

    date_planned = datetime.strptime(planned, '%m/%d/%Y %I:%M %p')

    if actual:
        date_actual = datetime.strptime(actual, '%m/%d/%Y %I:%M %p')
    else: # Date doesn't exist
        date_actual = None

    return (date_planned, date_actual)

def left_ontime(departure):
    """Return True if left ontime. False, otherwise."""
    planned = departure[0]
    actual = departure[1]
    if not actual:
        return False
    return actual == planned

def left_early(departure):
    """Return True if left early. False, otherwise."""
    planned = departure[0]
    actual = departure[1]
    if not actual:
        return False
    if actual < planned:
        print('Early:', departure)
    return actual < planned
```

```
def left_late(departure):
    """Return True if left late. False, otherwise."""
    planned = departure[0]
    actual = departure[1]
    if not actual:
        return False
    return actual > planned

def left_next_day(departure):
    """Return True if departed next day. False, otherwise."""
    planned = departure[0]
    actual = departure[1]
    if not actual:
        return False
    return actual.day > planned.day

def did_not_run(departure):
    """Return True if did not depart. False, otherwise."""
    return not departure[1]
-----Lines Omitted-----
```

Code Explanation

The result of running this file is shown below:

```
Total Departures: 2481
Ontime Departures: 1207
Early Departures: 0
Late Departures: 1260
Next Day Departures: 2
Cancelled Trips: 14
```

If you were not able to get your program to work, study the solution and then go back and try to fix your code. If you're not able to fix it after studying it once, try again. That iterative process of attacking a

problem is excellent practice for real-world development.

Conclusion

In this lesson, you have learned to work with the `time` and `datetime` modules.

LESSON 11

File Processing

Topics Covered

- Reading files.
- Creating and writing to files.
- Working with directories.
- Working with the `os` and `os.path` modules.

He made haste to sit down in his easy chair and opened the book. He tried to read, but he could not revive the very vivid interest he had felt before in Egyptian hieroglyphics.

– Anna Karenina, Leo Tolstoy

Python allows you to access and modify files and directories on the operating system. Among other things, you can:

1. Open new or existing files and store them in *file object* variables.
2. Read file contents, all at once or line by line.
3. Append to file contents.
4. Overwrite file contents.
5. List directory contents.
6. Rename files and directories.

Opening Files

The built-in `open()` function will attempt to open a file at a given path and return a corresponding file object, which can be read and, if opened with the right permissions, written to. This is the most basic syntax for opening a file:

```
open(path_to_file, file_mode)
```

`path_to_file` can either be a relative or an absolute path. File modes are described in the following list:

1. "r" – Open for reading (default). Returns `FileNotFoundError` if the file doesn't exist.
2. "w" – Open for writing. If the file exists, existing content is erased. If the file doesn't exist, a new file is created.
3. "x" – Create and open for writing. Returns `FileExistsError` if the file already exists.
4. "a" – Open for appending to end of content. If the file doesn't exist, a new file is created.
5. "a+" – Open for appending to end of content and reading. If the file doesn't exist, a new file is created.
6. "r+" – Open for reading and writing.
7. "w+" – Open for writing and reading. If the file exists, existing content is erased. If the file doesn't exist, a new file is created.
8. "b" – Opens in binary mode.
9. "t" – Opens in text mode. This is the default.

```
open('poem.txt') # Open for reading
open('poem.txt', 'w') # Open for writing
open('poem.txt', 'r+') # Open for reading and writing
open('logo.png', 'r+b') # Open in binary mode for reading and wr.
```

Methods of File Objects

Reading Files

Once you have a file object, you can read the file contents using any of the following methods:

- `f.read(size)` – reads `size` bytes of the file. If `size` is omitted, the entire file is read.

- `f.readline()` – reads a single line up to and including the newline character (`\n`).
- `f.readlines()` – reads the file into a list split after the newline characters. Each line will end with a newline character.
- `f.read().splitlines()` – reads the file into a string and then splits on the newline characters. Lines will not end with newline characters.
- `list(f)` – does the same thing as `f.readlines()`.

Closing Files

You should always be sure to close files when you are done with them. This can be done with the `f.close()` method like this:

```
f = open('my_files/zen_of_python.txt')
# Do stuff with file
f.close()
```

While this works, it creates a potential problem. What if an exception occurs between the time the file is opened and closed? We will be left with a dangling reference to the file holding onto system resources and potentially blocking other applications from accessing the file.

[As we have seen](#), a better practice is to use the `with` keyword when working with files, like this:

```
with open(file) as f:
    do_stuff()
```

Using this structure, we don't have to explicitly close the file with `f.close()`. File objects have a special built-in `__exit__()` method that closes the file and is always called at the end of a `with` block even if code within the `with` block raises an exception. Often, you will read a file into a variable within a `with` block and then use the variable throughout the script, as shown in the following demo:

Demo 73: file-processing/Demos/with.py

```
with open('my_files/zen_of_python.txt') as f:
    poem = f.read()

# The file is now closed, but we saved its content in the poem variable
for i, line in enumerate(poem.splitlines(), 1):
    print(f"{i}. {line}")
```

Code Explanation

Run this file by opening the terminal at [file-processing/Demos/](#) and running:

```
python with.py
```

This will output:

```
1. The Zen of Python
2. Tim Peters
3. https://www.python.org/dev/peps/pep-0020/
4.
5. Beautiful is better than ugly.
6. Explicit is better than implicit.
7. Simple is better than complex.
8. Complex is better than complicated.
9. Flat is better than nested.
10. Sparse is better than dense.
11. Readability counts.
12. Special cases aren't special enough to break the rules.
13. Although practicality beats purity.
14. Errors should never pass silently.
15. Unless explicitly silenced.
16. In the face of ambiguity, refuse the temptation to guess.
17. There should be one-- and preferably only one --obvious way
18. Although that way may not be obvious at first unless you're
19. Now is better than never.
20. Although never is often better than *right* now.
```


- 21. If the implementation is hard to explain, it's a bad idea.
- 22. If the implementation is easy to explain, it may be a good idea.
- 23. Namespaces are one honking great idea -- let's do more of those!

Exercise 29: Finding Text in a File

15 to 25 minutes.

In this exercise, you will create a tool for searching through a file. This is the starting code:

Exercise Code: file-processing/Exercises/word_search.py

```
def search(word, text):
    """Return tuple holding line num and line text (or None)."""
    pass

def main():
    # Open my_files/zen_of_python.txt and
    # create a list from its contents
    # Save the list as zop

    word = input('Enter search word: ')
    result = search(word, zop)
    if result:
        print(word, ' first appears on line ',
              result[0], ': ', result[1], sep='')
    else:
        print(word, 'was not found.')

main()
```

1. Open file-processing/Exercises/word_search.py in your editor.
2. In the `main()` function, open my_files/zen_of_python.txt and create a list from its contents. Save the list as `zop`.
3. The rest of the `main()` function is already written. Read through the code to make sure you understand what it is doing.
4. Replace `pass` in the `search()` function with code that returns a two-element tuple containing the line number and line text in which the passed-in word is found. Return `None` if the word is not found. You may find it useful to review the `enumerate()` [function](#).
5. Run your solution by opening the terminal at file-processing/Exercises and running:

```
python word_search.py
```

Challenge

Modify the code so that it prints all the lines in which the word is found, like this:

```
Enter search word: than
than appears on line 5: Beautiful is better than ugly.
than appears on line 6: Explicit is better than implicit.
than appears on line 7: Simple is better than complex.
than appears on line 8: Complex is better than complicated.
than appears on line 9: Flat is better than nested.
than appears on line 10: Sparse is better than dense.
than appears on line 19: Now is better than never.
than appears on line 20: Although never is often better than *ri
```

Solution: file-processing/Solutions/word_search.py

```
def search(word, text):
    """Return tuple holding line num and line text."""
    for line in enumerate(text, 1):
        if line[1].find(word) >= 0:
            return line
    return None

def main():
    with open('my_files/zen_of_python.txt') as f:
        zop = f.readlines()

    word = input('Enter search word: ')
    result = search(word, zop)
    if result:
        print(word, ' first appears on line ',
              result[0], ': ', result[1], sep='')
    else:
        print(word, 'was not found.')

main()
```

Challenge Solution: file-processing/Solutions/word_search_challenge.py

```
def search(word, text):
    """Return tuple holding line num and line text."""
    results = []
    for line in enumerate(text, 1):
        if line[1].find(word) >= 0:
            results.append(line)
    return results

def main():
    with open('my_files/zen_of_python.txt') as f:
        zop = f.readlines()

    word = input('Enter search word: ')
    results = search(word, zop)
    if not results:
        print(word, 'was not found.')

    for result in results:
        print(word, ' appears on line ', result[0],
              ': ', result[1], sep='', end='')

main()
```

Writing to Files

Files opened with a mode that permits writing can be written to using the `write()` method as shown in the following file:

Demo 74: file-processing/Demos/simple_write.py

```
with open('my_files/yoda.txt', 'w') as f:  
    f.write('Powerful you have become.')
```

Code Explanation

1. Before running this file, look in the file-processing/Demos/my_files folder. It should **not** contain a yoda.txt file. If it does, delete it.
 2. Now, run the following from file-processing/Demos at the terminal:

```
python simple_write.py
```
 3. Now, look again in the file-processing/Demos/my_files folder. It should now contain a yoda.txt file. Open it in a text editor to see its contents.
-

Exercise 30: Writing to Files

5 to 10 minutes.

In this exercise, you will learn how modes affect reading and writing files. This exercise starts out like the previous demo.

1. Open file-processing/Exercises/simple_write.py in your editor.
2. Before running this file, look in the file-processing/Exercises/my_files folder. It should **not** contain a yoda.txt file. If it does, delete it.
3. Now, run the file from file-processing/Exercises at the terminal:

```
python simple_write.py
```
4. Now, look again in the file-processing/Exercises/my_files folder. The yoda.txt file should be there. Open it in a text editor to see its

contents.

5. In `simple_write.py`, change the string passed to the `write()` method to “Pass on what you have learned” or whatever you like.
6. Run the file again. It should completely overwrite the file with the new string.
7. Now, change the mode from “w” to “a” and run it again. This time, it should append to the file.
8. Try adding a call to `print(f.read())` on the line after the call to `f.write()` and run the file again. You should get an error. Review the modes at the [beginning of this lesson](#) to see if you can figure out why and how to fix it.

Challenge

The `seek()` method is used to change the cursor position in the `file` object. Its most common purpose is to return to the beginning of the file by passing it `0`. If you were able to fix the issue causing the error when you added the call to `f.read()`, you may have been surprised to find that the content you wrote to the file didn’t get read. This is because, after writing, the cursor is at the end of the file. Use the `seek()` method to fix this so that the contents of the file are printed out.

Solution: file-processing/Solutions/simple_write.py

```
with open('my_files/yoda.txt', 'a+') as f:  
    f.write('Powerful you have become.')  
    print(f.read())
```

Code Explanation

By changing the mode to “a+”, the file is opened for appending and reading. However, when a file is opened for appending, the cursor is placed at the end of the file, and when you read from the file, it begins where the cursor is, which is why `print(f.read())` doesn't output anything.

Challenge Solution: file-processing/Solutions/simple_write_challenge.py

```
with open('my_files/yoda.txt', 'a+') as f:
    f.write('Powerful you have become.')
    f.seek(0)
    print(f.read())
```

Code Explanation

Placing the cursor at the beginning of the file with `f.seek(0)` makes it possible to read the entire contents of the file.

Exercise 31: List Creator

30 to 45 minutes.

In this exercise, you will create a module that allows you to maintain a list of items in a file.

1. Open [file-processing/Exercises/list_creator.py](#) in your editor.
2. The `main()` and `show_instructions()` functions are already written. Your job is to write the following functions, which are documented in the file:
 - A. `add_item(item)`
 - B. `remove_item(item)` – This one is a little tricky. You will need to open the file twice, once for reading and once for writing. You may also find the [splitlines\(\) method](#) of a string object useful.
 - C. `delete_list()`
 - D. `print_list()`
3. Now, run the file from [file-processing/Exercises](#) at the terminal:

```
PS ...\file-processing\Exercises> python list_creator.py
```

This is the starting code:

```
def add_item(item):
    """Appends item (after stripping leading and trailing
    whitespace) to list.txt followed by newline character

    Keyword arguments:
    item -- the item to append"""
    pass

def remove_item(item):
    """Removes first instance of item from list.txt
    If item is not found in list.txt, alerts user.

    Keyword arguments:
    item -- the item to remove"""
    pass

def delete_list():
    """Deletes the entire contents of the list by opening
    list.txt for writing."""
    pass

def print_list():
    """Prints list"""
    pass

def show_instructions():
    """Prints instructions"""
    print("""OPTIONS:
P
    -- Print List
+abc
    -- Add 'abc' to list
-abc
    -- Remove 'abc' from list
--all
    -- Delete entire list
Q
```

```

        -- Quit\n"")

def main():
    show_instructions()

    while True:
        choice = input('>> ')

        if choice.lower() == 'q':
            print('Goodbye!')
            break

        if choice.lower() == 'p':
            print_list()
        elif choice.lower() == '--all':
            delete_list()
        elif choice and choice[0] == '+':
            add_item(choice[1:])
        elif choice and choice[0] == '-':
            remove_item(choice[1:])
        else:
            print("I didn't understand.")
            show_instructions()

if __name__ == '__main__':
    main()

```

Challenge

1. The `delete_list()` function currently deletes the list without warning. Give the user a chance to confirm before deleting the list.
2. In the `print_list()` function, use the `enumerate()` function to print the items as a numbered list.

Solution: file-processing/Solutions/list_creator.py

```
def add_item(item):
    """Appends item (after stripping leading and trailing
    whitespace) to list.txt followed by newline character

    Keyword arguments:
    item -- the item to append"""
    with open('list.txt', 'a') as f:
        f.write(item.strip() + '\n')

def remove_item(item):
    """Removes first instance of item from list.txt
    If item is not found in list.txt, alerts user.

    Keyword arguments:
    item -- the item to remove"""
    item_found = False
    with open('list.txt', 'r') as f:
        items = f.read().splitlines()
        if item in items:
            items.remove(item)
            item_found = True
        else:
            print("'" + item + "' not found in list.")

    if item_found:
        with open('list.txt', 'w') as f:
            f.write('\n'.join(items) + '\n')

def delete_list():
    """Deletes the entire contents of the list by opening
    list.txt for writing."""
    with open('list.txt', 'w') as f:
        print('The list has been deleted.')

def print_list():
    """Prints list"""
    with open('list.txt', 'r') as f:
```

```
print(f.read())
```

```
-----Lines Omitted-----
```

-----Lines Omitted-----

```
def delete_list():
    """After confirming user really wants to delete list,
    deletes the entire contents of the list by opening
    list.txt for writing."""
    confirm = input('Are you sure you want to delete the list? y/n')
    if confirm.lower() == 'y':
        with open('list.txt', 'w') as f:
            print('The list has been deleted.')
    else:
        print('OK. Whew! That was a close one.')

def print_list():
    """Prints list"""
    with open('list.txt', 'r') as f:
        for i, item in enumerate(f, 1):
            print(i, '. ' + item, sep='', end='')
    print()
```

-----Lines Omitted-----

The os Module

The `os` module is a built-in Python module for interacting with the operating system. In this section, we cover some of its most useful methods.

`os.getcwd()` and `os.chdir(path)`

`os.getcwd()` returns the current working directory as a string.

`os.chdir(path)` changes the current directory to `path`.

The following code shows how to get the current working directory with `os.getcwd()`, then change it using `os.chdir()`, and then get it again to prove that it changed:

```
>>> import os
```

```
>>> os.getcwd()
'C:\\Webucator\\Python\\file-processing\\Demos'

>>> os.chdir('..')
>>> os.getcwd()
'C:\\Webucator\\Python\\file-processing'
```

Notice the double backslash. That's because the string is escaped. Remember that the backslash is used to [escape characters](#). For a backslash to be just a backslash, it needs to be escaped by putting another backslash in front of it.

```
os.listdir(path)
```

Returns a list of files and subdirectories in `path`, which defaults to the current directory.

Demo 75: file-processing/Demos/directory_list.py

```
import os
dir_contents = os.listdir("my_files")
for item in dir_contents:
    print(item)
```

Code Explanation

This prints out a list of files and directories in the my_files directory:

```
PS ...\\file-processing\\Demos> python directory_list.py
a b c.txt
d e f.txt
foo.txt
g h i.txt
logo.png
zen_of_python.txt
```

`os.mkdir(dirname)` and `os.makedirs(path)`

`os.mkdir(dirname)` creates a directory named `dirname`. It errors if the directory already exists.

`os.makedirs(path)` is like `mkdir()`, but it creates all the necessary directories along the path.

Demo 76: file-processing/Demos/make_dirs.py

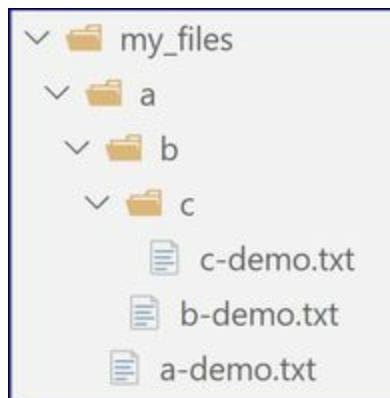
```
import os

os.mkdir("my_files/a")
os.makedirs("my_files/a/b/c")

with open('my_files/a/a-demo.txt', 'w') as f:
    f.write('Dummy text')
with open('my_files/a/b/b-demo.txt', 'w') as f:
    f.write('Dummy text')
```

Code Explanation

This creates directory a using `mkdir()` and then makes directories b and c using `makedirs()`. It then writes text files to the a and b directories. Run `make_dirs.py` from the `file-processing/Demos` directory and then look to see that the directories were created:



`os.rmdir(path)` and `os.removedirs(path)`

`os.rmdir(path)` deletes the directory at `path`. It errors if the directory is not empty or doesn't exist.

`os.removedirs(path)` is like `rmdir()`, but removes all empty directories in `path`, starting with the *leaf* (the last directory in the path). It will error if the leaf directory is not empty.

1. Open the Python terminal at [file-processing/Demos](#).
2. Run `import os`.
3. Run `os.rmdir('my_files/a/b/c')` to remove directory `c`.
4. Look in your file system to see that the directory was removed.
5. Run `os.removedirs('my_files/a/b')` to try to remove both the `a` and `b` directories. You will get an error because directory `b` is not empty.

`os.remove(path)` and `os.unlink(path)`

`os.remove(path)` and `os.unlink(path)` are identical. They delete the file at `path`, and they error if `path` doesn't point to a file.

1. Open the Python terminal at [file-processing/Demos](#) if it's not already open.
2. Run `import os`.
3. Run `os.listdir('my_files/a/b')` to show that directory `b` contains a file:

```
>>> import os
>>> os.listdir('my_files/a/b')
['b-demo.txt']
```

4. Run `os.remove('my_files/a/b/b-demo.txt')` to remove the file in directory `b`, and then run `os.listdir('my_files/a/b')` again to show that directory `b` is now empty:

```
>>> os.remove('my_files/a/b/b-demo.txt')
>>> os.listdir('my_files/a/b')
[]
```

5. Now, remove the file from directory `a`:

```
>>> os.remove('my_files/a/a-demo.txt')
```

6. Now that both directories a and b are empty, you can run `os.removedirs('my_files/a/b')` to remove both directories, and then use `os.listdir('my_files')` to check that they have been removed:

```
>>> os.remove('my_files/a/a-demo.txt')
>>> os.removedirs('my_files/a/b')
>>> os.listdir('my_files')
['a b c.txt', 'd e f.txt', 'foo.txt', 'g h i.txt', 'logo.png']
```

7. Close the Python terminal.

`os.rename(source, destination)` **and** `os.rename(oldpath, newpath)`

`os.rename(source, destination)` renames a file or a directory, changing the name from `source` to `destination`.

`os.rename(oldpath, newpath)` is like `rename(source, destination)`, but it starts by creating all necessary directories in `newpath` and ends by removing all empty directories in `oldpath`.

Demo 77: file-processing/Demos/renames.py

```
import os

foo = 'my_files/foo.txt'
bar = 'my_new_files/bar.txt'

def foo2bar():
    os.rename(foo, bar)
    print('Renamed', foo, 'to', bar)

def bar2foo():
    os.rename(bar, foo)
    print('Renamed', bar, 'to', foo)

foo2bar()
```

Code Explanation

1. Before running this file, look in the file-processing/Demos/my_files folder. It should contain a file named foo.txt.
2. Look in the file-processing/Demos folder. It should **not** contain a my_new_files folder. If it does, delete it.
3. Now, run renames.py from the file-processing/Demos directory:

```
PS ...\\file-processing\\Demos> python renames.py
Renamed my_files/foo.txt to my_new_files/bar.txt
```

4. Look again in the file-processing/Demos folder. A my_new_files folder should have appeared. Open it to find a bar.txt file, which is the foo.txt renamed and moved.
5. Look again in the file-processing/Demos/my_files folder. The foo.txt file should be gone. The my_files folder remains, because it has other things in it.
6. Open file-processing/Demos/renames.py in your editor. Change it so that it calls `bar2foo()` instead of `foo2bar()`. Run the file again.

The results:

- A. file-processing/Demos/my_files/foo.txt returns.
 - B. file-processing/Demos/my_new_files/bar.txt gets deleted.
 - C. file-processing/Demos/my_new_files also gets deleted as the only file it contained was bar.txt.
-

Walking a Directory

```
os.walk(top, topdown=True, onerror=None, followlinks=False)
```

The `os.walk()` method returns a generator by walking the directory tree and yielding a three-element tuple for each directory containing:

- `dirpath` – The path to the directory as a string.
- `dirnames` – A list of subdirectories in `dirpath`.
- `filenames` – A list of files in `dirpath`.

`os.walk()` Parameters

- `top` – The top-level directory for the walk.
- `topdown` – By default, `os.walk()` starts with `top` and works its way down; however, if `topdown` is `False`, it will work its way from bottom to top.
- `onerror` – By default, errors are ignored. If you want to handle errors in some way, set `onerror` to a function. When an error occurs, that function will be called and passed an `osError` instance.
- `followlinks` – By default, `os.walk()` ignores symbolic links (files that link to other directories). Set `followlinks` to `True` to include those linked directories. Be careful: you could end up with an infinite loop!

Let's take a look at an example. Review the following code to see if

you can understand what it does:

Demo 78: file-processing/Demos/walk.py

```
import os

def spaces2dashes():
    for dirpath, dirnames, filenames in os.walk('my_files'):
        for fname in filenames:
            if ' ' in fname:
                oldname = dirpath + '/' + fname
                newname = dirpath + '/' + fname.replace(' ', '-')
                print("Replacing", oldname, "with", newname)
                os.rename(oldname, newname)

def dashes2spaces():
    for dirpath, dirnames, filenames in os.walk('my_files'):
        for fname in filenames:
            if '-' in fname:
                oldname = dirpath + '/' + fname
                newname = dirpath + '/' + fname.replace('-', ' ')
                print("Replacing", oldname, "with", newname)
                os.rename(oldname, newname)

spaces2dashes()
```

Code Explanation

1. Before running this file, look in the [file-processing/Demos/my_files](#) folder. It should contain some files with spaces in their names (e.g., [a b c.txt](#)). That's ugly!
 2. Now, run [file-processing/Demos/walk.py](#).
 3. Now, look again in the [file-processing/Demos/my_files](#) folder. The spaces in those file names should have been replaced with dashes.
 4. If you like, edit the file so that it runs `dashes2spaces()` in place of `spaces2dashes()` and run it again to put the spaces back in the file names.
-

The `os.path` Module

The `os.path` module is used for performing functions on path names. In this section, we cover some of its most useful methods:

`os.path.abspath(path)`

`os.path.abspath(path)` returns the absolute path of `path`.

```
>>> os.path.abspath('.')  
'C:\\Webucator\\Python\\file-processing\\Demos'
```

`os.path.basename(path)`

`os.path.basename(path)` returns the basename of `path`.

```
>>> os.path.basename('my_files/logo.png')  
'logo.png'
```

`os.path.dirname(path)`

`os.path.dirname(path)` returns the path to the directory containing the leaf element of `path`.

```
>>> os.path.dirname('my_files/logo.png')  
'my_files'
```

`os.path.exists(path)`

`os.path.exists(path)` returns `True` if `path` exists.

```
>>> os.path.exists('my_files/logo.png')  
True
```

`os.path.getatime(path)`, `os.path.getmtime(path)`, and `os.path.getctime(path)`

`os.path.getatime(path)` returns the time `path` was last **accessed** in

number of seconds from the epoch.

`os.path.getmtime(path)` returns the time path was last **modified** in number of seconds from the epoch.

`os.path.getctime(path)` returns the time path was **created** in number of seconds from the epoch.

We can use the `datetime` module to convert and format these as readable dates and times:

```
>>> last_accessed = os.path.getatime('my_files/logo.png')
>>> last_modified = os.path.getmtime('my_files/logo.png')
>>> date_created = os.path.getctime('my_files/logo.png')
>>> last_accessed, last_modified, date_created
(1582050514.8227544, 1581523336.46663, 1580745606.86663)
>>> from datetime import datetime
>>> datetime.fromtimestamp(last_accessed).strftime('%c')
'Tue Feb 18 13:28:34 2020'
>>> datetime.fromtimestamp(last_modified).strftime('%c')
'Wed Feb 12 11:02:16 2020'
>>> datetime.fromtimestamp(date_created).strftime('%c')
'Mon Feb  3 11:00:06 2020'
```

`os.path.getsize(path)`

`os.path.getsize(path)` returns the size of the file at path.

```
>>> os.path.getsize('my_files/logo.png')
26216
```

`os.path.isabs(path)`

`os.path.isabs(path)` returns True if path is an absolute path.

```
>>> os.path.isabs('my_files/logo.png')
False
```

`os.path.relpath(path, start)`

`os.path.relpath(path, start)` returns a relative path to path from start, which defaults to the current directory.

```
>>> os.path.relpath(r'C:\Webucator\Python\file-processing\Demos\my_files\\logo.png')
```

If you are confused by the “r” prefix, review [Raw Strings](#).

`os.path.isfile(path)` and `os.path.isdir(path)`

`os.path.isfile(path)` returns True if path is a file.

```
>>> os.path.isfile('my_files/logo.png')
True
```

`os.path.isdir(path)` returns True if path is a directory.

```
>>> os.path.isdir('my_files/logo.png')
False
```

`os.path.join(path, *paths)`

`os.path.join(path, *paths)` returns a path by joining the passed-in paths intelligently. Note that different operating systems use different path structures. This will create the correct structure for the operating system it runs on.

```
>>> os.path.join('my_files', 'logo.png')
'my_files\\logo.png'
```

`os.path.split(path)`

Returns a 2-element tuple containing `os.path.dirname(path)` and `os.path.basename(path)`.

```
>>> os.path.split('my_files/logo.png')
('my_files', 'logo.png')
```

```
os.path.splitext(path)
```

Returns a 2-element tuple containing *the path minus the extension* and *the extension*.

```
>>> os.path.splitext('my_files/logo.png')  
('my_files/logo', '.png')
```

A Better Way to Open Files

Let's take another look at [Demos/with.py](#):

Demo 79: file-processing/Demos/with.py

```
with open('my_files/zen_of_python.txt') as f:
    poem = f.read()

# The file is now closed, but we saved its content in the poem variable
for i, line in enumerate(poem.splitlines(), 1):
    print(f"{i}. {line}")
```

We previously ran that file from the file-processing/Demos directory and it worked just fine, but try running it from a different directory:

1. Open file-processing in the Python terminal.
2. Run `python Demos/with.py`:

```
PS ...\\Python\\file-processing> python Demos/with.py
Traceback (most recent call last):
  File "Demos/with.py", line 1, in <module>
    with open('my_files/zen_of_python.txt') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'my_f
```

The issue is that Python is not looking for the file relative to where the Python file is; it is looking for it relative to where the Python script is being executed. One solution is to turn the relative path into an absolute path. To do that, we need the special `__file__` variable (that's two underscores on each side of "file").

`__file__` is a special variable that holds a relative path to the Python script from the present working directory. To see how it works, run file-processing/Demos/special_file_variable.py, which simply runs `print(__file__)`, from different directories:

```
PS ...\\file-processing\\Demos> python special_file_variable.py
special_file_variable.py

PS ...\\file-processing\\Demos> cd ..
PS ...\\Python\\file-processing> python Demos/special_file_variable.py
Demos/special_file_variable.py
```

```
PS ...\\Python\\file-processing> cd ..
PS ...\\Webucator\\Python> python file-processing/Demos/special_file_
file-processing/Demos/special_file_variable.py

PS ...\\Webucator\\Python> cd file-processing/Exercises
PS ...\\file-processing\\Exercises> python ../Demos/special_file_var.
../Demos/special_file_variable.py
```

We can take the absolute path to the folder holding the Python script and combine it with the parts of the relative path to the file we want to open to create an absolute path, like this:

```
relative_path = "my_files/zen_of_python.txt"

# Get absolute path to Python script
abs_path = os.path.abspath(__file__)

# Get absolute path to the directory that the script is in
folder = os.path.dirname(abs_path)

# Split the relative path on the forward slash
path_parts = relative_path.split('/')

# Join dir with path_parts to get an absolute path to the file to
new_path = os.path.join(folder, *path_parts)
```

The following script includes `print()` calls showing the values of the different variables:

Demo 80: file-processing/Demos/relpath_to_abspath.py

```
import os

relative_path = "my_files/zen_of_python.txt"
print("__file__:", __file__)

# Get absolute path to Python script
abs_path = os.path.abspath(__file__)
print("abs_path:", abs_path)

# Get absolute path to the directory that the script is in
folder = os.path.dirname(abs_path)
print("dir:", folder)

# Split the relative path on the forward slash
path_parts = relative_path.split("/")
print("path_parts:", path_parts)

# Join dir with path_parts to get an absolute path to the file to
new_path = os.path.join(folder, *path_parts)
print("new_path:", new_path)
```

Code Explanation

This will output something like:

```
PS ...\\file-processing\\Demos> python relpath_to_abspath.py
__file__: relpath_to_abspath.py
abs_path: C:\\Webucator\\Python\\file-processing\\Demos\\relpath_to_abspath.py
dir: C:\\Webucator\\Python\\file-processing\\Demos
path_parts: ['my_files', 'zen_of_python.txt']
new_path: C:\\Webucator\\Python\\file-processing\\Demos\\my_files\\zen_of_python.txt
```

For reuse purpose, we can create a function that takes a relative path to the file and returns an absolute path to the same file:

```
def file_path(relative_path):  
    folder = os.path.dirname(os.path.abspath(__file__))  
    path_parts = relative_path.split('/')  
    new_path = os.path.join(folder, *path_parts)  
    return new_path
```

Finally, we use that function in [file-processing/Demos/with2.py](#) so that we can run the script from anywhere:

Demo 81: file-processing/Demos/with2.py

```
import os
def file_path(relative_path):
    folder = os.path.dirname(os.path.abspath(__file__))
    path_parts = relative_path.split("/")
    new_path = os.path.join(folder, *path_parts)
    return new_path

path_to_file = file_path("my_files/zen_of_python.txt")
with open(path_to_file) as f:
    poem = f.read()

for i, line in enumerate(poem.splitlines(), 1):
    print(f"{i}. {line}")
```

Exercise 32: Comparing Lists

45 to 90 minutes.

An [article in the Atlantic](#)³⁷ claims that parents give girls boys names, but won't give boys girls names. Journalists need data to make this kind of claim. They rely on data analysts, many of whom are Python programmers like you, to analyze the data.

In the `file-processing/data` directory, there are files that contain lists of the most popular names in 1880 and 2018:

1. `1880-boys.txt`
2. `1880-girls.txt`
3. `2018-boys.txt`
4. `2018-girls.txt`

The data in these files is from <https://www.ssa.gov/oact/babynames/>.

Using the data in these files, try to answer these questions:

1. What names that were exclusively boys names in 1880 became exclusively girls names in 2018?
2. What names that were exclusively girls names in 1880 became exclusively boys names in 2018?

Does what you found support the journalist's claim?

Spend as much time as you like on this. It is an opportunity to practice your new Python skills.

Solution: file-processing/Solutions/boys_and_girls.py

```
import os

def file_path(relative_path):
    """Returns absolute path from relative path"""
    folder = os.path.dirname(os.path.abspath(__file__))
    path_parts = relative_path.split("/")
    new_path = os.path.join(folder, *path_parts)
    return new_path

def file_to_list(path):
    """Returns content of file at path as list"""
    with open(file_path(path)) as f:
        lines = f.read().splitlines()
    return lines

def subtract_lists(list1, list2):
    """Returns a list of all items in list1, but not in list2"""
    return [x for x in list1 if x not in list2]

def dups(list1, list2, sort=True):
    """Returns a list containing items in both lists"""
    dup_list = []
    for item in list1:
        if item in list2:
            dup_list.append(item)

    if sort:
        dup_list.sort()

    return dup_list

def list_to_file(path, the_list):
    """Creates/Overwrites file at path with content from the_list"""
    content = "\n".join(the_list)

    with open(file_path(path), "w") as f:
        f.write(content)
```

```

def main():
    boys_2018_path = "../data/2018-boys.txt"
    girls_2018_path = "../data/2018-girls.txt"
    boys_1880_path = "../data/1880-boys.txt"
    girls_1880_path = "../data/1880-girls.txt"

    boys_2018 = file_to_list(boys_2018_path)
    girls_2018 = file_to_list(girls_2018_path)
    boys_1880 = file_to_list(boys_1880_path)
    girls_1880 = file_to_list(girls_1880_path)

    boys_only_2018 = subtract_lists(boys_2018, girls_2018)
    girls_only_2018 = subtract_lists(girls_2018, boys_2018)
    boys_only_1880 = subtract_lists(boys_1880, girls_1880)
    girls_only_1880 = subtract_lists(girls_1880, boys_1880)

    boys_names_2_girls_names = dups(girls_only_2018, boys_only_1880)
    girls_names_2_boys_names = dups(boys_only_2018, girls_only_1880)

    list_to_file("../data/girls-names-that-were-boys-names.txt",
                 boys_names_2_girls_names)
    list_to_file("../data/boys-names-that-were-girls-names.txt",
                 girls_names_2_boys_names)

main()

```

Code Explanation

There are different ways of going about this. We went through the following process:

1. Create a `file_to_list()` function that takes a file path, reads it, and returns a list. We use this function to create four lists from the four files.
2. Find all the exclusively boys names and all the exclusively girls

names in 1880 and in 2018. We did this by creating a `subtract_lists()` function that gets all the items from one list that are not in another list.

3. Create a `dup()` function that takes two lists and returns a new list of all the items that are in both lists. We use this to compare exclusively girls names in 2018 with exclusively boys names in 1880. That tells us what names that used to be strictly boys names are now strictly girls names. We then do the same thing to compare exclusively boys names in 2018 with exclusively girls names in 1880.
4. We then write a `list_to_file()` function and use it to save the results in these new files:

A. girls-names-that-were-boys-names.txt – We found 16:

- i. Addison
- ii. Allison
- iii. Ashley
- iv. Aubrey
- v. Bailey
- vi. Charley
- vii. Dana
- viii. Holly
- ix. Ivory
- x. Kelly
- xi. Lindsey
- xii. Madison
- xiii. Monroe
- xiv. Palmer
- xv. Shelby
- xvi. Sydney

B. boys-names-that-were-girls-names.txt – We didn't find any.

These results appear to support the journalist's claim.

Conclusion

In this lesson, you have learned to work with files and directories on the operating system.

LESSON 12

PEP8 and Pylint

Topics Covered

- PEP8.
- Pylint.

[H]is personality has suggested to me an entirely new manner in art, an entirely new mode of style. I see things differently, I think of them differently. I can now recreate life in a way that was hidden from me before.

– The Picture of Dorian Gray, Oscar Wilde

PEP8³⁸ is the official style guide for Python code. Pylint is software that helps you follow the PEP8 guidelines and find potential problems with your code. In this lesson, you will learn about both.

PEP8

Here we provide a summary of PEP8's primary recommendations. We do not cover everything. We recommend that you read PEP8 yourself (see <https://www.python.org/dev/peps/pep-0008/>). Be aware that you may not understand everything as some of it deals with advanced Python functionality that we have not covered.

Maximum Line Length

Limit your lines to 79 characters whenever possible. For flowing lines (e.g., a long text string), limit lines to 72 characters.

Indentation

Use 4 spaces (not tabs) per indentation level.

Continuation Lines

When you must wrap a line of code across lines to adhere to the maximum line length, you should either:

1. Align the wrapped elements vertically:

Good

```
moonwalk = datetime.datetime(year=1969, month=7, day=21,  
                              hour=2, minute=56, second=15,  
                              tzinfo=datetime.timezone.utc)
```

2. Use a hanging indent:

Good

```
moonwalk = datetime.datetime(  
    year=1969, month=7, day=21,  
    hour=2, minute=56, second=15,  
    tzinfo=datetime.timezone.utc)
```

Note that, when using a hanging indent, no arguments should be on the first line:

Bad

```
moonwalk = datetime.datetime(year=1969, month=7, day=21,  
    hour=2, minute=56, second=15,  
    tzinfo=datetime.timezone.utc)
```

3. Long sequences that must wrap should be structured with the closing bracket aligned with the first line:

Good

```
fruit = [  
    "apple", "orange", "banana", "pear",  
    "lemon", "watermelon", "strawberry"  
]
```


Or with the closing bracket aligned with the last line:

Good

```
fruit = [  
    "apple", "orange", "banana", "pear",  
    "lemon", "watermelon", "strawberry"  
]
```

Blank Lines

1. Separate function definitions with two blank lines:[39](#)

Good

```
def do_this():  
    pass  
  
def do_that():  
    pass  
  
def do_this_other_thing():  
    pass  
  
# More Code
```

2. Blank lines can be used to separate logical sections within functions.

UTF-8 Encoding

Python files should be encoded using UTF-8. If you're using Visual Studio Code, the encoding is indicated in the bottom right:



Imports

1. All imports should be at the top of the file.
2. Import standard library imports before third-party imports.
3. Import third-party imports before local imports.
4. Import full packages before importing parts of packages:

Good

```
import math
from datetime import date, time
```

Bad

```
from datetime import date, time
import math
```

5. Imports should use separate lines:

Good

```
import math
import random
```

Bad

```
import math, random
```

Quotes

The only recommendations on single vs. double quotes are:

1. Use double quotes when string contains a single quote character (i.e., an apostrophe).
2. Use single quotes when string contains a double quote character.
3. Use double quote characters for triple-quoted strings.

4. Be consistent with yourself.

Whitespace in Expressions and Statements

Do not add whitespace...

1. Inside parentheses, brackets, or curly braces:

Good

```
if (a == b):
```

Bad

```
if ( a == b ):
```

Good

```
all([a, b, c, d])
```

Bad

```
all( [a, b, c, d] )
```

2. Before a comma:

Good

```
all([a, b, c, d])
```

Bad

```
all([a , b , c, d])
```

3. Before an open parentheses in a function call:

Good

```
print('Hello, world!')
```

Bad

```
print ('Hello, world!')
```

4. Around the equals (=) sign in keyword arguments:

Good

```
def dups(list1, list2, sort=True):
```

Bad

```
def dups(list1, list2, sort = True):
```

Surround assignment and comparison operators with single spaces:

Good

```
greeting = 'Hello'
```

Bad

```
greeting='Hello'
```

Good

```
a += 1
```

Bad

```
a+=1
```

Good

```
if (a == 1):
```

Bad

```
if (a==1):
```

Do not include extra whitespace at the end of a line or on blank lines.

Comments

Comments should use complete sentences and start with capital letters (unless they start with a keyword written in lowercase).

Additional Content in PEP8

We've covered what we feel are the most important PEP8 recommendations, but there are more. Again, we recommend that you read PEP8 and return to it from time to time, especially when you're unsure of the appropriate way to style a piece of code.

Pylint

Linters are tools that review your code and report stylistic or programming errors. The most popular Python linter is Pylint. To install Pylint, run the following command in the terminal:

```
pip install pylint
```

You can then use Pylint to analyze a file by running:

```
pylint path_to_file
```

For example, open the terminal at [date-time/Exercises](#) and run:

```
pylint departure_report.py
```

To save the results in a file, use:

```
pylint path_to_file > filename
```

For example:

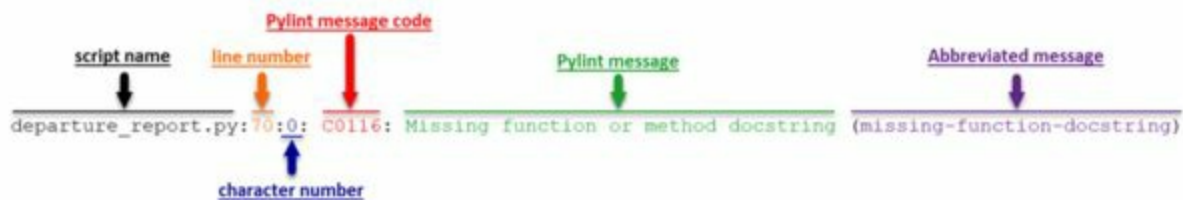
```
pylint departure_report.py > pylint.txt
```

When we run Pylint on departure_report.py in date-time/Solutions, we get this result:

```
***** Module departure_report
departure_report.py:85:0: C0304: Final newline missing (missing-
departure_report.py:1:0: C0114: Missing module docstring (missin
departure_report.py:6:47: C0103: Variable name "f" doesn't confo
departure_report.py:70:0: C0116: Missing function or method docs
```

Your code has been rated at 9.30/10 (previous run: 8.25/10, +1.0)

For each line that Pylint deems problematic, it provides the script name, the line number, the character number, the Pylint message code, the Pylint message, and the abbreviated message, as shown in the following image:



The report ends with a rating based on a scale of 10.

It is important to note that Pylint has opinions that go beyond PEP8, and that not all Python developers agree with all of its opinions. For example, we don't follow the convention of adding a newline at the end of every Python script. In our view, this was a historically useful practice for technical reasons that are no longer relevant with today's programming tools. We also use certain one-letter variable names, such as *i* for integer, *f* for file, and *k* and *v* for key and value. However, we only use them in small blocks of code.

Missing Docstrings

According to PEP8, every module and function should be commented. While we agree with this, we have not followed the practice in these lessons, because the accompanying readings explain how the modules and functions work. In many cases, the extra documentation would distract from the specific functionality being introduced.

`linter.py`

We have included a `linter.py` script in the root folder of your class files. This will run Pylint on all the Python files in a directory and its subdirectories. Run the file from the directory it is in like this:

```
python linter.py path_to_folder output_file
```

For example:

```
python linter.py date-time/Exercises > pylint.txt
```

After running this, you will find a `pylint.txt` file in the root directory containing a Pylint report on every Python file in the `date-time/Exercises` directory. You can then make fixes and run it again to generate a new report.

The `pylintrc` file, which is also in the root of your class files, has the settings used by `linter.py`. We have added a few modifications to the default settings:

1. Ignore the following rules:
 - `duplicate-code`
 - `missing-final-newline`
 - `missing-function-docstring`
 - `missing-module-docstring`
 - `invalid-name`

2. Allow variables named `foo` and `bar`. By default, Pylint disallows those variables and several others.

Feel free to explore the `pylintrc` file to see what other modifications you can make.

Conclusion

In this lesson, you have learned about the PEP8 coding standards and how to use Pylint to analyze your code.

LESSON 13

Advanced Python Concepts

Topics Covered

- Lambda functions.
- Advanced list comprehensions.
- The `collections` module.
- Mapping and filtering.
- Sorting sequences.
- Unpacking sequences in function calls.
- Modules and packages.

Packing the basket was not quite such pleasant work as unpacking the basket.

It never is.

– The Wind in the Willows, Kenneth Grahame

In this lesson, you will learn about some Python functionality and techniques that are commonly used but require a solid foundation in Python to understand.

Lambda Functions

Lambda functions are anonymous functions that are generally used to complete a small task, after which they are no longer needed. The syntax for creating a lambda function is:

```
lambda arguments: expression
```

Lambda functions are almost always used within other functions, but for demonstration purposes, we could assign a lambda function to a variable, like this:

```
f = lambda n: n**2
```

We could then call `f` like this:

```
f(5) # Returns 25  
f(2) # Returns 4
```

Try it at the Python terminal:

```
>>> f = lambda n: n**2  
>>> f(5)  
25  
>>> f(2)  
4
```

We will revisit lambda functions throughout this lesson.

Advanced List Comprehensions

Quick Review of Basic List Comprehensions

Before we get into advanced list comprehensions, let's do a quick review. The basic syntax for a list comprehension is:

```
my_list = [f(x) for x in iterable if condition]
```

In the preceding code `f(x)` could be any of the following:

1. Just a variable name (e.g., `x`).
2. An operation (e.g., `x**2`).
3. A function call (e.g., `len(x)` or `square(x)`).

Here is an example from earlier, in which we create a list by filtering another list:

Demo 82: advanced-python-concepts/Demos/sublist_from_list.py

```
def main():
    words = ['Woodstock', 'Gary', 'Tucker', 'Gopher', 'Spike', 'I
             'Faline', 'Willy', 'Rex', 'Rhino', 'Roo', 'Littlefo
             'Bagheera', 'Remy', 'Pongo', 'Kaa', 'Rudolph', 'Ban
             'Courage', 'Nemo', 'Nala', 'Alvin', 'Sebastian', 'I
    three_letter_words = [w for w in words if len(w) == 3]
    print(three_letter_words)

main()
```

Code Explanation

This will return:

```
['Rex', 'Roo', 'Kaa']
```

And here is a new example, in which we map all the elements in one list to another using a function:

Demo 83: advanced-python-concepts/Demos/list_comp_mapping.py

```
def get_inits(name):
    # Create list from first letter of each name part
    inits = [name_part[0] for name_part in name.split()]
    # Join inits list on "." and append "." to end
    return '.'.join(inits) + '.'

def main():
    people = ['George Washington', 'John Adams',
              'Thomas Jefferson', 'John Quincy Adams']

    # Create list by mapping person elements to get_inits()
    inits = [get_inits(person) for person in people]
    print(inits)

main()
```

Code Explanation

This will return:

```
['G.W.', 'J.A.', 'T.J.', 'J.Q.A.']
```

Now, on to the more advanced uses of list comprehension.

Multiple for Loops

Assume that you need to create a list of tuples showing all the possible *permutations* of rolling two six-sided dice. When dealing with permutations, order matters, so (1, 2) and (2, 1) are not the same. First, let's look at how we would do this without a list comprehension. We need to use a nested for loop:

Demo 84: advanced-python-concepts/Demos/dice_rolls.py

```
def main():
    dice_rolls = []
    for a in range(1, 7):
        for b in range(1, 7):
            roll = (a, b)
            dice_rolls.append(roll)

    print(dice_rolls)

main()
```

Code Explanation

This will return:

```
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6),
 (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6),
 (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6),
 (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6),
 (5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6),
 (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6)]
```

List comprehensions can include multiple `for` loops with each subsequent loop nested within the previous loop. This provides an easy way to create something similar to a two-dimensional array or a matrix:

Demo 85: advanced-python-concepts/Demos/dice_rolls_list_comp.py

```
def main():
    dice_rolls = [
        (a, b)
        for a in range(1, 7)
        for b in range(1, 7)
    ]

    print(dice_rolls)

main()
```

Code Explanation

This code will create the same list of tuples containing all the possible permutations of two dice rolls.

Notice that the list of permutations contains what game players would consider duplicates. For example, (1, 2) and (2, 1) are considered the same in dice. We can remove these pseudo-duplicates by starting the second `for` loop with the current value of `a` in the first `for` loop. Let's do this first without a list comprehension:

Demo 86: advanced-python-concepts/Demos/dice_combos.py

```
def main():
    dice_rolls = []
    for a in range(1, 7):
        for b in range(a, 7):
            roll = (a, b)
            dice_rolls.append(roll)

    print(dice_rolls)

main()
```

Code Explanation

The first time through the outer loop, the inner loop from 1 to 7 (not including 7), the second time through, it will loop from 2 to 7, then 3 to 7, and so on...

The `dice_rolls` list will now contain the different possible rolls (from a dice rolling point of view):

```
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6),
 (2, 2), (2, 3), (2, 4), (2, 5), (2, 6),
 (3, 3), (3, 4), (3, 5), (3, 6),
 (4, 4), (4, 5), (4, 6),
 (5, 5), (5, 6),
 (6, 6)]
```

Where we previously showed *permutations*, in which order matters, we are now showing *combinations*, in which order does not matter. The following two tuples represent different permutations, but the same combination: (1, 2) and (2, 1).

Now, let's see how we can do the same thing with a list comprehension:

Demo 87: advanced-python-concepts/Demos/dice_combos_list_comp.py

```
def main():
    dice_rolls = [
        (a, b)
        for a in range(1, 7)
        for b in range(a, 7)
    ]

    print(dice_rolls)

main()
```

Code Explanation

This code will create the same list of tuples containing all the possible combinations of two dice rolls.

Exercise 33: Rolling Five Dice

10 to 15 minutes.

There is no limit to the number of `for` loops in a list comprehension, so we can use this same technique to get the possibilities for more than two dice.

1. Create a new file in advanced-python-concepts/Exercises named list_comprehensions.py.
2. Write two separate list comprehensions:
 - A. The first should create five-item tuples for all unique **permutations** from rolling five identical six-sided dice. Remember, when looking for permutations, order matters.
 - B. The second should create five-item tuples for all unique **combinations** from rolling five identical six-sided dice.

Remember, when looking for combinations, order doesn't matter.

3. Print the length of each tuple.

Solution: advanced-python-concepts/Solutions/list_comprehensions.py

```
# Get unique permutations:
dice_rolls_p = [(a, b, c, d, e)
                 for a in range(1, 7)
                 for b in range(1, 7)
                 for c in range(1, 7)
                 for d in range(1, 7)
                 for e in range(1, 7)]

print('Number of permutations:', len(dice_rolls_p))

# Get unique combinations:
dice_rolls_c = [(a, b, c, d, e)
                 for a in range(1, 7)
                 for b in range(a, 7)
                 for c in range(b, 7)
                 for d in range(c, 7)
                 for e in range(d, 7)]

print('Number of combinations:', len(dice_rolls_c))
```

Code Explanation

This file will output:

```
Number of permutations: 7776
Number of combinations: 252
```

Collections Module

The `collections` module includes specialized containers (objects that hold data) that provide more specific functionality than Python's built-in containers (`list`, `tuple`, `dict`, and `set`). Some of the more useful containers are *named tuples* (created with the `namedtuple()` function), `defaultdict`, and `Counter`.

Named Tuples

Imagine you are creating a game in which you need to set and get the position of a target. You could do this with a regular tuple like this:

```
# Set target position:
target_pos = (100, 200)

# Get x value of target position
target_pos[0] # 100
```

But someone reading your code might not understand what `target_pos[0]` refers to.

A named tuple allows you to reference `target_pos.x`, which is more meaningful and helpful. Here is a simplified signature for creating `namedtuple` objects:

```
namedtuple(typename, field_names)
```

1. `typename` – The value passed in for `typename` will be the name of a new tuple subclass. It is standard for the name of the new subclass to begin with a capital letter. We have not yet covered classes and subclasses yet. For now, it is enough to know that the new tuple subclass created by `namedtuple()` will inherit all the properties of a tuple, and also make it possible to refer to elements of the tuple by name.
2. `field_names` – The value for `field_names` can either be a whitespace-delimited string (e.g., `'x y'`), a comma-delimited string (e.g., `'x, y'`), or a sequence of strings (e.g., `['x', 'y']`).

Demo 88: advanced-python-concepts/Demos/namedtuple.py

```
from collections import namedtuple

Point = namedtuple('Point', 'x, y')

# Set target position:
target_pos = Point(100, 200)

# Get x value of target position
print(target_pos.x)
```

Code Explanation

As the preceding code shows, the `namedtuple()` function allows you to give a name to the elements at different positions in a tuple and then refer to them by that name.

Default Dictionaries (defaultdict)

With regular dictionaries, trying to modify a key that doesn't exist will cause an exception. For example, the following code will result in a `KeyError`:

```
foo = {}
foo['bar'] += 1
```

A `defaultdict` is like a regular dictionary except that, when you look up a key that doesn't exist, it creates the key and assigns it the value returned by a function specified when creating it.

To illustrate how a `defaultdict` can be useful, let's see how we would create a *regular dictionary* that shows the number of different ways each number (2 through 12) can be rolled when rolling two dice, like this:

```
{
    2: 1,
    3: 2,
    4: 3,
    5: 4,
    6: 5,
    7: 6,
    8: 5,
    9: 4,
    10: 3,
    11: 2,
    12: 1
}
```

- There is only one way to roll a 2: (1, 1).
- There are two ways to roll a 3: (1, 2) and (2, 1).
- There are five ways to roll a 6: (1, 5), (2, 4), (3, 3), (4, 2), and (5, 1).

1. First, create the list of possibilities as we did earlier:

```
dice_rolls = [
    (a, b)
    for a in range(1, 7)
    for b in range(1, 7)
]
```

2. Next, create an empty dictionary, `roll_counts`, and then loop through the `dice_rolls` list checking for the existence of a key that is the sum of the dice roll. For example, on the first iteration, we find (1, 1), which when added together, gives us 2. Since `roll_counts` does not have a key 2, we need to add that key and set its value to 1. The same is true for when we find (1, 2), which adds up to 3. But later when we find (2, 1), which also adds up to 3, we don't need to recreate the key. Instead, we increment the existing key's value by 1. The code looks like this:

```

roll_counts = {}
for roll in dice_rolls:
    if sum(roll) in roll_counts:
        roll_counts[sum(roll)] += 1
    else:
        roll_counts[sum(roll)] = 1

```

This method works fine and gives us the following `roll_counts` dictionary that we looked at earlier:

```

{
    2: 1,
    3: 2,
    4: 3,
    5: 4,
    6: 5,
    7: 6,
    8: 5,
    9: 4,
    10: 3,
    11: 2,
    12: 1
}

```

An alternative to using conditionals to make sure the key exists is to just go ahead and try to increment the value of each potential key we find and then, if we get a `KeyError`, assign 1 for that key, like this:

```

roll_counts = {}
for roll in dice_rolls:
    try:
        roll_counts[sum(roll)] += 1
    except KeyError:
        roll_counts[sum(roll)] = 1

```

This also works and produced the same dictionary.

But with a `defaultdict`, we don't need the `if-else` block or the `try-`

except block. The code looks like this:

```
from collections import defaultdict

roll_counts = defaultdict(int)
for roll in dice_rolls:
    roll_counts[sum(roll)] += 1
```

The result is a defaultdict object that can be treated just like a normal dictionary:

```
defaultdict(<class 'int'>, {
    2: 1,
    3: 2,
    4: 3,
    5: 4,
    6: 5,
    7: 6,
    8: 5,
    9: 4,
    10: 3,
    11: 2,
    12: 1
})
```

Here are the three methods again:

Demo 89: advanced-python-concepts/Demos/dict_if_else.py

```
-----Lines Omitted-----  
roll_counts = {}  
for roll in dice_rolls:  
    if sum(roll) in roll_counts:  
        roll_counts[sum(roll)] += 1  
    else:  
        roll_counts[sum(roll)] = 1
```

Demo 90: advanced-python-concepts/Demos/dict_try_except.py

```
-----Lines Omitted-----  
roll_counts = {}  
for roll in dice_rolls:  
    try:  
        roll_counts[sum(roll)] += 1  
    except KeyError:  
        roll_counts[sum(roll)] = 1
```

```
from collections import defaultdict
-----Lines Omitted-----
roll_counts = defaultdict(int)
for roll in dice_rolls:
    roll_counts[sum(roll)] += 1
```

Notice in the preceding code that we passed `int` to `defaultdict()`:

```
roll_counts = defaultdict(int)
```

Remember, when you try to look up a key that doesn't exist in a `defaultdict`, it creates the key and assigns it the value returned by a function you specified when creating it. In this case, that function is `int()`.

When passing the function to `defaultdict()`, you do not include parentheses, because you are not calling the function at the time you pass it to `defaultdict()`. Rather, you are specifying that you want to use this function to give you default values for new keys. By passing `int`, we are stating that we want new keys to have a default value of whatever `int()` returns when no argument is passed to it. That value is 0:

```
>>> int()
0
```

You can create default dictionaries with any number of functions, both built-in and user-defined:

defaultdict with Built-in Functions

```
a = defaultdict(list) # Default key value will be []
b = defaultdict(str) # Default key value will be ''
```

defaultdict with lambda Function

```
c = defaultdict(lambda: 5) # Default key value will be 5
c['a'] += 1 # c['a'] will contain 6
```

defaultdict with User-defined Function

```
def foo():
    return 'bar'

d = defaultdict(foo) # Default key value will be 'bar'
d['a'] = d['a'].upper() # d['a'] will contain 'BAR'
```

Exercise 34: Creating a defaultdict

15 to 20 minutes.

In this exercise, you will organize the 1927 New York Yankees by position by creating a default dictionary that looks like this:

```
defaultdict(<class 'list'>,
{
    'OF': ['Earle Combs', 'Cedric Durst', 'Bob Meusel',
           'Ben Paschal', 'Babe Ruth'],
    'C': ['Benny Bengough', 'Pat Collins', 'Johnny Grabowski'],
    '2B': ['Tony Lazzeri', 'Ray Morehart'],
    'SS': ['Mark Koenig'],
    '3B': ['Joe Dugan', 'Mike Gazella', 'Julie Wera'],
    'P': ['Walter Beall', 'Joe Giard', 'Waite Hoyt',
           'Wilcy Moore', 'Herb Pennock', 'George Pipgras',
           'Dutch Ruether', 'Bob Shawkey', 'Urban Shocker',
           'Myles Thomas'],
    '1B': ['Lou Gehrig']
})
```

You will start with this list of dictionaries:

```
yankees_1927 = [
    {'position': 'P', 'name': 'Walter Beall'},
```

```
{'position': 'C', 'name': 'Benny Bengough'},
{'position': 'C', 'name': 'Pat Collins'},
{'position': 'OF', 'name': 'Earle Combs'},
{'position': '3B', 'name': 'Joe Dugan'},
{'position': 'OF', 'name': 'Cedric Durst'},
{'position': '3B', 'name': 'Mike Gazella'},
{'position': '1B', 'name': 'Lou Gehrig'},
{'position': 'P', 'name': 'Joe Giard'},
{'position': 'C', 'name': 'Johnny Grabowski'},
{'position': 'P', 'name': 'Waite Hoyt'},
{'position': 'SS', 'name': 'Mark Koenig'},
{'position': '2B', 'name': 'Tony Lazzeri'},
{'position': 'OF', 'name': 'Bob Meusel'},
{'position': 'P', 'name': 'Wilcy Moore'},
{'position': '2B', 'name': 'Ray Morehart'},
{'position': 'OF', 'name': 'Ben Paschal'},
{'position': 'P', 'name': 'Herb Pennock'},
{'position': 'P', 'name': 'George Pipgras'},
{'position': 'P', 'name': 'Dutch Ruether'},
{'position': 'OF', 'name': 'Babe Ruth'},
{'position': 'P', 'name': 'Bob Shawkey'},
{'position': 'P', 'name': 'Urban Shocker'},
{'position': 'P', 'name': 'Myles Thomas'},
{'position': '3B', 'name': 'Julie Wera'}
]
```

1. Open [advanced-python-concepts/Exercises/defaultdict.py](#) in your editor.
2. Write code so that the script creates the `defaultdict` above from the given list.
3. Output the pitchers stored in your new `defaultdict`.

```
from collections import defaultdict
```

```
yankees_1927 = [  
    {'position': 'P', 'name': 'Walter Beall'},  
    {'position': 'C', 'name': 'Benny Bengough'},  
    {'position': 'C', 'name': 'Pat Collins'},  
    {'position': 'OF', 'name': 'Earle Combs'},  
    {'position': '3B', 'name': 'Joe Dugan'},  
    {'position': 'OF', 'name': 'Cedric Durst'},  
    {'position': '3B', 'name': 'Mike Gazella'},  
    {'position': '1B', 'name': 'Lou Gehrig'},  
    {'position': 'P', 'name': 'Joe Giard'},  
    {'position': 'C', 'name': 'Johnny Grabowski'},  
    {'position': 'P', 'name': 'Waite Hoyt'},  
    {'position': 'SS', 'name': 'Mark Koenig'},  
    {'position': '2B', 'name': 'Tony Lazzeri'},  
    {'position': 'OF', 'name': 'Bob Meusel'},  
    {'position': 'P', 'name': 'Wilcy Moore'},  
    {'position': '2B', 'name': 'Ray Morehart'},  
    {'position': 'OF', 'name': 'Ben Paschal'},  
    {'position': 'P', 'name': 'Herb Pennock'},  
    {'position': 'P', 'name': 'George Pipgras'},  
    {'position': 'P', 'name': 'Dutch Ruether'},  
    {'position': 'OF', 'name': 'Babe Ruth'},  
    {'position': 'P', 'name': 'Bob Shawkey'},  
    {'position': 'P', 'name': 'Urban Shocker'},  
    {'position': 'P', 'name': 'Myles Thomas'},  
    {'position': '3B', 'name': 'Julie Wera'}  
]
```

```
# Each value will be a list of players, so we pass list to default dict  
positions = defaultdict(list)
```

```
# Loop through list of yankees appending player names to their position  
for player in yankees_1927:  
    positions[player['position']].append(player['name'])
```

```
print(positions['P'])
```

Code Explanation

This will output the list of pitchers:

```
[
    'Walter Beall',
    'Joe Giard',
    'Waite Hoyt',
    'Wilcy Moore',
    'Herb Pennock',
    'George Pipgras',
    'Dutch Ruether',
    'Bob Shawkey',
    'Urban Shocker',
    'Myles Thomas'
]
```

Add the following line of code to the `for` loop to watch as the players get added:

```
print(player['position'], positions[player['position']])
```

The beginning of the output will look like this:

```
PS ...\\advanced-python-concepts\\Solutions> python defaultdict.py
P ['Walter Beall']
C ['Benny Bengough']
C ['Benny Bengough', 'Pat Collins']
OF ['Earle Combs']
3B ['Joe Dugan']
OF ['Earle Combs', 'Cedric Durst']
3B ['Joe Dugan', 'Mike Gazella']
1B ['Lou Gehrig']
P ['Walter Beall', 'Joe Giard']
C ['Benny Bengough', 'Pat Collins', 'Johnny Grabowski']
...
```

Counters

Consider again the `defaultdict` object we created to get the number of different ways each number could be rolled when rolling two dice. This type of task is very common. You might have a collection of plants and want to get a count of the number of each species or the number of plants by color. The objects that hold these counts are called *counters*, and the `collections` module includes a special `Counter()` class for creating them.

Although there are different ways of creating counters, they are most often created with an iterable, like this:

```
from collections import Counter
c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green',
```

This will create the following counter:

```
Counter({
    'blue': 3,
    'green': 2,
    'red': 1,
    'yellow': 1
}))
```

To create a counter from the `dice_rolls` list we used earlier, we need to first create a list of sums from it, like this:

```
roll_sums = [sum(roll) for roll in dice_rolls]
```

`roll_sums` will contain the following list:

```
[
    2, 3, 4, 5, 6, 7,
    3, 4, 5, 6, 7, 8,
    4, 5, 6, 7, 8, 9,
    5, 6, 7, 8, 9, 10,
```

```
        6, 7, 8, 9, 10, 11,  
        7, 8, 9, 10, 11, 12  
    ]
```

We then create the counter like this:

```
c = Counter(roll_sums)
```

That creates a counter that is very similar to the `defaultdict` we saw earlier:

```
Counter({  
    7: 6,  
    6: 5,  
    8: 5,  
    5: 4,  
    9: 4,  
    4: 3,  
    10: 3,  
    3: 2,  
    11: 2,  
    2: 1,  
    12: 1  
})
```

The code in the following file creates and outputs the colors and dice rolls counters:

Demo 92: advanced-python-concepts/Demos/counter.py

```
from collections import Counter
c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
print('Colors Counter:', c, sep='\n', end='\n\n')

dice_rolls = [(a,b)
               for a in range(1,7)
               for b in range(1,7)]

roll_sums = [sum(roll) for roll in dice_rolls]
c = Counter(roll_sums)
print('Dice Roll Counter:', c, sep='\n')
```

Run this file. It will output:

```
Colors Counter:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})

Dice Roll Counter:
Counter({7: 6, 6: 5, 8: 5, 5: 4, 9: 4, 4: 3, 10: 3, 3: 2, 11: 2,
```

Updating Counters

Counter is a subclass of dict. We will learn more about subclasses later, but for now all you need to understand is that a subclass generally has access to all of its superclass's methods and data. So, Counter supports all the standard dict instance methods. The update() method behaves differently though. In standard dict objects, update() replaces key values with those of the passed-in dictionary:

Updating with a Dictionary

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
grades.update({'Math':97, 'Gym':93})
```

The grades dictionary will now contain:

```
{
    'English': 97,
    'Math': 97, # 97 replaces 93
    'Art': 74,
    'Music': 86,
    'Gym': 93 # Key is added with value of 93
}
```

In Counter objects, `update()` adds the values of a passed-in iterable or another Counter object to its own values:

Updating a Counter with a List

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green'])
>>> c # Before update:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})
>>> c.update(['red', 'yellow', 'yellow', 'purple'])
>>> c # After update:
Counter({'blue': 3, 'yellow': 3, 'green': 2, 'red': 2, 'purple': 1})
```

Updating a Counter with a Counter

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green'])
>>> d = Counter(['green', 'violet'])
>>> c.update(d)
>>> c
Counter({'green': 3, 'blue': 3, 'red': 1, 'yellow': 1, 'violet': 1})
```

Counters also have a corresponding `subtract()` method. It works just like `update()` but subtracts rather than adds the passed-in iterable counts:

Subtracting with a Counter

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green'])
>>> c # Before subtraction:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})
>>> c.subtract(['red', 'yellow', 'yellow', 'purple'])
```

```
>>> c # After subtraction:  
Counter({'blue': 3, 'green': 2, 'red': 0, 'yellow': -1, 'purple':
```

Notice that the value for the 'yellow' and 'purple' keys are negative, which is a little odd. We will learn how to create a non-negative counter in a [later lesson](#).

The `most_common([n])` Method

Counters include a `most_common([n])` method that returns the `n` most common elements and their counts, sorted from most to least common. If `n` is not passed in, all elements are returned.

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green'])  
>>> c.most_common()  
[('blue', 3), ('green', 2), ('red', 1), ('yellow', 1)]  
>>> c.most_common(2)  
[('blue', 3), ('green', 2)]
```

Exercise 35: Creating a Counter

10 to 15 minutes.

In this exercise, you will create a counter that holds the most common words used and the number of times they show up in the *U.S. Declaration of Independence*.

Declaration of Independence.

1. Create a new Python script in `advanced-python-concepts/Exercises` named `counter.py`.
2. Write code that:
 - A. Reads the `Declaration_of_Independence.txt` file in the same folder.
 - B. Creates a list of all the words that have at least six characters.
 - Use `split()` to split the text into words. This will split the text on whitespace. This isn't quite correct as it doesn't

account for punctuation, but for now, it's good enough.

- Use `upper()` to convert the words to uppercase.

C. Creates a counter from the word list.

D. Outputs the most common ten words and their counts. The result should look like this:

```
[
    ('PEOPLE', 13),
    ('STATES', 7),
    ('SHOULD', 5),
    ('INDEPENDENT', 5),
    ('AGAINST', 5),
    ('GOVERNMENT', 4),
    ('ASSENT', 4),
    ('OTHERS', 4),
    ('POLITICAL', 3),
    ('POWERS', 3)
]
```

Solution: `advanced-python-concepts/Solutions/counter.py`

```
from collections import Counter

with open('Declaration_of_Independence.txt') as f:
    doi = f.read()

word_list = [word for word in doi.upper().split() if len(word) > 3]

c = Counter(word_list)
print(c.most_common(10))
```

Mapping and Filtering

`map(function, iterable, ...)`

The built-in `map()` function is used to sequentially pass all the values of an iterable (or multiple iterables) to a function and return an iterator containing the returned values. It can be used as an alternative to list comprehensions. First, consider the following code sample that does not use `map()`:

Demo 93: advanced-python-concepts/Demos/without_map.py

```
def multiply(x, y):
    return x * y

def main():
    nums1 = range(0, 10)
    nums2 = range(10, 0, -1)

    multiples = []
    for i in range(len(nums1)):
        multiple = multiply(nums1[i], nums2[i])
        multiples.append(multiple)

    for multiple in multiples:
        print(multiple)

main()
```

Code Explanation

This code creates an iterator (a list) by multiplying 0 by 10, 1 by 9, 2 by 8,... 9 by 1, and 10 by 0. It then loops through the iterator printing each result. It will output:

```
0
9
16
21
24
25
24
21
16
9
```

The following code sample does the same thing using `map()`:

Demo 94: advanced-python-concepts/Demos/with_map.py

```
def multiply(x, y):
    return x * y

def main():
    nums1 = range(0, 10)
    nums2 = range(10, 0, -1)

    multiples = map(multiply, nums1, nums2)

    for multiple in multiples:
        print(multiple)

main()
```

Code Explanation

We could also include the `map()` function right in the `for` loop:

```
for multiple in map(multiply, nums1, nums2):
    print(multiple)
```

Note that you can accomplish the same thing with a list comprehension:

```
multiples = [multiply(nums1[i], nums2[i]) for i in range(len(nums1))]
```

One possible advantage of using `map()` in combination with multiple sequences is that it will not error if the sequences are different lengths. It will stop mapping when it reaches the end of the shortest sequence. In some cases, this might also be a disadvantage as it might hide a bug in the code. Also, this “feature” can be reproduced easily enough using the `min()` function:

```
[multiply(nums1[i], nums2[i]) for i in range(min(len(nums1), len(nums2)))]
```

`filter(function, iterable)`

The built-in `filter()` function is used to sequentially pass all the values of a single iterable to a function and return an iterator containing the values for which the function returns `True`. As with `map()`, `filter()` can be used as an alternative to list comprehensions. First, consider the following code sample that does not use `filter()`:

Demo 95: advanced-python-concepts/Demos/without_filter.py

```
def is_odd(num):  
    return num % 2  
  
def main():  
    nums = range(0, 10)  
  
    odd_nums = []  
    for num in nums:  
        if is_odd(num):  
            odd_nums.append(num)  
  
    for num in odd_nums:  
        print(num)  
  
main()
```

Code Explanation

This code passes a range of numbers one by one to the `is_odd()` function to create an iterator (a list) of odd numbers. It then loops through the iterator printing each result. It will output:

```
1  
3  
5  
7  
9
```

The following code sample does the same thing using `filter()`:

Demo 96: advanced-python-concepts/Demos/with_filter.py

```
def is_odd(num):
    return num % 2

def main():
    nums = range(0, 10)

    odd_nums = filter(is_odd, nums)

    for num in odd_nums:
        print(num)

main()
```

Code Explanation

As with `map()`, we can include the `filter()` function right in the `for` loop:

```
for num in filter(is_odd, nums):
    print(num)
```

Again, you can accomplish the same thing with a list comprehension:

```
odd_nums = [num for num in nums if is_odd(num)]
```

Using Lambda Functions with `map()` and `filter()`

The `map()` and `filter()` functions are both often used with lambda functions, like this:

```
>>> nums1 = range(0, 10)
>>> nums2 = range(10, 0, -1)
>>> for multiple in map(lambda n: nums1[n] * nums2[n], range(10))
...     print(multiple)
```

```
...
0
9
16
21
24
25
24
21
16
9

>>> for num in filter(lambda n: n % 2 == 1, range(10)):
...     print(num)
...
1
3
5
7
9
```

Let's just **keep** lambda

Some programmers, including Guido van Rossum, the creator of Python, dislike `lambda`, `filter()` and `map()`. These programmers feel that list comprehension can generally be used instead. However, other programmers love these functions and Guido eventually gave up the fight to remove them from Python. In February, 2006, [he wrote](#):⁴⁰

After so many attempts to come up with an alternative for lambda, perhaps we should admit defeat. I've not had the time to follow the most recent rounds, but I propose that we keep lambda, so as to stop wasting everybody's talent and time on an impossible quest.

You'll have to decide for yourself whether or not to use them.

Mutable and Immutable Built-in Objects

The difference between mutable objects, such as lists and dictionaries, and immutable objects, such as strings, integers, and tuples, may seem pretty straightforward: mutable objects can be changed; immutable objects cannot. But it helps to have a deeper understanding of how this can affect your code.

Consider the following code:

```
>>> v1 = 'A'
>>> v2 = 'A'
>>> v1 is v2
True
>>> list1 = ['A']
>>> list2 = ['A']
>>> list1 is list2
False
```

Immutable objects cannot be modified in place. Every time you “change” a string, you are actually creating a new string:

```
>>> name = 'Nat'
>>> id(name)
2613698710320
>>> name += 'haniel'
>>> id(name)
2613698710512
```

Notice the ids are different. It is impossible to modify an immutable object in place.

Lists, on the other hand, are mutable and can be modified in place. For example:

```
>>> v1 = [1, 2]
>>> v2 = v1
>>> id(v1) == id(v2)
True # Both variables point to the same list
```

```
>>> v1, v2
([1, 2], [1, 2])
>>> v2 += [3]
>>> v1, v2
([1, 2, 3], [1, 2, 3])
>>> id(v1) == id(v2)
True # Both variables still point to the same list
```

Notice that with lists, `v2` changes when we change `v1`. Both are pointing at the same list object, which is mutable. So, when we modify the `v2` list, we see the change in `v1`, because it points to the same object.

Be careful though. If you use the assignment operator, you will overwrite the old list and create a new list object:

```
>>> v1 = [1, 2]
>>> v2 = v1
>>> v1, v2
([1, 2], [1, 2])
>>> v1 = v1 + [3]
>>> v1, v2
([1, 2, 3], [1, 2])
```

Assigning `v1` explicitly with the assignment operator, rather than appending a value via `v1.append(3)`, results in a new object.

Sorting

Sorting Lists in Place

Python lists have a `sort()` method that sorts the list in place:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort()
```

The `colors` list will now contain:

```
['blue', 'green', 'orange', 'red']
```

The `sort()` method can take two keyword arguments: `key` and `reverse`.

reverse

The `reverse` argument is a boolean:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort(reverse=True)
```

The `colors` list will now contain:

```
['red', 'orange', 'green', 'blue']
```

key

The `key` argument takes a function to be called on each list item and performs the sort based on the result. For example, the following code will sort by word length:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort(key=len)
```

The `colors` list will now contain:

```
['red', 'blue', 'green', 'orange']
```

And the following code will sort by last name:

```
def get_lastname(name):
    return name.split()[-1]

people = ['George Washington', 'John Adams',
          'Thomas Jefferson', 'John Quincy Adams']
people.sort(key=get_lastname)
```

The people list will now contain:

```
[
    'John Adams',
    'John Quincy Adams',
    'Thomas Jefferson',
    'George Washington'
]
```

Note that John Quincy Adams shows up after John Adams in the result only because he shows up after him in the initial list. Our code as it stands does not take into account middle or first names.

Using Lambda Functions with key

If you don't want to create a new named function just to perform the sort, you can use a lambda function. For example, the following code would do the same thing as the code above without the need for the `get_lastname()` function:

```
people = ['George Washington', 'John Adams',,
          'Thomas Jefferson', 'John Quincy Adams']
people.sort(key=lambda name: name.split()[-1])
```

Combining key and reverse

The `key` and `reverse` arguments can be combined. For example, the following code will sort by word length in descending order:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort(key=len, reverse=True)
```

The colors list will now contain:

```
['orange', 'green', 'blue', 'red']
```

The sorted() Function

The built-in `sorted()` function requires an iterable as its first argument and can take `key` and `reverse` as keyword arguments. It works just like the list's `sort()` method except that:

1. It does **not** modify the iterable in place. Rather, it returns a new sorted list.
2. It can take any iterable, not just a list (but it always returns a list).

Exercise 36: Converting `list.sort()` to `sorted(iterable)`

15 to 25 minutes.

In this exercise, you will convert all the examples of `sort()` we saw earlier to use `sorted()` instead.

1. Open [advanced-python-concepts/Exercises/sorting.py](#) in your editor.
2. The code in the first example has already been converted to use `sorted()`.
3. Convert all other code examples in the script.


```
# Simple sort() method
colors = ['red', 'blue', 'green', 'orange']
# colors.sort()
new_colors = sorted(colors) # This one has been done for you
print(new_colors)

# The reverse argument:
colors.sort(reverse=True)
print(colors)

# The key argument:
colors.sort(key=len)
print(colors)

# The key argument with named function:
def get_lastname(name):
    return name.split()[-1]

people = ['George Washington', 'John Adams',
          'Thomas Jefferson', 'John Quincy Adams']
people.sort(key=get_lastname)
print(people)

# The key argument with lambda:
people.sort(key=lambda name: name.split()[-1])
print(people)

# Combing key and reverse
colors.sort(key=len, reverse=True)
print(colors)
```

Solution: `advanced-python-concepts/Solutions/sorting.py`

```
# Simple sort() method
colors = ['red', 'blue', 'green', 'orange']
# colors.sort()
new_colors = sorted(colors) # This one has been done for you
print(new_colors)

# The reverse argument:
# colors.sort(reverse=True)
# print(colors)
new_colors = sorted(colors, reverse=True)
print(new_colors)

# The key argument:
# colors.sort(key=len)
# print(colors)
new_colors = sorted(colors, key=len)
print(new_colors)

# The key argument with named function:
def get_lastname(name):
    return name.split()[-1]

people = ['George Washington', 'John Adams',
          'Thomas Jefferson', 'John Quincy Adams']
# people.sort(key=get_lastname)
# print(people)
new_people = sorted(people, key=get_lastname)
print(new_people)

# The key argument with lambda function:
people = ['George Washington', 'John Adams',
          'Thomas Jefferson', 'John Quincy Adams']
# people.sort(key=lambda name: name.split()[-1])
# print(people)
new_people = sorted(people, key=lambda name: name.split()[-1])
print(new_people)
```

```
# Combining key and reverse
# colors.sort(key=len, reverse=True)
# print(colors)
new_colors = sorted(colors, key=len, reverse=True)
print(new_colors)
```

Sorting Sequences of Sequences

When you sort a sequence of sequences, Python first sorts by the first element of each sequence, then by the second element, and so on. For example:

```
ww2_leaders = [
    ('Charles', 'de Gaulle'),
    ('Winston', 'Churchill'),
    ('Teddy', 'Roosevelt'), # Not a WW2 leader, but helps make p
    ('Franklin', 'Roosevelt'),
    ('Joseph', 'Stalin'),
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Hideki', 'Tojo')
]

ww2_leaders.sort()
```

The `ww2_leaders` list will be sorted by first name and then by last name. It will now contain:

```
[
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Charles', 'de Gaulle'),
    ('Franklin', 'Roosevelt'),
    ('Hideki', 'Tojo'),
    ('Joseph', 'Stalin'),
    ('Teddy', 'Roosevelt'),
    ('Winston', 'Churchill')
]
```

To change the order of the sort, use a lambda function:

```
ww2_leaders.sort(key=lambda leader: (leader[1], leader[0]))
```

The `ww2_leaders` list will now be sorted by last name and then by first name. It will now contain:

```
[
    ('Winston', 'Churchill'),
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Franklin', 'Roosevelt'),
    ('Teddy', 'Roosevelt'),
    ('Joseph', 'Stalin'),
    ('Hideki', 'Tojo'),
    ('Charles', 'de Gaulle')
]
```

It may seem strange that “de Gaulle” comes after “Tojo,” but that is correct. Lowercase letters come after uppercase letters in sorting. To change the result, you can use the `lower()` function:

```
ww2_leaders.sort(key=lambda leader: (leader[1].lower(), leader[0]
```

`ww2_leaders` will now contain:

```
[
    ('Winston', 'Churchill'),
    ('Charles', 'de Gaulle'),
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Franklin', 'Roosevelt'),
    ('Teddy', 'Roosevelt'),
    ('Joseph', 'Stalin'),
    ('Hideki', 'Tojo')
]
```

The preceding code can also be found in [advanced-python-](#)

[concepts/Demos/sequence_of_sequences.py](#).

Sorting Sequences of Dictionaries

You may often find data stored as lists of dictionaries:

```
from datetime import date
ww2_leaders = []
ww2_leaders.append(
    {'fname': 'Winston', 'lname': 'Churchill', 'dob': date(1889, 4, 21)}
)
ww2_leaders.append(
    {'fname': 'Charles', 'lname': 'de Gaulle', 'dob': date(1883, 7, 22)}
)
ww2_leaders.append(
    {'fname': 'Adolph', 'lname': 'Hitler', 'dob': date(1890, 11, 22)}
)
ww2_leaders.append(
    {'fname': 'Benito', 'lname': 'Mussolini', 'dob': date(1882, 1, 30)}
)
ww2_leaders.append(
    {'fname': 'Franklin', 'lname': 'Roosevelt', 'dob': date(1884, 12, 31)}
)
ww2_leaders.append(
    {'fname': 'Joseph', 'lname': 'Stalin', 'dob': date(1878, 12, 18)}
)
ww2_leaders.append(
    {'fname': 'Hideki', 'lname': 'Tojo', 'dob': date(1874, 11, 30)}
)
```

This data can be sorted using a lambda function similar to how we sorted lists of tuples:

```
ww2_leaders.sort(key=lambda leader: leader['dob'])
```

You can use this same technique to sort by a tuple:

```
ww2_leaders.sort(key=lambda leader: (leader['lname'], leader['fn
```

The preceding code can also be found in [advanced-python-concepts/Demos/sequence_of_dictionaries.py](#).

```
itemgetter()
```

While the method shown above works fine, the `operator` module provides an `itemgetter()` method that performs this same task a bit faster. It works like this:

```
from datetime import date
from operator import itemgetter

def main():
    -----Lines Omitted-----
    ww2_leaders.sort(key=itemgetter('dob'))
    print('First born:', ww2_leaders[0]['fname'])

    ww2_leaders.sort(key=itemgetter('lname', 'fname'))
    print('First in Encyclopedia:', ww2_leaders[0]['fname'])

main()
```

Creating a Dictionary from Two Sequences

Follow these steps to make a dictionary from two lists using the first list for keys and the second list for values:

1. Use the built-in `zip()` function to make a list of two-element tuples from the two lists:

```
>>> courses = ['English', 'Math', 'Art', 'Music']
>>> grades = [96, 99, 88, 94]
>>> z = zip(courses, grades)
```

This will result in a `zip` object.

2. Pass the `zip` object to the `dict()` constructor:

```
>>> course_grades = dict(z)
>>> course_grades
{'English': 96, 'Math': 99, 'Art': 88, 'Music': 94}
```

You can do the above in one step, like this:

```
course_grades = dict(zip(courses, grades))
```

This works with any type of sequence. For example, you could create a dictionary mapping letters to numbers like this:

```
>>> letter_mapping = dict(zip('abcdef', range(6)))
>>> letter_mapping
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5}
```

Unpacking Sequences in Function Calls

Sometimes you'll have a sequence that contains the exact arguments a function needs. To illustrate, consider the following function:

```
import math
def distance_from_origin(a, b):
    return math.sqrt(a**2 + b**2)
```

The function expects two arguments, *a* and *b*, which are the *x*, *y* coordinates of a point. It uses the Pythagorean theorem to determine the distance the point is from the origin.

We can call the function like this:

```
c = distance_from_origin(3, 4)
```

But it would be nice to be able to call the function like this too:

```
point = (3, 4)
c = distance_from_origin(point)
```

However, that will cause an error because the function expects two arguments and we're only passing in one.

One solution would be to pass the individual elements of our point:

```
point = (3, 4)
c = distance_from_origin(point[0], point[1])
```

But Python provides an even easier solution. We can use an asterisk in the function call to unpack the sequence into separate elements:


```
point = (3, 4)
c = distance_from_origin(*point)
```

When you pass a sequence preceded by an asterisk into a function, the sequence gets *unpacked*, meaning that the function receives the individual elements rather than the sequence itself.

The preceding code can also be found in [advanced-python-concepts/Demos/unpacking_function_arguments.py](#).

Exercise 37: Converting a String to a `datetime.date` Object

10 to 20 minutes.

In this exercise, you will convert a string representing a date to a `datetime.date` object. This is the starting code:

Exercise Code: [advanced-python-concepts/Exercises/converting_date_string_to_datetime.py](#)

```
import datetime

def str_to_date(str_date):
    # Write function
    pass

str_date = input('Input date as YYYY-MM-DD: ')
date = str_to_date(str_date)
print(date)
```

1. Open [advanced-python-concepts/Exercises/converting_date_string_to_datetime.py](#) in your editor.
2. The imported `datetime` module includes a `date()` method that can create a `date` object from three passed-in parameters: `year`, `month`, and `day`. For example:

```
datetime.date(1776, 7, 4)
```

3. Write the code for the `str_to_date()` function so that it...
 - A. Splits the passed-in string into a list of date parts. Each part should be an integer.
 - B. Returns a `date` object created by passing the unpacked list of date parts to `datetime.date()`.

Solution: `advanced-python-concepts/Solutions/converting_date_string_to_datetime.py`

```
import datetime

def str_to_date(str_date):
    date_parts = [int(i) for i in str_date.split("-")]
    return datetime.date(*date_parts)

str_date = input('Input date as YYYY-MM-DD: ')
date = str_to_date(str_date)
print(date)
```

Modules and Packages

You have worked with different Python modules (e.g., `random` and `math`) and packages (e.g., `collections`). In general, it's not all that important to know whether a library you want to use is a module or a package, but there is a difference, and when you're creating your own, it's important to understand that difference.

Modules

A module is a single file. It can be made up of any number of functions and classes. You can import the whole module using:

```
import module_name
```

Or you can import specific functions or classes from the module using:

```
from module_name import class_or_function_name
```

For example, if you want to use the `random()` function from the `random` module, you can do so by importing the whole module or by importing just the `random()` function:

```
>>> import random
>>> random.random()
```

```
0.8843194837647139
>>> from random import random
>>> random()
0.251050616400771
```

As shown above, when you import the whole module, you must prefix the module's functions with the module name.

Every .py file is a module. When you build a module with the intention of making it available to other modules for importing, it is common to include a `_test()` function that runs tests when the module is run directly. For example, if you run `random.py`, which is in the `Lib` directory of your Python home, the output will look something like this:

```
2000 times random
0.003 sec, avg 0.500716, stddev 0.285239, min 0.000495333, max 0

2000 times normalvariate
0.004 sec, avg 0.0061499, stddev 0.971102, min -2.86188, max 3.0

2000 times lognormvariate
0.004 sec, avg 1.64752, stddev 2.12612, min 0.0310675, max 28.51
...
```

Where is My Python Home?

To find your Python home, run the following code:

Demo 98: advanced-python-concepts/Demos/find_python_home.py

```
import sys
import os

python_home = os.path.dirname(sys.executable)
print(python_home)
```

Open random.py in an editor and you will see it ends with this code:

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

The `__name__` variable of any module that is imported holds that module's name. For example, if you import `random` and then print `random.__name__`, it will output "random". However, if you open random.py, add a line that reads `print(__name__)`, and run it, it will print "`__main__`". So, the `if` condition in the code above just checks to see if the file has been imported. If it hasn't (i.e., if it's running directly), then it will call the `_test()` function.

If you do not want to write tests, you could include code like this:

```
if __name__ == '__main__':
    print('This module is for importing, not for running directly')
```

Packages

A package is a group of files (and possibly subfolders) stored in a directory that includes a file named `__init__.py`. The `__init__.py` file does not need to contain any code. Some libraries' `__init__.py` files have a simple comment like this:

```
# Dummy file to make this a package.
```

However, you can include code in the `__init__.py` file that will initialize

the package. You can also (but do not have to) set a global `__all__` variable, which should contain a list of files to be imported when a file imports your package using `from package_name import *`. If you do not set the `__all__` variable, then that form of import will not be allowed, which may be just fine.

Search Path for Modules and Packages

The Python interpreter must locate the imported modules. When import is used within a script, the interpreter searches for the imported module in the following places sequentially:

1. The current directory (same directory as script doing the importing).
2. The library of standard modules.
3. The paths defined in `sys.path`.[41](#)

As you see, the steps involved in creating modules and packages for import are relatively straightforward. However, designing useful and easy-to-use modules and packages takes a lot of planning and thought.

Conclusion

In this lesson, you have learned several advanced techniques with sequences. You have also learned to do mapping and filtering, and to use lambda functions. Finally, you have learned how modules and packages are created.

LESSON 14

Regular Expressions

Topics Covered

- Understanding regular expressions.
- Python's `re` module.

Tom's whole class were of a pattern--restless, noisy, and troublesome. When they came to recite their lessons, not one of them knew his verses perfectly, but had to be prompted all along.

– The Adventures of Tom Sawyer, Mark Twain

Regular expressions are used to do pattern matching in many programming languages, including Java, PHP, JavaScript, C, C++, and Perl. We will provide a brief introduction to regular expressions and then we'll show you how to work with them in Python.

Regular Expression Tester

We will use the online regular expression testing tool at <https://pythex.org> to demonstrate and test our regular expressions. To see how it works, open the page in your browser:

The screenshot shows a web-based regular expression testing tool. It has three main sections, each with a numbered callout:

- 1** **Your regular expression:** A text input field containing the string `rose`.
- 2** **Your test string:** A larger text area containing the string `A rose is a rose is a rose.`. To the right of this field are two buttons: `IGNORECASE` (disabled) and `MULTILINE` (active/highlighted).
- 3** **Match result:** A text area showing the result of the search. The string `A rose is a rose is a rose.` is displayed, with the words `rose` highlighted in green.

As shown in the screenshot:

1. Enter “rose” in the **Your regular expression** field.
2. Enter “A rose is a rose is a rose.” in the **Your test string** field.
3. Notice the **Match result**. The parts of the string that match your pattern will be highlighted.

Usually, you will want to have the **MULTILINE** option selected so that each line will be tested individually.

In the **Your test string** field, you can test multiple strings:

Your regular expression:

foo

IGNORECASE

Your test string:

```
food
Bigfoot
foo foo
barefoot
footwork
```

Match result:

```
food
Bigfoot
foo foo
barefoot
footwork
```

These examples just find occurrences of a substring (e.g., “rose”) in a string (e.g., “A rose is a rose is a rose.”). But the power of regular expressions is in pattern matching. The best way to get a feel for them is to try them out. So, let’s do that.

Regular Expression Syntax

Here we’ll show the different symbols used in regular expressions. You should use <https://pythex.org> to test the patterns we show.

Start and End (^ \$)

A caret (^) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern `^dog` can be found in “**dog**fish”, but not in “bulldog” or “boondoggle”.

A dollar sign (`$`) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern `dog$` can be found in “bulld**og**”, but not in “dogfish” or “boondoggle”.

Word Boundaries (`\b \B`)

Backslash-b (`\b`) denotes a word boundary. It matches a location at the beginning or end of a word. A word is a sequence of numbers, letters, and underscores. Any other character is considered a word boundary.

- The pattern `dog\b` matches the first but not the second occurrence of “dog” in the phrase “The bulld**og** bit the dogfish.”
- In the phrase “The dogfish bit the bulld**og**.”, it only matches the second occurrence of “dog”, because a period is a word boundary.

Backslash-B (`\B`) is the opposite of backslash-b (`\b`). It matches a location that is **not** a word boundary.

- The pattern `dog\B` matches the second but not the first occurrence of “dog” in the phrase “The bulldog bit the **dog**fish.”
- But in the phrase “The **dog**fish bit the bulldog.”, it only matches the first occurrence of “dog”.

Number of Occurrences (`? + * {}`)

The following symbols affect the number of occurrences of the preceding character: `?`, `+`, `*`, and `{}`.

A question mark (`?`) indicates that the preceding character should

appear zero or one times in the pattern.

- The pattern `go?ad` can be found in “**goad**” and “**gad**”, but not in “**gooad**”. Only zero or one “o” is allowed before the “a”.

A plus sign (`+`) indicates that the preceding character should appear one or more times in the pattern.

- The pattern `go+ad` can be found in “**goad**”, “**gooad**” and “**gooodad**”, but not in “**gad**”.

An asterisk (`*`) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern `go*ad` can be found in “**gad**”, “**goad**”, “**gooad**” and “**gooodad**”.

Curly braces with one parameter (`{n}`) indicate that the preceding character should appear exactly *n* times in the pattern.

- The pattern `fo{3}d` can be found in “**foood**” , but not in “**food**” or “**fooodd**”.

Curly braces with two parameters (`{n1, n2}`) indicate that the preceding character should appear between *n1* and *n2* times in the pattern.

- The pattern `fo{2, 4}d` can be found in “**food**”, “**foood**” and “**fooodd**”, but not in “**fod**” or “**fooooood**”.

Curly braces with one parameter and an empty second parameter (`{n, }`) indicate that the preceding character should appear at least *n* times in the pattern.

- The pattern `fo{2, }d` can be found in “**food**” and “**fooooood**”, but not in “**fod**”.

Common Characters (. \d \D \w \W \s \S)

A period (.) represents any character except a newline.

- The pattern `fo.d` can be found in “food”, “foad”, “fo9d”, and “fo d”.

Backslash-d (`\d`) represents any digit. It is the equivalent of `[0-9]` (to be discussed soon).

- The pattern `fo\dd` can be found in “fo1d”, “fo4d” and “fo0d”, but not in “food” or “fodd”.

Backslash-D (`\D`) represents any character except a digit. It is the equivalent of `[^0-9]` (to be discussed soon).

- The pattern `fo\Dd` can be found in “good” and “gold”, but not in “go4d”.

Backslash-w (`\w`) represents any word character (letters, digits, and the underscore (`_`)).

- The pattern `fo\wd` can be found in “food”, “fo_d” and “fo4d”, but not in “fo*d”.

Backslash-W (`\W`) represents any character except a word character.

- The pattern `fo\Wd` can be found in “fo*d”, “fo@d” and “fo.d”, but not in “food”.

Backslash-s (`\s`) represents any whitespace character (e.g., space, tab, newline).

- The pattern `fo\s d` can be found in “fo d”, but not in “food”.

Backslash-S (`\S`) represents any character except a whitespace

character.

- The pattern `fo\sd` can be found in “fo*d”, “food” and “fo4d”, but not in “fo d”.

Character Classes (`[]`)

Square brackets (`[]`) are used to create a character class (or character set), which specifies a set of characters to match.

- The pattern `f[aeiou]d` can be found in “fad” and “fed”, but not in “food”, “fyd” or “fd”.
 - `[aeiou]` matches an “a”, an “e”, an “i”, an “o”, or a “u”
- The pattern `f[aeiou]{2}d` can be found in “faed” and “feod”, but not in “fod”, “fold” or “fd”.
- The pattern `[A-Za-z]+` can be found twice in “**Webucator, Inc.**”, but not in “13066”.
 - `[A-Za-z]` matches any lowercase or uppercase letter.
 - `[A-Z]` matches any uppercase letter.
 - `[a-z]` matches any lowercase letter.
- The pattern `[1-9]+` can be found twice in “**13066**”, but not in “Webucator, Inc.”

Negation (`^`)

When used as the first character within a character class, the caret (`^`) is used for negation. It matches any characters **not** in the set.

- The pattern `f[^aeiou]d` can be found in “fqd” and “f4d”, but not in “fad” or “fed”.

Groups (`()`)

Parentheses (`()`) are used to capture subpatterns and store them as

groups, which can be retrieved later.

- The pattern `f(oo)?d` can be found in “**food**” and “**fd**”, but not in “**fod**”.

The whole group can show up zero or one time.

Alternatives (|)

The pipe (|) is used to create optional patterns.

- The pattern `^web|or$` can be found in “**website**”, “**educator**”, and twice in “**webucator**”, but not in “cobweb” or “orphan”.

Escape Character (\)

The backslash (\) is used to escape special characters.

- The pattern `fo\d` can be found in “fo.d”, but not in “food” or “fo4d”.

Backreferences

Backreferences are special wildcards that refer back to a group within a pattern. They can be used to make sure that two subpatterns match. The first group in a pattern is referenced as `\1`, the second group is referenced as `\2`, and so on.

For example, the pattern `([bmpr])o\1` matches “**bobcat**”, “ther**mometer**”, “**popped**”, and “**prorate**”.

A more practical example has to do with matching the delimiter in social security numbers. Examine the following regular expression:

```
^\d{3}([\d- ]?)\d{2}([\d- ]?)\d{4}$
```

Within the caret (^) and dollar sign (\$), which are used to specify the

beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. This pattern will be matched in all of the following strings (and more):

- 123-45-6789
- 123 45 6789
- 123456789
- 123-45 6789
- 123 45-6789
- 123-456789

The last three strings are not ideal, but they do match the pattern. Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this:

```
^\d{3}([\d- ]?)\d{2}\1\d{4}$
```

The `\1` refers back to the first subpattern. Only the first three strings listed above match this regular expression.

Python's Handling of Regular Expressions

In Python, you use the `re` module to access the regular expression engine. Here is a very simple illustration. Imagine you're looking for the pattern `"r[aeiou]se"` in the string `"A rose is a rose is a rose."`

1. Import the `re` module:

```
import re
```

2. Compile the pattern:

```
p = re.compile('r[aeiou]se')
```

3. Search the string for a match:

```
result = p.search('A rose is a rose is a rose.')
```

4. Print the result:

```
print(result)
```

This will print the following, showing that the result is a `match` object and that it found the match “rose” starting at index 2 and ending at index 6:

```
<_sre.SRE_Match object; span=(2, 6), match='rose'>
```

Compiling a regular expression pattern into an object is a good idea if you’re going to reuse the expression throughout the program, but if you’re just using it once or twice, you can use the module-level `search()` method, like this:

```
>>> result = re.search('r[aeiou]se', 'A rose is a rose is a rose')
>>> result
<re.Match object; span=(2, 6), match='rose'>
```

Raw String Notation

Python uses the backslash character (`\`) to escape special characters. For example `\n` is a newline character. A call to `print('a\nb\nc')` will print the letters a, b, and c each on its own line:

```
>>> print('a\nb\nc')
a
b
c
```

If you actually want to print a backslash followed by an “n”, you need to escape the backslash with another backslash, like this: `print('a\\nb\\nc')`. That will print the literal string “a\nb\nc”:

```
>>> print('a\\nb\\nc')
a\nb\nc
```


Python provides another way of doing this. Instead of escaping all the backslashes, you can use *rawstring notation* by placing the letter “r” before the beginning of the string, like this: `print(r'a\nb\nc')`:

```
>>> print(r'a\nb\nc')
a\nb\nc
```

While this may not come in very handy in most areas of programming, it is very helpful when writing regular expression patterns. That is because the regular expression syntax also uses the backslash for special characters. If you don’t use raw string notation, you may find your patterns filled with backslashes.

The takeaway: Always use raw string notation for your patterns.

Regular Expression Object Methods

1. `p.search(string)` – Finds the first substring that matches the pattern. Returns a `Match` object or `None`.

```
>>> p = re.compile(r'\W')
>>> p.search('andré@example.com')
<re.Match object; span=(5, 6), match='@'>
```

This finds the first non-word character.

2. `p.match(string)` – Like `search()`, but the match must be found at the beginning of the string. Returns a `Match` object or `None`.

```
>>> p = re.compile(r'\W')
>>> p.match('andré@example.com') # Returns None
>>> p.match('@example.com')
<re.Match object; span=(0, 1), match='@'>
```

This matches the first character if it is a non-word character. The first example returns `None` as the passed-in string begins with “a”.

3. `p.fullmatch(string)` – Like `search()`, but the whole string must match. Returns a `Match` object or `None`.

```
>>> p = re.compile(r'[\w\.] +@example.com')
>>> p.match('andré@example.com')
<re.Match object; span=(0, 17), match='andré@example.com'>
```

This matches a string made up of word characters and periods followed by “@example.com”.

4. `p.findall(string)` – Finds all non-overlapping matches. Returns a list of strings.

```
>>> p = re.compile(r'\W')
>>> p.findall('andré@example.com')
['@', '.']
```

This returns a list of all matches of non-word characters.

5. `p.split(string, maxsplit=0)` – Splits the string on pattern matches. If `maxsplit` is nonzero, limits splits to `maxsplit`. Returns a list of strings.

```
>>> p = re.compile(r'\W')
>>> p.split('andré@example.com')
['andré', 'example', 'com']
```

This splits the string on non-word characters.

6. `p.sub(repl, string, count=0)` – Replaces all non-overlapping matches in `string` with `repl`. If `count` is nonzero, limits replacements to `count`. More details on `sub()` under [Using sub\(\) with a Function](#). Returns a string.

All the methods that search a string for a pattern (`search()`, `match()`, `fullmatch()`, and `findall()`) include `start` and `end` parameters that indicate what positions in the string to start and end the search.

Groups

As discussed earlier, parentheses in regular expression patterns are used to capture groups. You can access these groups individually using a `match` object’s `group()` method or all at once using its `groups()`

method.

The `group()` method takes an integer⁴² as an argument and returns a string:

- `match.group(0)` returns the whole match.
- `match.group(1)` returns the first group found.
- `match.group(2)` returns the second group found.
- And so on...

You can also get multiple groups at the same time returned as a tuple of strings by passing in more than one argument (e.g., `match.group(1, 2)`).

When nested parentheses are used in the pattern, the outer group is returned before the inner group. Here is an example that illustrates that:

```
>>> import re
>>> p = re.compile(r'(\w+)@(\w+\.((\w+)))')
>>> match = p.match('andre@example.com')
>>> email = match.group(0)
>>> handle = match.group(1)
>>> domain = match.group(2)
>>> domain_type = match.group(3)
>>> print(email, handle, domain, domain_type, sep='\n')
andre@example.com
andre
example.com
com
```

Notice that “example.com” is group 2 and “com”, which is nested within “example.com” is group 3.

And you can use the `groups()` method to get them all at once:

```
>>> print(match.groups())
('andre', 'example.com', 'com')
```

Flags

The `compile()` method takes an optional second argument: `flags`. The flags are constants that can be used individually or combined with a pipe (`|`).

```
re.compile(pattern, re.FLAG1|re.FLAG2)
```

The two most useful flags are (shortcut versions in parentheses):

1. `re.IGNORECASE` (`re.I`) – Makes the pattern case insensitive.
2. `re.MULTILINE` (`re.M`) – Makes `^` and `$` consider each line as an independent string.

Using `sub()` with a Function

The `sub()` method can either replace each match with a string or with the return value of a specified function. The function receives the match as an argument and must return a string that will replace the matched pattern. Here is an example:

Demo 99: regular-expressions/Demos/clean_cusses.py

```
import re
import random

def clean_cuss(match):
    # Get the whole match
    cuss = match.group(0)
    # Generate a random list of characters the length of cuss
    chars = [random.choice('!@#$$%^&*') for letter in cuss]
    # Return the list joined into a string
    return ''.join(chars)

def main():
    pattern = r'\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]';
    p = re.compile(pattern, re.IGNORECASE|re.MULTILINE)
    s = """Shucks! What a cruddy day I've had.
I spent the whole darn day with my slobbiest
friend darning his STINKY socks."""
    result = p.sub(clean_cuss, s)
    print(result)

main()
```

Code Explanation

Reading regular expressions is tricky. You have to think like a computer and parse it part by part:

Word boundary:

```
\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b
```

0 or more lowercase letters:

```
\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b
```

Any one of the words delimited by the pipes (|):

```
\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b
```

0 or more lowercase letters:

```
\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b
```

Word boundary:

```
\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b
```

Notice that we compile the pattern using the `re.IGNORECASE` and `re.MULTILINE` flags:

```
p = re.compile(pattern, re.IGNORECASE|re.MULTILINE)
```

The following screenshot shows the regular expression matches in pythex.org:

The screenshot shows the pythex.org interface. At the top, it says "Your regular expression:" followed by a text input containing the pattern `\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b`. Below this are four buttons: "IGNORECASE", "MULTILINE" (which is highlighted in yellow), "DOTALL", and a partially visible "V" button. Underneath is the "Your test string:" section with a text area containing the text: "Shucks! What a cruddy day I've had. I spent the whole darn day with my slobbiest friend darning his stinky socks." The bottom section is labeled "Match result:" and shows the same text with several words highlighted in green: "cruddy", "darn", "slobbiest", "darning", and "stinky". To the right of the match results, there is a partially visible table with headers "Ma" and "M".

Run the file at the terminal to see that it replaces all those matches

with a random string of characters:

```
PS ...\regular-expressions\Demos> python clean_cusses.py
@$@$#!! What a !&!#%* day I've had.
I spent the whole &*$* day with my @$@!%$^*^
friend ^&#*&#& his ^!@*#@ socks.
```

In an [earlier lesson](#), we split the text of the *U.S. Declaration of Independence* on spaces to create a counter showing which words were used the most often. The resulting list looked like this:

```
[('PEOPLE', 13), ('STATES', 7), ('SHOULD', 5), ('INDEPENDENT', 5),
 ('AGAINST', 5), ('GOVERNMENT', 4), ('ASSENT', 4),
 ('OTHERS', 4), ('POLITICAL', 3), ('POWERS', 3)]
```

In the following demo, we use a regular expression to split on any character that is not a capital letter:

Demo 100: regular-expressions/Demos/counter_re.py

```
import re
from collections import Counter

with open('Declaration_of_Independence.txt') as f:
    doi = f.read().upper()

word_list = [word for word in re.split('[^A-Z]', doi) if len(word) > 1]

c = Counter(word_list)
print(c.most_common(10))
```

Code Explanation

Because we use `upper()` to convert the whole text to uppercase, we can split on `[^A-Z]`. If we didn't know that there were only uppercase letters, we would have used `[^A-Za-z]` instead.

The new results are:

```
[('PEOPLE', 17), ('STATES', 12), ('GOVERNMENT', 7), ('POWERS', 6),
 ('BRITAIN', 6), ('SHOULD', 5), ('COLONIES', 5), ('INDEPENDENT', 5),
 ('AGAINST', 5), ('MANKIND', 4)]
```

Exercise 38: Green Glass Door

20 to 30 minutes.

In this exercise, you will modify a function so that it uses a regular expression. But first, a little riddle:

The following items can pass through the green glass door:

1. puddles
2. mommies

3. aardvarks
4. balloons

The following items cannot pass through the green glass door:

1. ponds
2. moms
3. anteaters
4. kites

Knowing that, which of the following can pass through the green glass door?

1. bananas
2. apples
3. pears
4. grapes
5. cherries

Did you figure it out? The two that can pass are **apples** and **cherries**. Any word with a double letter can pass through the **green glass door**.

Now, take a look at the following code:

Exercise Code: regular-expressions/Exercises/green_glass_door.py

```
def green_glass_door(word):
    prev_letter = ''
    for letter in word:
        letter = letter.upper()
        if letter == prev_letter:
            return True
        prev_letter = letter
    return False

fruits = ['banana', 'apple', 'pear', 'grape', 'cherry',
          'persimmons', 'orange', 'passion fruit']

for fruit in fruits:
    if green_glass_door(fruit):
        print(f'YES! {fruit} can pass through the green glass door')
    else:
        print(f'NO! {fruit} cannot pass through the green glass door')
```

Study the code, paying particular attention to the `green_glass_door()` function. Your job is to rewrite that function to use a regular expression. Don't forget to import `re`.

Solution: regular-expressions/Solutions/green_glass_door.py

```
import re

def green_glass_door(word):
    pattern = re.compile(r'(\.)\1')
    return pattern.search(word)

fruits = ['banana', 'apple', 'pear', 'grape', 'cherry',
          'persimmons', 'orange', 'passion fruit']

for fruit in fruits:
    if green_glass_door(fruit):
        print(f'YES! {fruit} can pass through the green glass door')
    else:
        print(f'NO! {fruit} cannot pass through the green glass door')
```

Code Explanation

The first part of the pattern matches any character. It uses parentheses to create a group:

```
pattern = re.compile(r'(\.)\1')
```

The second part of the pattern uses a backreference to match the first group: that is, the character matched by (.):

```
pattern = re.compile(r'(\.)\1')
```

The function then returns the result of searching the string for that pattern:

```
return pattern.search(word)
```

That will either return a `Match` object, which evaluates to `True`, or it will return `None`, which evaluates to `False`.

Conclusion

In this lesson, you have learned how to work with regular expressions in Python. To learn more about regular expressions, see *Python's Regular Expression HOWTO*⁴³.

LESSON 15

Working with Data

Topics Covered

- Data stored in a relational database.
- Data stored in a CSV file.
- Data from a web page.
- HTML, XML, and JSON.
- Accessing an API.

They saw more rooms and made more discoveries than Mary had made on her first pilgrimage. They found new corridors and corners and flights of steps and new old pictures they liked and weird old things they did not know the use of.

– *The Secret Garden, Frances Hodgson Burnett*

Data is stored in many different places and in many different ways. In this lesson, you'll learn about the Python modules that help you access data.

Virtual Environment

In this lesson, you will be installing some libraries. So as not to mess up your standard environment, you should create a virtual environment and work within it the entire lesson:

1. Open a terminal at [working-with-data](#) and run the following command:

```
PS ...\\Python\\working-with-data> python -m venv .venv
```

This will create and populate a new [.venv](#) directory.

2. Activate the new virtual environment:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

Your prompt should now be prefaced with “(.venv)”.

3. Throughout this lesson, you should work in the virtual directory. So, if you deactivate it to do other work (or play), be sure to reactivate it when you’re ready to proceed.

Relational Databases

In this lesson, we will be working with MySQL and SQLite, but Python is able to connect to all the commonly used databases, including PostgreSQL, Microsoft SQL Server, and Oracle. All implementations for working with different relational databases should follow [PEP 0249 -- Python Database API Specification v2.0](#)⁴⁴, which we will describe shortly.

Lahman’s Baseball Database

The database we will use for our examples is [Lahman’s Baseball Database](#)⁴⁵, which includes a huge amount of data on Major League Baseball from 1871 to the present.

We have a hosted version of the MySQL database on Amazon. The connection information is as follows:

- **host:** lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com
- **user:** python
- **password:** python
- **database:** lahmanbaseballdb

If this connection doesn't work for you, see <https://www.webucator.com/books/errata.cfm> to see if something has changed.

Local Version of the MySQL Database

If you have MySQL installed locally and wish to create a local version of the MySQL database, see <https://github.com/WebucatorTraining/lahman-baseball-mysql> for instructions.

We will be using Oracle's `mysql-connector-python`⁴⁶ to connect Python to MySQL. Install `mysql-connector-python` in your virtual environment by running:

```
pip install mysql-connector-python
```

PEP⁴⁷ 0249 -- Python Database API Specification v2.0

PEP 0249 defines an API for Python interfaces that work with databases. Generally, you follow the following steps to pull data from a database (be sure to open a Python terminal and follow along):

1. Import a Python Database API-2.0-compliant interface.

```
import mysql.connector
```

2. Open a connection to the database.

```
connection = mysql.connector.connect(  
    host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',  
    user='python',  
    passwd='python',  
    db='lahmansbaseballdb'  
)
```

3. Write your query. For example, this query will get the first and last

names, weight, and year of debut for the heaviest five people in the people table:

```
query = """SELECT nameFirst, nameLast, weight, year(debut)
            FROM people
            ORDER BY weight DESC
            LIMIT 5
            """
```

4. Create a cursor for the connection:

```
cursor = connection.cursor()
```

5. Use the cursor to execute one or more queries:

```
cursor.execute(query)
```

6. Get the results of the query/queries from the cursor:

```
results = cursor.fetchall()
```

7. Do something with the results. Here we just output the results:

```
results
```

8. Close the cursor:

```
cursor.close()
```

9. Close the connection to the database:

```
connection.close()
```

Here is the complete code:

Demo 101: working-with-data/Demos/fetch_all.py

```
# Be sure to install via pip install mysql-connector-python
import mysql.connector

connection = mysql.connector.connect(
    host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
    user='python',
    passwd='python',
    db='lahmansbaseballdb'
)

query = """SELECT nameFirst, nameLast, weight, year(debut)
           FROM people
           ORDER BY weight DESC
           LIMIT 5"""

cursor = connection.cursor()
cursor.execute(query)
results = cursor.fetchall()

cursor.close()
connection.close()

print(results)
```

Code Explanation

This will output a list of tuples, one tuple for each record returned:

```
[
    ('Walter', 'Young', 320, 2005),
    ('Jumbo', 'Diaz', 315, 2014),
    ('CC', 'Sabathia', 300, 2001),
    ('Dmitri', 'Young', 295, 1996),
    ('Jumbo', 'Brown', 295, 1925)
]
```

Cursor Methods

These are the most common cursor methods:

1. `cursor.execute(operation [, parameters])` – Prepares and executes a database query or command. The returned value varies by implementation.
2. `cursor.executemany(operation, seq_of_parameters)` – Prepares a database query or command and executes it once for each item in `seq_of_parameters`. This is usually used for `INSERT` and `UPDATE` statements, not for queries that return result sets. The returned value varies by implementation.
3. `cursor.fetchone()` – Fetches the next row of a result set. Returns a single row of data.
4. `cursor.fetchmany(n=cursor.arraysize)` – Fetches the next `n` rows of a result set. `cursor.arraysize` defaults to 1. Returns a list of data rows.
5. `cursor.fetchall()` – Fetches all data rows of a result set. Returns a list of data rows.

When to Use `fetchone()`

As a general rule, after executing a `SELECT` statement, you will use `cursor.fetchall()` to fetch all the results. If you want a limited number of records, you should use SQL to set that limit.

One exception to this rule is when you know you are just getting one record from the database. In the following example, we limit the number of records to one by selecting on the primary key field:

Demo 102: working-with-data/Demos/fetch_one.py

```
import mysql.connector

connection = mysql.connector.connect(
    host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
    user='python',
    passwd='python',
    db='lahmansbaseballdb'
)

query = """SELECT nameFirst, nameLast, birthCity, birthState, bi
            FROM people
            WHERE playerID = 'jeterde01';"""

cursor = connection.cursor()
cursor.execute(query)
result = cursor.fetchone()

if result:
    player_name = result[0] + ' ' + result[1]
    birth_place = result[2] + ', ' + result[3]
    birth_year = result[4]
    print(f'{player_name} was born in {birth_place} in {birth_ye
else:
    print('No records returned.')

cursor.close()
connection.close()
```

Code Explanation

The result is a single tuple, which we use to create more meaningfully named variables, and then output:

```
Derek Jeter was born in Pequannock, NJ in 1974.
```

Notice that we use an `if` condition to check to make sure a result was

returned just in case there are no players in the `people` table with that `playerID`.

Returning Dictionaries instead of Tuples

According to PEP 0249, the cursor fetch methods must return a single sequence of values (for one row) or a sequence of sequences (for multiple rows). By default, `mysql-connector-python` returns a tuple (for `fetchone()`) and a list of tuples (for `fetchmany()` and `fetchall()`), but you can change the cursor type to a dictionary by passing in `dictionary=True` to the `cursor()` method:

```
cursor = connection.cursor(dictionary=True)
```

For the query getting the five heaviest baseball players of all time, this would change the result from a list of tuples to a list of dictionaries. Give it a try:

1. Open `working-with-data/Demos/fetch_all.py` in your editor.
2. Pass `dictionary=True` to the `connection.cursor()` method:

```
cursor = connection.cursor(dictionary=True)
```

3. Run the file. It will output something like this (though not laid out so nicely^{[48](#)}):

```
[
  {
    'nameFirst': 'Walter',
    'nameLast': 'Young',
    'weight': 320,
    'year(debut)': 2005
  },
  {
    'nameFirst': 'Jumbo',
    'nameLast': 'Diaz',
```

```
        'weight': 315,  
        'year(debut)': 2014  
    },  
    {  
        'nameFirst': 'CC',  
        'nameLast': 'Sabathia',  
        'weight': 300,  
        'year(debut)': 2001  
    },  
    {  
        'nameFirst': 'Dmitri',  
        'nameLast': 'Young',  
        'weight': 295,  
        'year(debut)': 1996  
    },  
    {  
        'nameFirst': 'Jumbo',  
        'nameLast': 'Brown',  
        'weight': 295,  
        'year(debut)': 1925  
    }  
]
```

A big advantage of having the data returned as dictionaries is that you can then reference the columns by name instead of by position. The code in the following file illustrates this:

Demo 103: working-with-data/Demos/fetch_one_as_dict.py

```
-----Lines Omitted-----
query = """SELECT nameFirst, nameLast, birthCity, birthState, bi
           FROM people
           WHERE playerID = 'jeterde01';"""

cursor = connection.cursor(dictionary=True)
cursor.execute(query)
result = cursor.fetchone()

if result:
    player_name = result['nameFirst'] + ' ' + result['nameLast']
    birth_place = result['birthCity'] + ', ' + result['birthState']
    birth_year = result['birthYear']
    print(f'{player_name} was born in {birth_place} in {birth_year}')
-----Lines Omitted-----
```

Code Explanation

Notice that we can reference the columns as `result['nameFirst']`, `result['nameLast']`, etc. That's much more readable than `result[0]`, `result[1]`, etc.

Passing Parameters

Often, your Python code won't know some of the values in the SQL query until execution time. For example, we could write a program that allowed users to find out which five players had the most home runs in a given year. The SQL query for that would look like this:

```
SELECT p.nameFirst, p.nameLast, b.HR, t.name AS team, b.yearID
FROM batting b
     JOIN people p ON p.playerID = b.playerID
     JOIN teams t ON t.ID = b.team_ID
WHERE b.yearID = 1950
```

```
ORDER BY b.HR DESC
LIMIT 5;
```

But we need to make the year a variable.

The Wrong Way

You might be tempted to do this with a Python variable like this:

Don't do this!

```
year_id = int(input('Enter a year: '))

query = f"""SELECT p.nameFirst, p.nameLast, b.HR, t.name AS team
FROM batting b
JOIN people p ON p.playerID = b.playerID
JOIN teams t ON t.ID = b.team_ID
WHERE b.yearID = {year_id}
ORDER BY b.HR DESC
LIMIT 5;"""
```

In this case, Python creates the whole query as a string and sends it to the database to run.

There are multiple problems with the above approach, but the biggest one is that it opens the database to hacking. A savvy and nefarious person could try to pass in a value to `year_id` that ended one query and started another that either sought to extract extra data (e.g., passwords) from the database or tried to wreak havoc on your database by updating or deleting records.

The Right Way

The right way to construct a SQL query is to pass parameters to the database and let it do the work of constructing the query. This is beneficial for at least a couple of reasons:

1. It mitigates the security risk. The database can check the passed-

in parameters to make sure that they match the data types of the corresponding columns. Any code that tried to end one query and start another would get rejected.

2. It allows the database to compile the query so that it can reuse it with different passed-in parameters without recompiling every time.

Different databases use different placeholders for parameters:

1. MySQL and PostgreSQL use %s.
2. Oracle⁴⁹ uses a : followed by a variable name or index.
3. SQL Server and SQLite use a ?.

The following demo uses MySQL:

Demo 104: working-with-data/Demos/homerun_leaders_mysql.py

```
import mysql.connector

def main():
    connection = mysql.connector.connect(
        host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
        user='python',
        passwd='python',
        db='lahmansbaseballdb'
    )

    cursor = connection.cursor(prepared=True)

    query = f"""SELECT p.nameFirst, p.nameLast, b.HR,
                    t.name AS team, b.yearID
                FROM batting b
                    JOIN people p ON p.playerID = b.playerID
                    JOIN teams t ON t.ID = b.team_ID
                WHERE b.yearID = %s
                ORDER BY b.HR DESC
                LIMIT 5;"""

    checking = True
    while checking:
        year_id = int(input('Enter a year (0 to quit): '))
        if year_id == 0:
            break

        cursor.execute(query, [year_id])
        results = cursor.fetchall()

        for i, result in enumerate(results, 1):
            row = dict(zip(cursor.column_names, result))
            name = f"{row['nameFirst']} {row['nameLast']}"
            print(f"{i}. {name}: {row['HR']}")

    cursor.close()
    connection.close()
```

```
main()
```

Code Explanation

Notice the “%s” parameter on line 18 and the way the parameter is passed into `cursor.execute()` on line 28:

```
cursor.execute(query, [year_id])
```

Run the program. You will see that it prompts the user repeatedly for a year and then retrieves and prints out the five leading home-run hitters for that year.

Things to notice:

1. We pass `prepared=True` to `connection.cursor()`. This tells MySQL to create a `MySQLCursorPrepared` cursor, which is the type of cursor it uses to prepare SQL queries.
2. Unfortunately, you cannot use `dictionary=True` and `prepared=True` together, so we zip the `cursor.column_names` with each returned result to create a dictionary:

```
for i, result in enumerate(results , 1):  
    row = dict(zip(cursor.column_names, result))  
    name = f"{row['nameFirst']} {row['nameLast']}"  
    print(f"{i}. {name}: {row['HR']}")
```

This allows us to access the values by name.

The `zip()` function was covered in **Creating a Dictionary from Two Sequences** in the [Advanced Python Concepts lesson](#).

SQLite⁵⁰

SQLite is a server-less SQL database engine. Each SQLite database is stored in a single file that can easily be transported between

computers. While SQLite is not as robust as enterprise relational database management systems, it works great for local databases or databases that don't have large loads.

Python's `sqlite3` module conforms to PEP 0249.

Download the SQLite version of the Database

Webucator maintains a SQLite version of Lahman's Baseball Database at <https://github.com/WebucatorTraining/lahman-baseball-mysql>.

Download `lahmansbaseballdb.sqlite` into the `working-with-data/data` folder.

The following code shows how to create the same home-run-leader program, but this time we connect to a SQLite database instead of MySQL:

```
import sqlite3

def main():
    connection = sqlite3.connect('../data/lahmansbaseballdb.sqlite')
    connection.row_factory = sqlite3.Row

    cursor = connection.cursor()

    query = f"""SELECT p.nameFirst, p.nameLast, b.HR,
                    t.name AS team, b.yearID
                FROM batting b
                JOIN people p ON p.playerID = b.playerID
                JOIN teams t ON t.ID = b.team_ID
                WHERE b.yearID = ?
                ORDER BY b.HR DESC
                LIMIT 5;"""

    checking = True
    while checking:
        year_id = int(input('Enter a year (0 to quit): '))
        if year_id == 0:
            break

        cursor.execute(query, [year_id])
        results = cursor.fetchall()

        for i, result in enumerate(results, 1):
            name = f"{result['nameFirst']} {result['nameLast']}"
            print(f"{i}. {name}: {result['HR']}")

    cursor.close()
    connection.close()

main()
```

Things to notice:

1. We connect to a SQLite database stored in working-with-data/data.
 2. By default, data is returned as a tuple, so you have to access values by index. Setting `connection.row_factory` to `sqlite3.Row` allows you to access values in returned records by column name or index. It is analagous to passing `dictionary=True` to `connection.cursor()` when using `mysql-connector-python`.
 3. SQLite uses question marks as placeholders for parameters.
 4. Everything else is done the same as it was done with MySQL.
-

Exercise 39: Querying a SQLite Database

10 to 15 minutes.

In this exercise, you will use your knowledge of PEP 0249 to connect to and query the lahmansbaseballdb.sqlite database.

1. Open working-with-data/Exercises/querying_a_sqlite_database.py in your editor.
2. The connection to the SQLite database has already been made and the query has been written. We have also included the following line, which allows you to access the values by column name:

```
connection.row_factory = sqlite3.Row
```

3. Add code that runs the query and assigns the results to the `results` variable.
4. Print out the results using a for loop, so that it outputs:

```
Walter Young weighed 320 when he debuted in 2005.  
Jumbo Diaz weighed 315 when he debuted in 2014.
```

```
CC Sabathia weighed 300 when he debuted in 2001.  
Jumbo Brown weighed 295 when he debuted in 1925.  
Dmitri Young weighed 295 when he debuted in 1996.
```

Note that the `debut` field is returned as a string in the format `YYYY-MM-DD`. You could use the `datetime` module to get at the year, or you can just use slicing.

5. Don't forget to close your cursor and connection.

Solution: `working-with-data/Solutions/querying_a_sqlite_database.py`

```
import sqlite3
connection = sqlite3.connect('../data/lahmansbaseballdb.sqlite')
connection.row_factory = sqlite3.Row

query = """SELECT nameFirst, nameLast, weight,
                debut AS debut
            FROM people
            ORDER BY weight DESC
            LIMIT 5"""

cursor = connection.cursor()
cursor.execute(query)
results = cursor.fetchall()
cursor.close()
connection.close()

for result in results:
    player_name = result['nameFirst'] + ' ' + result['nameLast']
    weight = result['weight']
    year = result['debut'][:4]
    print(f'{player_name} weighed {weight} when he debuted in {year}')
```

SQLite Database in Memory

Python allows you to create in-memory databases with SQLite. This can be useful when you have a lot of data in a tab-delimited file that you want to query using SQL, but don't want to maintain as a database file as well. To create a connection to an in-memory database, use the following code:

```
connection = sqlite3.connect(':memory:')
```

In-memory SQLite Database

For the rest of the database section, we will work with an in-memory SQLite database, but the concepts apply to all databases.

You then create your tables with `CREATE TABLE` statements and populate them with `INSERT` statements. For example:


```
import sqlite3
connection = sqlite3.connect(':memory:')
cursor = connection.cursor()

create = """CREATE TABLE beatles (
            'fname' text,
            'lname' text,
            'nickname' text
        )"""

cursor.execute(create)

members = [
    ('John', 'Lennon', 'The Smart One'),
    ('Paul', 'McCartney', 'The Cute One'),
    ('George', 'Harrison', 'The Funny One'),
    ('Ringo', 'Starr', 'The Quiet One')
]

insert = 'INSERT INTO beatles VALUES (?, ?, ?)'

# Loop through the members list, inserting each member
for member in members:
    cursor.execute(insert, member)

select = 'SELECT fname, lname, nickname FROM beatles'
cursor.execute(select)

results = cursor.fetchall()
cursor.close()
connection.close()

print(results)
```

Executing Multiple Queries at Once

You might have noticed that, to insert the records, we looped through a list of lists, inserting one record at a time using:

```
cursor.execute(insert, member)
```

A better way of doing this is to use the `executemany()` method, which takes two arguments:

1. The query to run, which usually includes some placeholders to replace with passed-in values.
2. A sequence of sequences, each of which contains the values with which to replace the placeholders.

The query will run once for each sequence in the sequence of sequences. Different databases may implement this in different ways; however, the Python code should work with any database as long as you're using a Python Database API-2.0-compliant interface.

Here is an example:

```
-----Lines Omitted-----
members = [
    ('John', 'Lennon', 'The Smart One'),
    ('Paul', 'McCartney', 'The Cute One'),
    ('George', 'Harrison', 'The Funny One'),
    ('Ringo', 'Starr', 'The Quiet One')
]

insert = 'INSERT INTO beatles VALUES (?, ?, ?)'
cursor.executemany(insert, members)
-----Lines Omitted-----
```

Exercise 40: Inserting File Data into a Database

20 to 30 minutes.

In this exercise, you will use the data from a text file to populate a database table.

Open [working-with-data/data/states.txt](#) in your editor. The file has 52 lines of data (50 states and Washington, D.C. and Puerto Rico). Each line contains three pieces of data^{[51](#)} separated by tabs:

1. State Name
2. Population in 2020
3. Population in 2000

For example, the line for California reads:

California	39,631,049	37,254,523
------------	------------	------------

Here is the starting code:

Exercise Code: working-with-data/Exercises/inserting_file_data_into_a_database.py

```
import sqlite3
connection = sqlite3.connect(':memory:')
connection.row_factory = sqlite3.Row

# Create the cursor.

create = """CREATE TABLE states (
            'state' text,
            'pop2020' integer,
            'pop2000' integer
        )"""

# Execute the create statement.

insert = 'INSERT INTO states VALUES (?, ?, ?)'

# Create a list of tuples from the data in '../data/states.txt'.

# Insert the data into the database.

select = """SELECT state,
                    CAST( (pop2020*1.0/pop2000) * pop2020 AS INTEGER) pop
                FROM states ORDER BY pop2040 DESC"""

# Execute the select statement.

# Fetch the rows into a variable.

# Close the cursor and connection.

results = cursor.fetchall()
cursor.close()
connection.close()

# Print out the results.
```

1. Open working-with-data/Exercises/inserting_file_data_into_a_database.py in your editor. A connection to an in-memory database has already been established and the SQL statements for creating the table, inserting the records, and selecting the data have been written and stored in variables.
2. Your job is to:
 - A. Create a cursor.
 - B. Run the `CREATE` statement.
 - C. Get the data from the states.txt file into a list of 52 three-element tuples. **Note** that you will have to remove the commas from the population numbers so that they are valid integers.
 - D. Insert the rows using the list of tuples (i.e., the sequence of sequences) you just created.
 - E. Run the `SELECT` statement. This returns two columns: the state and the projected population in 2040. The column names are `state` and `pop2040`.
 - F. Fetch the results and output a sentence for each row (e.g., “The projected 2040 population of California is 42,159,177.”).
 - G. Don’t forget to close your cursor and connection.

Solution: working-with-data/Solutions/inserting_file_data_into_a_database.py

```
import sqlite3
connection = sqlite3.connect(':memory:')
connection.row_factory = sqlite3.Row

cursor = connection.cursor()

create = """CREATE TABLE states (
            'state' text,
            'pop2020' integer,
            'pop2000' integer
        )"""

cursor.execute(create)

insert = 'INSERT INTO states VALUES (?, ?, ?)'

data = []
with open('../data/states.txt') as f:
    for line in f.readlines():
        state_data = line.split('\t')
        tpl_state_data = (state_data[0],
                           int(state_data[1].replace(',', '')),
                           int(state_data[2].replace(',', '')))

        data.append(tpl_state_data)

cursor.executemany(insert, data)

select = """SELECT state,
                    CAST( (pop2020*1.0/pop2000) * pop2020 AS INTEGER) pop
                FROM states ORDER BY pop2040 DESC"""

cursor.execute(select)

results = cursor.fetchall()
cursor.close()
connection.close()
```

```
for record in results:
    state = record['state']
    pop2040 = record['pop2040']
    print(f'The projected 2040 population of {state} is {pop2040}
```

Drivers for Other Databases

We have used `mysql-connector-python` to connect Python to MySQL and Python's `sqlite3` module to connect to SQLite. We recommend the following drivers for other major databases:

- PostgreSQL: `psycopg2` (<https://pypi.org/project/psycopg2/>)
- SQL Server: `pyodbc` (<https://pypi.org/project/pyodbc/>)
- Oracle: `cx-Oracle` (<https://pypi.org/project/cx-Oracle/>)

All of these comply with PEP 0249.

CSV

CSV (for “Comma Separated Values”) is a format commonly used for sharing data between applications, in particular, database and spreadsheet applications. Because the format had been around for awhile before any attempt was made at standardization, not all CSV files use exactly the same format. Fortunately, Python's `csv` module does a good job of handling and hiding these differences so the programmer generally doesn't have to worry about them.

Microsoft Excel is perhaps the most common application used for making CSV files. Here is a sample CSV file ([working-with-data/data/population-by-state.csv](#)) in Microsoft Excel showing the United States population breakdown⁵² over several years:

	A	B	C	D	E	F
1	State	2010	2011	2012	2013	2014
2	Alabama	4,785,437	4,799,069	4,815,588	4,830,081	4,841,791
3	Alaska	713,910	722,128	730,443	737,068	736,283
4	Arizona	6,407,172	6,472,643	6,554,978	6,632,764	6,730,411
5	Arkansas	2,921,964	2,940,667	2,952,164	2,959,400	2,967,391
6	California	37,319,502	37,638,369	37,948,800	38,260,787	38,596,971

Open the CSV file in a text editor and you'll see this (lines are cut off):

```
(.venv) PS ...\\working-with-data\\Demos> python csv_reader.py
State,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020
Alabama,"4,785,437","4,799,069","4,815,588","4,830,081","4,841,791"
Alaska,"713,910","722,128","730,443","737,068","736,283","737,491"
Arizona,"6,407,172","6,472,643","6,554,978","6,632,764","6,730,411"
Arkansas,"2,921,964","2,940,667","2,952,164","2,959,400","2,967,391"
California,"37,319,502","37,638,369","37,948,800","38,260,787","38,596,971"
...
```

Python's `csv` module is used for:

1. Reading from a CSV file.
2. Creating a new CSV file.
3. Writing to an existing CSV file.

Reading from a CSV File

To read data from a CSV file:

1. Open the file using the built-in `open()` function with `newline` set to an empty string.
2. Pass the file object to the `csv.reader()` method.
3. Read the file row by row. Each row is a list of strings.

Demo 108: working-with-data/Demos/csv_reader.py

```
import csv

csv_file = '../data/population-by-state.csv'
with open(csv_file, newline='', encoding="utf-8") as csvfile:
    pops = csv.reader(csvfile)
    for row in pops:
        print(' '.join(row))
```

Code Explanation

This will output the following (lines are cut off):

```
(.venv) PS ...\\working-with-data\\Demos> python csv_dictreader_1.py:
Alabama, 4,785,437, 4,799,069, 4,815,588, 4,830,081, 4,841,799, .
Alaska, 713,910, 722,128, 730,443, 737,068, 736,283, 737,498, 74:
Arizona, 6,407,172, 6,472,643, 6,554,978, 6,632,764, 6,730,413, (
Arkansas, 2,921,964, 2,940,667, 2,952,164, 2,959,400, 2,967,392,
California, 37,319,502, 37,638,369, 37,948,800, 38,260,787, 38,5:
...
```

DictReader

When retrieving rows using the `reader()` method, it's possible to manipulate them item by item, but you have to know the positions of the different fields. For a CSV with a lot of columns, that can be pretty difficult. You will likely find it easier to use a `DictReader`, which gives you access to the fields by key.

For example, in the population CSV, the first column holds the name of the state, and each subsequent columns holds the population for a given year. The following report shows the growth between 2010 and 2020 for each state.

Demo 109: working-with-data/Demos/csv_dictreader_1.py

```
import csv

csvfile = '../data/population-by-state.csv'
with open(csvfile, newline='', encoding="utf-8") as csvfile:
    pops = csv.DictReader(csvfile)
    print('Headers:', pops.fieldnames)
    for row in pops:
        # Convert to integers
        pop_2020 = int(row['2020'].replace(',', ''))
        pop_2010 = int(row['2010'].replace(',', ''))

        growth = pop_2020 - pop_2010

        # Use "," format spec to separate 1000s with commas
        print(f"{row['State']}: {pop_2020:,} - {pop_2010:,} = {growth:,}")
```

Code Explanation

This will output the following:

```
Headers: ['State', '2010', '2011', '2012', '2013', '2014', '2015']
Alabama: 4,914,833 - 4,785,437 = 129,396
Alaska: 742,866 - 713,910 = 28,956
Arizona: 7,341,977 - 6,407,172 = 934,805
Arkansas: 3,030,315 - 2,921,964 = 108,351
California: 40,037,709 - 37,319,502 = 2,718,207
...
```

Code Explanation

Things to notice:

1. `csv.DictReader` objects have a `fieldnames` property that contains a list holding the keys taken from the first row of data.
2. We use the `replace()` method of strings to get rid of the commas

in the population numbers and then we `int()` the result.

3. When we print the result, we add the commas back in [using formatting](#).
-

fieldnames

By default the `fieldnames` property of `csv.DictReader` objects contains a list holding the keys taken from the first row of data. If the CSV doesn't have a header row, you can pass `fieldnames` in when creating the `DictReader`:

```
fieldnames = ['State', '2010', '2011', '2012', '2013', '2014',  
              '2015', '2016', '2017', '2018', '2019', '2020']  
pops = csv.DictReader(csvfile, fieldnames)
```

Getting CSV Data as a List

Review the following Python code:

Demo 110: working-with-data/Demos/csv_dictreader_2.py

```
import csv

def get_data_from_csv(csvfile):
    with open(csvfile, newline='', encoding="utf-8") as csvfile:
        data = csv.DictReader(csvfile)
        return data

def main():
    data = get_data_from_csv('../data/population-by-state.csv')

    for row in data:
        print(row['State'])

main()
```

What do you expect to happen? Clearly, the intention is to print all the states in the “State” column of the CSV, but instead the code will error:

```
ValueError: I/O operation on closed file.
```

The reason it errors is that files opened using `with` are automatically closed at the end of the `with` block. An effective way of dealing with this issue is to convert the `csv.DictReader` object to a list and return that instead:

Demo 111: working-with-data/Demos/csv_dictreader_3.py

```
import csv

def get_data_as_list_from_csv(csvfile):
    with open(csvfile, newline='', encoding="utf-8") as csvfile:
        data = csv.DictReader(csvfile)
        return list(data)

def main():
    data = get_data_as_list_from_csv('../data/population-by-state.csv')

    for row in data:
        print(row['State'])

main()
```

Exercise 41: Finding Data in a CSV File

20 to 30 minutes.

In this exercise, you will use your knowledge of Python lists and dictionaries to search data in a CSV file.

1. Create a new file and save it as `csv_search.py` in `working-with-data/Exercises`.
2. Write code that prompts the user for a state name and a year between 2010 and 2020 and then returns the population of that state in that year. The program should work like this:

```
PS ...\\working-with-data\\Exercises> python csv_search.py
State: New York
Year: 2020
New York's population in 2020: 19,588,068.
```

3. The code in `working-with-data/Demos/csv_dictreader_3.py` should serve as a good reference / starting point.

Solution: working-with-data/Solutions/csv_search.py

```
import csv

def get_data_as_list_from_csv(csvfile):
    with open(csvfile, newline='', encoding="utf-8") as csvfile:
        data = csv.DictReader(csvfile)
        return list(data)

def get_population(data, state, year):
    # Loop through data
    for row in data:
        # Is this the row that matches the passed-in state?
        if row['State'] == state:
            # Return the value for the column for the passed in year
            return row[year]
    return None # No matching state found

def main():
    data = get_data_as_list_from_csv('../data/population-by-state.csv')
    state = input('State name: ')
    year = input('Year between 2010 and 2020: ')
    population = get_population(data, state, year)
    if population:
        print(f'{state}\''s population in {year}: {population}.')
    else:
        print(f'No state found matching "{state}".')

main()
```

Creating a New CSV File

Writer

To write data to a CSV file using a `Writer` object:

1. Open the file using the built-in `open()` function in writing mode and with `newline` set to an empty string:

```
with open('../csvs/mlb-weight-over-time.csv', 'w', newline='')
```

2. Pass the file object to the `csv.writer()` method:

```
writer = csv.writer(csvfile)
```

3. Write the file row by row with the `writerow(sequence)` method or all at once with the `writerows(sequence_of_sequences)` method.

Note that the sequences mapping to rows may only contain strings and numbers.

The following example shows how you could write data retrieved from a database into a CSV file:

Demo 112: working-with-data/Demos/csv_writer.py

```
import mysql.connector
import csv

connection = mysql.connector.connect(
    host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
    user='python',
    passwd='python',
    db='lahmansbaseballdb'
)

query = """SELECT year(debut) year, avg(weight) weight
FROM people
WHERE debut is NOT NULL
GROUP BY year(debut)
ORDER BY year(debut)"""

cursor = connection.cursor()
cursor.execute(query)
results = cursor.fetchall()

cursor.close()
connection.close()

csv_file = '../data/mlb-weight-over-time.csv'
with open(csv_file, 'w', newline='', encoding='utf-8') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['Year', 'Weight'])
    writer.writerows(results)
```

DictWriter

`csv.DictWriter()` works like `csv.writer()` except that instead of mapping lists onto rows, it maps dictionaries onto rows. As such, it requires a second argument: `fieldnames`, which it uses to determine the order in which the dictionary values get mapped. To illustrate this,

let's look at a very simple example:

Demo 113: working-with-data/Demos/csv_dictwriter_1.py

```
import csv

grades = [
    {
        "English": 97,
        "Math": 93,
        "Art": 74,
        "Music": 86
    },
    {
        "English": 89,
        "Math": 83,
        "Art": 97,
        "Music": 94
    }
]

csv_file = '../data/grades.csv'
with open(csv_file, 'w', newline='', encoding='utf-8') as csvfile:
    fieldnames = ['Math', 'Art', 'English', 'Music']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(grades)
```

Code Explanation

This will create grades.csv with the following contents:

```
Math,Art,English,Music
93,74,97,86
83,97,89,94
```

Notice that the order of the fields maps to the `fieldnames` list.

To get the order of the fields from the first dictionary in the `grades` list,

use:

```
fieldnames = grades[0].keys()
```

Try making that change in csv_dictwriter_1.py. After making that change and running the file again, grades.csv will hold the following contents:

```
English,Math,Art,Music  
97,93,74,86  
89,83,97,94
```

Appending to a CSV File

To add lines to an existing CSV file, just open in it in append mode:

Demo 114: working-with-data/Demos/csv_dictwriter_2.py

```
import csv

grades = [
    {
        "English": 88,
        "Math": 88,
        "Art": 88,
        "Music": 88
    },
    {
        "English": 77,
        "Math": 77,
        "Art": 77,
        "Music": 77
    }
]

csv_file = '../data/grades.csv'
with open(csv_file, 'a', newline='', encoding='utf-8') as csvfile:
    fieldnames = grades[0].keys()
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writerows(grades)
```

Code Explanation

When appending, you don't need to call `writeheader()`.

Exercise 42: Creating a CSV with DictWriter

10 to 15 minutes.

1. Open [working-with-data/Exercises/csv_dictwriter.py](#) in your editor. It contains the same code we saw in [working-with-data/Demos/csv_writer.py](#)

2. Modify the code so that it uses a `DictWriter` instead of a `Writer` object.

Solution: working-with-data/Solutions/csv_dictwriter.py

```
import mysql.connector
import csv

connection = mysql.connector.connect(
    host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
    user='python',
    passwd='python',
    db='lahmansbaseballdb'
)

query = """SELECT year(debut) year, avg(weight) weight
FROM people
WHERE debut is NOT NULL
GROUP BY year(debut)
ORDER BY year(debut)"""

cursor = connection.cursor(dictionary=True)
cursor.execute(query)
results = cursor.fetchall()

cursor.close()
connection.close()

csv_file = '../data/mlb-weight-over-time.csv'
with open(csv_file, 'w', newline='', encoding='utf-8') as csvfile:
    fieldnames = results[0].keys()
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(results)
```

Getting Data from the Web

The Requests Package

Python has a built-in `urllib` module for making HTTP requests, but the [Requests](#) ⁵³ package is far more developer-friendly. In this section,

you'll learn how the `Requests` package works and how to combine it with the `Beautiful Soup`⁵⁴ library to parse the code.

The Requests Package

The first thing is to install the `Requests` package. With your virtual environment activated, run:

```
pip install requests
```

Although all HTTP request methods (e.g., `post`, `put`, `head`,...) can be used, in most cases, you will use the `get` method using `requests.get()`, to which you will pass in the URL as a string like this:

Demo 115: working-with-data/Demos/using_requests.py

```
import requests

url = 'https://www.webucator.com/course-demos/python/hrleaders.cfm'
r = requests.get(url)
content = r.text
print(content[:125]) # print first 125 characters
```

Code Explanation

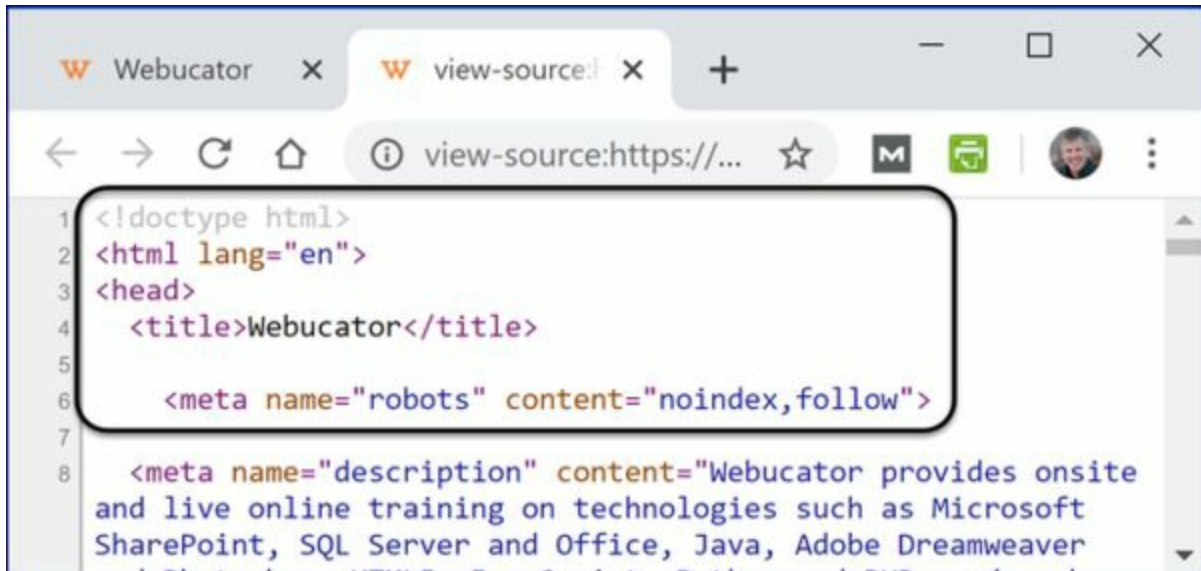
This will grab all the content from the web page at <https://www.webucator.com/course-demos/python/hrleaders.cfm> and print out the first 125 characters. The result will be something like:

```
<!doctype html>
<html lang="en">
<head>
  <title>Webucator</title>

  <meta name="robots" content="noindex,follow">
```

To see where this code is coming from...

1. Navigate to <https://www.webucator.com/course-demos/python/hrleaders.cfm> in your web browser.
2. Right-click on the page and select **View page source** or **View source**. You should see something like:



```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <title>Webucator</title>
5
6   <meta name="robots" content="noindex,follow">
7
8   <meta name="description" content="Webucator provides onsite
and live online training on technologies such as Microsoft
SharePoint, SQL Server and Office, Java, Adobe Dreamweaver
```

This is the code returned by the web server that the browser uses to draw the web page. Notice that the page begins with the text that our script output.

Custom Headers

A web server might choose to block any requests that don't identify the user agent. You may be able to get around this by passing in headers that include a value for "user-agent", like this:

```
r = requests.get(url, headers={'user-agent': 'my-app/0.0.1'})
```

The `text` attribute of the response object returned by `requests.get()` will hold the full code of the HTML page delivered from the specified URL. In this case the URL is:

<https://www.webucator.com/course-demos/python/hrleaders.cfm>.

When viewed in a browser, the page looks like this:

webucator.com/course-demos/python/hrleaders.ctm

find a course

webucator
customized INSTRUCTOR-LED training services

1-877-932-8228 CONTACT US

Course Catalog Training Options

Best Home run Seasons of the 20th Century

Steroids-era years are highlighted.

#	Player	Home runs	Year	Birthday
1	Mark McGwire	70	1998	1963-10-01
2	Sammy Sosa	66	1998	1968-11-12
3	Mark McGwire	65	1999	1963-10-01
4	Sammy Sosa	63	1999	1968-11-12
5	Roger Maris	61	1961	1934-09-10
6	Babe Ruth	60	1927	1895-02-06
7	Babe Ruth	59	1921	1895-02-06

HTML

Data in web pages is stored in HTML, a relatively simple markup language. Elements of an HTML page are marked up with tags. For example, a paragraph is started with an opening `<p>` tag and ended with a closing `</p>` tag. The page shown in the previous screenshot displays home-run records in a table. Each record is in a table row that is marked up like this:

```
<tr class="steroids-era">
  <td>1</td>
  <td>Mark McGwire</td>
  <td>70</td>
  <td>1998</td>
  <td>1963-10-01</td>
</tr>
```

- Table rows begin with an opening `<tr>` tag and end with a closing `</tr>` tag.
- Within table rows, table data cells begin with an opening `<td>`

tag and end with a closing `</td>` tag, and table header cells begin with an opening `<th>` tag and end with a closing `</th>` tag.

Tags often have *attributes* for further defining the element. Attributes usually come in name-value pairs.

Note that attributes only appear in the opening tag, like so:

```
<tagname att1="value" att2="value">Element content</tagname>
```

Some of the home-run table rows, like the one just shown, have a `class` attribute with the value of “steroids-era”:

```
<tr class="steroids-era">
```

It is common to search for elements by tag name and by class.

Writing code to extract data using string manipulation and regular expressions would be quite challenging. Beautiful Soup to the rescue!

Beautiful Soup

Beautiful Soup is a Python library for extracting HTML and XML data. It is often used to find specific content on a web page, a process known as “scraping.”

Install Beautiful Soup in your virtual environment:

```
pip install beautifulsoup4
```

You can choose between several options⁵⁵ for parsing the HTML content. BeautifulSoup recommends `lxml`. Install that as well:

lxml and Case

The `lxml` parser converts all elements and attributes to lowercase letters, so you should only search for lowercase tags and attributes.

```
pip install lxml
```

To get a feel for how BeautifulSoup works, start up Python at the terminal:

1. The first step is to import BeautifulSoup:

```
>>> from bs4 import BeautifulSoup
```

2. Import requests and get some data to work with:

```
>>> import requests
>>> url = 'https://www.webucator.com/course-demos/python/hrle
>>> r = requests.get(url)
>>> content = r.text
```

3. Using the content of that web page, create a BeautifulSoup object using the lxml parser:

```
>>> soup = BeautifulSoup(content, 'lxml')
```

4. Use the `find()`, `find_all()`, or `select()` methods to find the tags you are looking for.
 - `find()` returns a single `bs4.element.Tag` object, which we will just call a Tag object.
 - `find_all()` and `select()` each return a `bs4.element.ResultSet`, which is essentially a list of Tag objects.

The `find()` and `find_all()` methods provide a lot of options for finding tags. We'll use `find_all()` to illustrate, but `find()` works in the same way.

5. Get all the table rows (`tr` elements) in the content:

```
>>> trs = soup.find_all('tr')
>>> len(trs)
101
```

6. Because `find_all()` is the most common method used, there is a shortcut method for using it. You can simply treat `soup` as a method itself, like this:

```
>>> trs = soup('tr')
```

Find all the table data cells:

```
>>> tds = soup('td')
>>> len(tds)
500
```

7. BeautifulSoup `Tag` objects can also take the `find()`, `find_all()`, and `select()` methods. Find all the `td` elements in the first table row:

```
>>> first_row = trs[0]
>>> first_row.find_all('td')
[]
```

There are none, because the first row contains table header elements (`<th>` tags). Just as we can treat the main `soup` object as a method as a shortcut for `find_all()`, we can treat a `Tag` object as a method as well:

```
>>> first_row('th')
[<th>#</th>, <th>Player</th>, <th>Home runs</th>, <th>Year</t
```

8. As we mentioned earlier, it is common to search for elements by their tag name and class value. You can pass a value for the class as the second argument of `find_all()` and `find()`. The following code would find the first table row with the “steroids-era” class:

```
>>> soup.find('tr', 'steroids-era')
<tr class="steroids-era">
<td>1</td>
<td>Mark McGwire</td>
<td>70</td>
```

```
<td>1998</td>
<td>1963-10-01</td>
</tr>
```

9. You can also pass in a `text` argument to search for elements that contain certain text:

```
>>> ruths = soup.find_all('td', text='Babe Ruth')
>>> len(ruths)
9
```

0. Use the `parent` property to get a Tag's parent Tag:

```
>>> first_ruth_row = ruths[0].parent
>>> first_ruth_row
<tr>
<td>6</td>
<td>Babe Ruth</td>
<td>60</td>
<td>1927</td>
<td>1895-02-06</td>
</tr>
```

1. In addition to searching on the text of an element, you can access the text of Tag object using the `text` property:

```
>>> ruth_birth_day = first_ruth_row.find_all('td')[-1].text
>>> ruth_birth_day
'1895-02-06'
```

Other Attributes

In addition to finding elements by name and class value, you can find elements by any of their attributes. To do so, pass in name-value pairs when you create the `soup` object. For example, the following code would get all a elements (HTML links) with a `target` attribute set to

_blank:

```
soup.find_all('a', target='_blank')
```

The following file prompts the user for a URL and then searches the web page for links with a target of “_blank”:

Demo 116: working-with-data/Demos/a_blanks.py

```
import requests
from bs4 import BeautifulSoup

def get_content():
    url = input('Enter a URL: ')
    r = requests.get(url)
    return r.text

def main():
    content = get_content()
    soup = BeautifulSoup(content, 'lxml')

    external_links = soup.find_all('a', target='_blank')

    found = False
    for i, link in enumerate(external_links, 1):
        found = True
        print(f'{i}. {link}')

    if not found:
        print('None found.')

main()
```

Code Explanation

Run the file and enter the URL of your choice to see if it has any links that target “_blank”.

At the time of this writing, <https://www.stackoverflow.com> had some such links.

Finding Non-Section-508-Compliant Images

Images on HTML pages are created with the `` tag. People with impaired vision rely on the value of the `img` element's `alt` attribute to know what the image represents. All `img` elements on web pages should include `alt` attributes with values. You can check whether an attribute is included or not using a boolean value:

```
soup.find_all('img', alt=False)
```

The following file prompts the user for a URL and then searches the web page for `img` elements with no `alt` value:

Demo 117: working-with-data/Demos/imgs_wo_alt.py

```
import requests
from bs4 import BeautifulSoup

def get_content():
    url = input('Enter a URL: ')
    r = requests.get(url)
    return r.text

def main():
    content = get_content()
    soup = BeautifulSoup(content, 'lxml')

    images = soup.find_all('img', alt=False)

    found = False
    for i, img in enumerate(images, 1):
        found = True
        print(f'{i}. {img}')

    if not found:
        print('None found.')

main()
```

Code Explanation

Run the file and enter the URL of your choice to see if it has any images are missing the `alt` value.

At the time of this writing, <https://www.syracuse.com> had some such images.

Exercise 43: HTML Scraping

In this exercise, you will try to scrape data from the web page at:

<https://www.webucator.com/course-demos/python/hrleaders.cfm>

Working at the terminal or in a file, try to find:

1. The last cell of the last row of the table.
2. The number of records that occurred during the steroids era.
3. The name of the player with the most home runs in a single season. Note that the first `tr` element contains the column headings in `th` tags. That `tr` is contained within a `thead` element. The rows with the data in them are all contained within a single `tbody` element. You want to get the text in the second cell of the first row in the `tbody` element. The relevant HTML segment looks like this:

```
<tbody>
<tr class="steroids-era">
  <td>1</td>
  <td>Mark McGwire</td>
  <td>70</td>
  <td>1998</td>
  <td>1963-10-01</td>
</tr>...
```

4. The most home runs Willie Mays ever got in a season.
5. The list of player names who hit 50 or more home runs in a single season. Do this one in a Python file. It should print out the player names. There will be 17, beginning with:

1. Mark McGwire
2. Sammy Sosa
3. Roger Maris

Solution

First, get the BeautifulSoup object:

```
>>> import requests
>>> from bs4 import BeautifulSoup
>>> url = 'https://www.webucator.com/course-demos/python/hrleade
>>> r = requests.get(url)
>>> content = r.text
>>> soup = BeautifulSoup(content, 'lxml')
```

1. The last cell of the last row of the table:

```
>>> trs = soup.find_all('tr')
>>> trs[-1]('td')[-1]
<td>1936-08-08</td>
```

2. The number of records that occurred during the steroids era:

```
>>> len(soup.find_all('tr', 'steroids-era'))
29
```

3. The name of the player with the most home runs in a single season:

```
>>> data_container = soup.find('tbody')
>>> first_row = data_container.find('tr')
>>> player_name = first_row.find('td').text
>>> player_name = first_row.find_all('td')[1].text
>>> player_name
'Mark McGwire'
```

4. The most home runs Willie Mays ever got in a season:

```
>>> mays_first_td = soup.find('td', text='Willie Mays')
>>> mays_hr_record = mays_first_td.parent.find_all('td')[2].t
>>> mays_hr_record
'52'
```

5. The number of players who hit 50 or more home runs in a single season: See the following file:

Solution: working-with-data/Solutions/fifty_hrs.py

```
import requests
from bs4 import BeautifulSoup

# Constants to express which content is in which cells
NUM = 0
PLAYER = 1
HRS = 2
YEAR = 3
BIRTH_DAY = 4

def get_content():
    url = 'https://www.webucator.com/course-demos/python/hrleader'
    r = requests.get(url)
    return r.text

def get_soup(content):
    return BeautifulSoup(content, 'lxml')

def get_players(soup):
    data_container = soup.find('tbody')

    # Get all table rows in the tbody
    rows = data_container.find_all('tr')

    players = []
    # Loop through the rows
    for row in rows:
        # Get int value of HRS text
        hrs = int(row.find_all('td')[HRS].text)
        if hrs >= 50:
            player = row.find_all('td')[PLAYER].text
            # Add the name of the player to players
            # but only if they're not already in there
            if player not in players:
                players.append(player)
        if hrs < 50:
            return players # No need to keep looking
```

```
    return players # Just in case all have 50 or more hrs
```

```
def main():  
    content = get_content()  
    soup = get_soup(content)  
    players = get_players(soup)  
  
    for i, player in enumerate(players, 1):  
        print(f'{i}. {player}')  
main()
```

XML

XML (eXtensible Markup Language) is a meta-language; that is, it is a language in which other languages are created. In XML, data is "marked up" with tags, similar to HTML tags. In fact, one version of HTML, called XHTML, is an XML-based language, which means that XHTML follows the syntax rules of XML.

XML is used to store data or information; this data might be intended to be read by people or by machines. It can be highly structured data such as data typically stored in databases or spreadsheets, or loosely structured data, such as data stored in letters or manuals.

Beautiful Soup is also used to parse XML documents. `lxml` is the recommended parser for parsing XML as well as HTML:

```
soup = BeautifulSoup(content, 'lxml')
```

The home-runs page we have been working with has an XML version at:

<https://www.webucator.com/course-demos/python/hrleadersxml.cfm>

The XML for the page contains `record` elements, an abridged version of which is shown here:

```
<records>
  <record>
    <Player>Mark McGwire</Player>
    <HR>70</HR>
    <Year>1998</Year>
    <Birthday>1963-10-01</Birthday>
  </record>
  <record>
    <Player>Sammy Sosa</Player>
    <HR>66</HR>
    <Year>1998</Year>
    <Birthday>1968-11-12</Birthday>
  </record>
  ...
</records>
```

Note that each player name is stored in the `Player` tag, but `lxml` will convert all tags and attributes to lowercase letters, so we will search for “player”. Here is the code to list all players in the XML file:


```
import requests
from bs4 import BeautifulSoup

url = 'https://www.webucator.com/course-demos/python/hrleadersxml'
r = requests.get(url)
content = r.text

soup = BeautifulSoup(content, 'lxml')

players = soup.find_all('player')
for i, player in enumerate(players, 1):
    print(f'{i}. {player.text}')
```

JSON

JSON stands for **J**ava**S**cript **O**bject **N**otation. According to the [official JSON website⁵⁶](#), JSON is:

1. A lightweight data-interchange format.
2. Easy for humans to read and write.
3. Easy for machines to parse and generate.

Numbers 1 and 3 are certainly true. Number 2 depends on the type of human. Experienced Python programmers will find the JSON syntax extremely familiar as it uses the same syntax as Python's `dict` and `list` objects.

Here is an example of JSON holding weather data:

Demo 119: working-with-data/data/weather.json

```
{
  "list": [{
    "dt": 1581886800,
    "main": {
      "temp": 36.86,
      "feels_like": 29.97,
      "temp_min": 33.22,
      "temp_max": 36.86,
      "humidity": 98
    },
    "wind": {
      "speed": 7.11,
      "deg": 243
    },
    "dt_txt": "2020-02-16 21:00:00"
  }, {
    "dt": 1581897600,
    "main": {
      "temp": 30.65,
      "feels_like": 23.94,
      "temp_min": 27.93,
      "temp_max": 30.65,
      "humidity": 94
    },
    "wind": {
      "speed": 4.85,
      "deg": 254
    },
    "dt_txt": "2020-02-17 00:00:00"
  }],
  "city": {
    "id": 5140405,
    "name": "Syracuse",
    "country": "US",
    "population": 145170,
    "timezone": -18000,
    "sunrise": 1581854499,
```

```
    "sunset": 1581892559
  }
}
```

Code Explanation

This is a simplified version of the JSON returned from the [OpenWeatherMap API's current weather data](#)⁵⁷. As you can see, it is formatted just like a Python dict:

1. The value for the “list” key is a list of dicts, each representing the weather for the subsequent three-hour period.
 2. The value for the “city” key is a dict containing data about the city.
-

After assigning the dict to a `weather` variable, you could get the name of the city and the max temperature for the next period as follows:

```
city_name = weather['city']['name']
max_temp = weather['list'][0]['main']['temp_max']
```

Many organizations, including Google, Twitter, Facebook, Reddit, and Microsoft, provide APIs that return data in JSON.⁵⁸ Each API has its own rules and parameters. Most, including the OpenWeatherMap API, require an API key.

Getting an OpenWeatherMap API Key

The OpenWeatherMap API has a free tier, but it requires an API key. To get an API key, sign up at https://home.openweathermap.org/users/sign_up.

To get the forecast for Syracuse, NY in JSON format, go to this URL (replacing **yourapikey** with a valid API key):

```
https://api.openweathermap.org/data/2.5/forecast/?q=Syracuse, New
```

Notice the parameters passed in the URL:

1. **APPID** – a valid API key.

2. **q**: a string in the format:

```
city_name,state_name,country_code
```

3. **units**: the system of measurement (`imperial` for fahrenheit, `metric` for celsius)

The following code uses the API to report the forecasted max temperature in Syracuse, NY for the next ten days:

Demo 120: working-with-data/Demos/weather.py

```
import requests
from datetime import datetime

API_KEY = 'abca198b092b0295697beb48914a442c'
FEED = 'https://api.openweathermap.org/data/2.5/forecast/'

def main():
    city = 'Syracuse'
    state = 'New York'
    country_code = 'us'

    params = {
        'q': city + ',' + state + ',' + country_code,
        'units': 'imperial',
        'APPID': API_KEY
    }

    r = requests.get(FEED, params)
    print(r.url) # prints the URL created using the params

    weather = r.json()

    fmt_in = '%Y-%m-%d %H:%M:%S'
    fmt_out = '%A, %B %d, %Y at %I %p'
    for item in weather['list']:
        max_temp = item['main']['temp_max']
        dt = item['dt_txt']
        day_time = datetime.strptime(dt, fmt_in).strftime(fmt_out)
        print(f'High on {day_time} in {city}: {max_temp} fahrenheit')

main()
```

Code Explanation

The `json()` method of the response from `requests.get()` converts the

JSON code to a Python `dict`. From there, you can work with the Python `dict` object as you normally would.

Exercise 44: JSON Home Runs

20 to 30 minutes.

In this exercise, you will work with the JSON data at the following URL:

<https://www.webucator.com/course-demos/python/hrleadersjson.cfm>

1. Open a new file and save it as `json_hrs.py` in `working-with-data/Exercises`.
2. Using the data returned from <https://www.webucator.com/course-demos/python/hrleadersjson.cfm>, write code to print out the number of home runs each player in the JSON data hit. The first ten results are shown here:

1. Mark McGwire hit 70 home runs in 1998.
2. Sammy Sosa hit 66 home runs in 1998.
3. Mark McGwire hit 65 home runs in 1999.
4. Sammy Sosa hit 63 home runs in 1999.
5. Roger Maris hit 61 home runs in 1961.
6. Babe Ruth hit 60 home runs in 1927.
7. Babe Ruth hit 59 home runs in 1921.
8. Jimmie Foxx hit 58 home runs in 1932.
9. Hank Greenberg hit 58 home runs in 1938.
10. Hack Wilson hit 56 home runs in 1930.

Solution: working-with-data/Solutions/hrs_json.py

```
import requests

data="https://www.webucator.com/course-demos/python/hrleadersjson"

r = requests.get(data)
records = r.json()

for i, record in enumerate(records, 1):
    player = record['Player']
    hrs = record['HR']
    year = record['Year']
    print(f'{i}. {player} hit {hrs} home runs in {year}.')
```

Don't forget to deactivate your virtual environment:

```
deactivate
```

Conclusion

In this lesson, you have learned to work with data stored in databases, CSV files, HTML, XML, and JSON.

LESSON 16

Testing and Debugging

Topics Covered

- Testing performance with timers and the `timeit` module.
- The `unittest` module.

After testing both smiles and frowns, and proving that neither mode of treatment possessed any calculable influence, Hester was ultimately compelled to stand aside, and permit the child to be swayed by her own impulses.

– *The Scarlet Letter*, Nathaniel Hawthorne

In this lesson, you will learn to test the performance and the functionality of your Python code.

Testing for Performance

`time.perf_counter()`

The `time` module includes a `perf_counter()` method that is used to precisely measure relative times. The following code illustrates:

Demo 121: testing-debugging/Demos/time_diff.py

```
import time

t1 = time.perf_counter()
t2 = time.perf_counter()
print(t1, t2, t2-t1)
```

Code Explanation

Note that the value returned by `time.perf_counter()` is only meaningful when compared to other values returned by `time.perf_counter()` at different times. If we run the same file multiple times, `t1` and `t2` will have different values, but the time difference between them should be similar:

```
PS ...\\testing-debugging\\Demos> python time_diff.py
0.0178728 0.0178734 5.999999999999062e-07
PS ...\\testing-debugging\\Demos> python time_diff.py
0.0167957 0.0167963 5.999999999999062e-07
PS ...\\testing-debugging\\Demos> python time_di
```

Scientific Notation

5.999999999999062e-07 seconds is a very small amount of time. The number is displayed in scientific notation and is the equivalent of 0.000000599999999999062. That's approximately .00006 milliseconds. If you need to brush up on your scientific notation, check out the short video at <https://youtu.be/JrbvjFqFITI>.

The following code shows how to use `time.perf_counter()` to compare how fast two pieces of code run:

Demo 122: testing-debugging/Demos/compare_code_speed1.py

```
import random
import time

start_time = time.perf_counter()
numbers = str(random.randint(1, 100))
for i in range(1000):
    num = random.randint(1, 100)
    numbers += ',' + str(num)
end_time = time.perf_counter()
td1 = end_time - start_time

start_time = time.perf_counter()
numbers = [str(random.randint(1, 100)) for i in range(1, 1000)]
numbers = ','.join(numbers)
end_time = time.perf_counter()
td2 = end_time - start_time

print(f"""Time Delta 1: {td1}
Time Delta 2: {td2}""")
```

Code Explanation

Both snippets of code create strings from 1,000 random numbers between 1 and 100.

1. The first snippet (lines 5-8) uses the `+=` operator to repeatedly append to the `numbers` string.
2. The second snippet (lines 13-14) uses a list comprehension to create a list of random numbers and then uses the `join()` method to convert the list to a string.

We capture the time before and after each code snippet and then print the results. Here are the results from running it four times:

```
PS ...\\testing-debugging\\Demos> python time_diff.py
PS ...\\testing-debugging\\Demos> python compare_code_speed1.py
```

```
Time Delta 1: 0.0011696999999999992
Time Delta 2: 0.00095860000000000004
PS ...\\testing-debugging\\Demos> python compare_code_speed1.py
Time Delta 1: 0.00116600000000000004
Time Delta 2: 0.00093690000000000009
PS ...\\testing-debugging\\Demos> python compare_code_speed1.py
Time Delta 1: 0.00112109999999999999
Time Delta 2: 0.00093729999999999986
PS ...\\testing-debugging\\Demos> python compare_code_speed1.py
Time Delta 1: 0.00114879999999999984
Time Delta 2: 0.00093160000000000012
```

To get an ever more accurate comparison of the efficiency of the two methods, we can run each snippet through a loop 1,000 times and then compare the results:

Demo 123: testing-debugging/Demos/compare_code_speed2.py

```
import random
import time

start_time = time.perf_counter()
for j in range(1000):
    numbers = str(random.randint(1, 100))
    for i in range(1000):
        num = random.randint(1, 100)
        numbers += ',' + str(num)
end_time = time.perf_counter()
td1 = end_time - start_time

start_time = time.perf_counter()
for j in range(1000):
    numbers = [str(random.randint(1, 100)) for i in range(1, 1000)]
    numbers = ', '.join(numbers)
end_time = time.perf_counter()
td2 = end_time - start_time

print(f"""Time Delta 1: {td1}
Time Delta 2: {td2}""")
```

Code Explanation

Here is the output of this file:

```
PS ...\\testing-debugging\\Demos> python compare_code_speed2.py
Time Delta 1: 1.2513799
Time Delta 2: 0.8971576000000001
```

The `timeit` Module

Python comes with a built-in `timeit` module that does everything we just saw (and more) for you. The two methods you will use most often

are:

1. `timeit.timeit()`
2. `timeit.repeat()`

`timeit.timeit()`

The `timeit()` method can take multiple parameters, but the most important are:

1. `stmt` – The statement to run.
2. `setup` – Any code that should be run before the time testing begins. The time it takes to execute this code will **not** be included in the results.
3. `number` – The number of times to run it. This argument is optional, but it defaults to 1000000, which could take an awful long time to run.

It returns the number of seconds it took to run `stmt` `number` times.

Often, you will have to rewrite a piece of code to test it using `timeit`. One way of doing this is to create temporary functions to hold the different pieces of code you want to compare. Here's an example:

Demo 124: testing-debugging/Demos/using_timeit1.py

```
import random
from timeit import timeit

def string_nums1():
    numbers = str(random.randint(1, 100))
    for i in range(1000):
        num = random.randint(1, 100)
        numbers += ', ' + str(num)

def string_nums2():
    numbers = [str(random.randint(1, 100)) for i in range(1, 1000)]
    numbers = ', '.join(numbers)

td1 = timeit(string_nums1, number=1000)
td2 = timeit(string_nums2, number=1000)

print("Results from using timeit()")
print(td1, td2, sep="\n")
print('-' * 70)

print('string_nums1 compared to string_nums2:')
print(f'{td1/td2:.2%}')
```

Code Explanation

This will show that the `string_nums1()` function is about 30% less efficient than the `string_nums2()` function:

```
PS ...\\testing-debugging\\Demos> python using_timeit1.py
Results from using timeit()
1.1584021
0.8860760999999997
-----
string_nums1 compared to string_nums2:
130.73%
```

Note that, because the two `string_nums` functions are both defined in the *global namespace*, they have access to the imported `random` module, so we do not need to import that via the `setup` parameter.

Namespaces

In Python, a *namespace* is a *names-to-objects mapping*. Namespaces are closely related to scope and are used to distinguish between identically named attributes and variables defined in different modules. The namespace of the top-level of the running module is *globals*. And the namespace of each imported module is the name of that module. Functions defined within a module have their own *local* namespaces.

Another way to do this is to place the code you want to test in strings and then import `random` in the `setup` argument, like this:

Demo 125: testing-debugging/Demos/using_timeit2.py

```
from timeit import timeit

str_nums1 = """
numbers = str(random.randint(1, 100))
for i in range(1000):
    num = random.randint(1, 100)
    numbers += ', ' + str(num)"""

str_nums2 = """
numbers = [str(random.randint(1, 100)) for i in range(1, 1000)]
numbers = ', '.join(numbers)"""

td1 = timeit(str_nums1, number=1000, setup='import random')
td2 = timeit(str_nums2, number=1000, setup='import random')
-----Lines Omitted-----
```

In this code, the code in the strings is executed within `timeit`'s namespace, so we need to use `setup` to import `random` into that namespace.

A third way as of Python 3.5 is specify the `globals` namespace, which will give the code run by `timeit()` access to global attributes and imported modules:

Demo 126: testing-debugging/Demos/using_timeit3.py

```
import random
from timeit import timeit

str_nums1 = """
numbers = str(random.randint(1, 100))
for i in range(1000):
    num = random.randint(1, 100)
    numbers += ', ' + str(num)"""

str_nums2 = """
numbers = [str(random.randint(1, 100)) for i in range(1, 1000)]
numbers = ', '.join(numbers)"""

td1 = timeit(str_nums1, number=1000, globals=globals())
td2 = timeit(str_nums2, number=1000, globals=globals())
-----Lines Omitted-----
```

All of these methods are fine and should yield similar results.

```
timeit.repeat()
```

The `repeat()` method is similar to the `timeit()` method, but it runs the loop multiple times and returns a list with the results of each repetition. In addition to `stmt`, `setup`, and `number`, the `repeat()` method has a `repeat` parameter, which takes the number of times to repeat the loop. The default value of `repeat` is 5.[59](#)

Here is the previous example using `repeat()` instead of `timeit()`:

Demo 127: testing-debugging/Demos/using_repeat.py

```
import random
from timeit import repeat

str_nums1 = """
numbers = str(random.randint(1, 100))
for i in range(1000):
    num = random.randint(1, 100)
    numbers += ', ' + str(num)"""

str_nums2 = """
numbers = [str(random.randint(1, 100)) for i in range(1, 1000)]
numbers = ', '.join(numbers)"""

tds1 = repeat(str_nums1, number=1000, repeat=4, globals=globals())
tds2 = repeat(str_nums2, number=1000, repeat=4, globals=globals())

print("Results from using repeat()")
print(tds1, tds2, sep="\n")
print('-' * 70)

print('str_nums1 compared to str_nums2:')
print(f'{sum(tds1)/sum(tds2):.2%}')
```

Code Explanation

This will output something like this:

```
PS ...\\testing-debugging\\Demos> python using_repeat.py
Results from using repeat()
[1.0138493, 1.302279, 1.1477145000000002, 1.2157394999999998]
[0.8648758000000001, 0.8432685000000006, 0.8369980000000004, 0.8369980000000004]
-----
str_nums1 compared to str_nums2:
138.37%
```

After four iterations of looping through each approach 1,000 times, it's

pretty clear that the second approach is faster.

Using `timeit` Interactively

You may find that using `timeit()` interactively is more convenient for small snippets of code.

Let's try the code that generates a list of 1,000 random integers in the range of 1 to 100:

```
>>> import random
>>> from timeit import timeit
>>> timeit(' '.join([str(random.randint(1, 100)) for i in range
0.008025599999999855
```

Your output will likely be slightly different from the output shown above.

Exercise 45: Comparing Times to Execute

10 to 15 minutes.

1. Open `testing-debugging/Exercises/compare_code.py` in your editor.
2. Notice that the functions use random words rather than receiving a word as a parameter as they normally would.
3. Write code to compare the time it takes to run the different functions.

Exercise Code: testing-debugging/Exercises/compare_code.py

```
import re
import random

def get_word():
    words = ['Charlie', 'Woodstock', 'Snoopy', 'Lucy', 'Linus',
            'Schroeder', 'Patty', 'Sally', 'Marcie']
    return random.choice(words).upper()

def green_glass_door_1():
    word = get_word()
    prev_letter = ''
    for letter in word:
        letter = letter.upper()
        if letter == prev_letter:
            return True
        prev_letter = letter
    return False

def green_glass_door_2():
    word = get_word()
    pattern = re.compile(r'(\.)\1')
    return pattern.search(word)
```

Solution: testing-debugging/Solutions/compare_code.py

```
import re
import random
from timeit import repeat

def get_word():
    words = ['Charlie', 'Woodstock', 'Snoopy', 'Lucy', 'Linus',
            'Schroeder', 'Patty', 'Sally', 'Marcie']
    return random.choice(words).upper()

def green_glass_door_1():
    word = get_word()
    prev_letter = ''
    for letter in word:
        letter = letter.upper()
        if letter == prev_letter:
            return True
        prev_letter = letter
    return False

def green_glass_door_2():
    word = get_word()
    pattern = re.compile(r'(\.)\1')
    return pattern.search(word)

tds1 = repeat(green_glass_door_1, number=1000, repeat=4, globals:
tds2 = repeat(green_glass_door_2, number=1000, repeat=4, globals:

print(tds1, tds2, sep="\n")
print('-' * 70)

print('green_glass_door_1 compared to green_glass_door_2:')
print('{:.2%}'.format(sum(tds1)/sum(tds2)))
```

Code Explanation

Here are our results:

```
[0.00122430000000000046, 0.00119200000000000056, 0.0011614999999999999]
[0.00168240000000000006, 0.0017012, 0.00139999999999999985, 0.0017012]
-----
green_glass_door_1 compared to green_glass_door_2:
72.12%
```

It appears that the regular expression version is less efficient than the other.

When Efficiency Matters

You don't need to test all your code for efficiency. A little bit faster is not always a little bit better, especially if it comes at the cost of code clarity. We recommend that you follow this procedure:

1. Get your code working.
2. Clean up your code to make it as readable as possible. This includes adding comments.
3. **If** your code has an efficiency problem, use `timeit` to identify the source of the problem and fix it.

This is not to say that you shouldn't be concerned with efficiency while coding. If you already know one method is significantly more efficient than another, all else being equal, use the more efficient method. Just don't let the quest for efficiency slow you down.

The unittest Module

The built-in `unittest` module provides a framework for writing unit tests by extending the `unittest.TestCase` class. It is best understood through an example.

We'll start with three functions, the third of which has an error:

```
def prepend(s, c):  
    return c + s  
  
def append(s, c):  
    return s + c  
  
def insert(s, c, pos):  
    return s[0:pos] + c + s[pos:-1] # wrong
```

Classes

In this lesson, we will be discussing *classes*. You will learn to create your own classes in the [Classes and Objects lesson](#). In that lesson, you will write tests for the classes you create. So, we have a *catch 22*: it is helpful to understand classes when learning to write tests, but it is also helpful to understand testing when learning to write classes. Fortunately, you only need to know a little about classes for the rest of this lesson:

1. Classes are created using the `class` keyword.
2. Classes are named in upper camel case (e.g., `TestCase`).
3. Classes generally have methods, which are functions tied to the class.
4. Classes refer to themselves (or rather to the objects they create) as `self`, and they call their own methods with `self.method_name()`.

Don't get hung up on the class syntax for now. Look to understand the methods (functions) within the class.

To test our functions using the `unittest` module, we must do the following (at a minimum):

1. Import the `unittest` module.
2. Create a class that extends `unittest.TestCase`.

3. Write test methods. Note that the names of the test methods must start with the string “test”.

To run the tests, call `unittest.main()`, but we only want to do that when we run the test file directly, so will put it in the [following conditional](#):

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

Here is a class to test the three functions above along with code to create and run the suite of tests:

Demo 129: testing-debugging/Demos/unit_test_functions.py

```
import unittest
from functions_error import prepend, append, insert

class TestMyMethods(unittest.TestCase):
    def test_prepend(self):
        self.assertEqual(prepend('bar', 'foo'), 'foobar')
    def test_append(self):
        self.assertEqual(append('bar', 'foo'), 'barfoo')
    def test_insert(self):
        self.assertEqual(insert('wetor', 'buca', 2), 'webucator')

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

When we run the test suite, we get the following result:

```
PS ...\\testing-debugging\\Demos> python unit_test_functions.py
.F.
=====
FAIL: test_insert (__main__.TestMyMethods)
-----
Traceback (most recent call last):
  File "...\\unit_test_functions.py", line 10, in test_insert
    self.assertEqual(insert('wetor', 'buca', 2), 'webucator')
AssertionError: 'webucato' != 'webucator'
- webucato
+ webucator
?          +

-----

Ran 3 tests in 0.001s

FAILED (failures=1)
```

This shows that three tests ran and one failed. And it gives details on

the test that failed. Our code expected `insert('weter', 'buca', 2)` to equal “webucator”, but instead it resulted in “webucato”. That means there is something wrong with our `insert()` function. **Can you fix it?**

Unittest Test Files

You can run many test files at once from the command line with the following command:

```
python -m unittest discover directory_with_tests
```

That will search the directory_with_tests folder for Python files with names that begin with “test”. For example, in the testing-debugging/Demos folder, there are two files with helper functions in them:

1. math_functions.py
2. string_functions.py

Each of these has an associated unit test file:

1. test_math_functions.py
2. test_string_functions.py

All four files are shown here:

Demo 130: testing-debugging/Demos/math_functions.py

```
import math

def round_down(f):
    return int(f) # doesn't work for negative numbers

def round_up(f):
    return math.ceil(f)
```

Code Explanation

Note that the `int()` function does not round down. It strips the decimal portion of the number, leaving just the integer. For negative numbers, this effectively rounds up (e.g., `-5.4` becomes `-5`). We should have used `math.floor()` instead.

Demo 131: testing-debugging/Demos/string_functions.py

```
import random
import string
import re

def prepend(s, c):
    return c + s

def append(s, c):
    return s + c

def insert(s, c, pos):
    return s[0:pos] + c + s[pos:-1] # wrong
```

Code Explanation

Recall that `s[first_pos:last_pos]` returns a slice that starts with the character at `first_pos` and includes all the characters up to *but not including* the character at `last_pos`. In `insert()`, we should have used `s[pos:]` to get a slice that includes the last character of the original string.

Demo 132: testing-debugging/Demos/test_math_functions.py

```
import unittest
from math_functions import *

class TestMathFunctions(unittest.TestCase):

    def test_round_down(self):
        self.assertEqual(round_down(1.3), 1)
        self.assertEqual(round_down(-1.3), -2)
        self.assertEqual(round_down(1.7), 1)
        self.assertEqual(round_down(-1.7), -2)
        self.assertEqual(round_down(0), 0)

    def test_round_up(self):
        self.assertEqual(round_up(1.3), 2)
        self.assertEqual(round_up(-1.3), -1)
        self.assertEqual(round_up(1.7), 2)
        self.assertEqual(round_up(-1.7), -1)
        self.assertEqual(round_up(0), 0)

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

Code Explanation

Notice that this file imports all the functions from the `math_functions` module.

Demo 133: testing-debugging/Demos/test_string_functions.py

```
import unittest
from string_functions import *

class TestStringFunctions(unittest.TestCase):

    def test_prepend(self):
        self.assertEqual(prepend('bar', 'foo'), 'foobar')

    def test_append(self):
        self.assertEqual(append('bar', 'foo'), 'barfoo')

    def test_insert(self):
        self.assertEqual(insert('wetor', 'buca', 2), 'webucator')

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

Code Explanation

Notice that this file imports all the functions from the `string_functions` module.

To discover and run all the unit tests in the testing-debugging/Demos folder, open testing-debugging at the terminal and run:

```
python -m unittest discover Demos
```

Or, if you want to see a list of all the tests that run, include the `-v` option, like this:

```
python -m unittest discover Demos -v
```

Here are the results with the `-v` option included (run in Windows PowerShell):

```

PS ...\\Python\\testing-debugging> python -m unittest discover Demos
test_round_down (test_math_functions.TestMathFunctions) ... FAIL
test_round_up (test_math_functions.TestMathFunctions) ... ok
test_append (test_string_functions.TestStringFunctions) ... ok
test_insert (test_string_functions.TestStringFunctions) ... FAIL
test_prepend (test_string_functions.TestStringFunctions) ... ok

=====
FAIL: test_round_down (test_math_functions.TestMathFunctions)
-----
Traceback (most recent call last):
  File "C:\\Users\\ndunn\\OneDrive\\Documents\\Webucator\\Courseware\\c
    self.assertEqual(round_down(-1.3), -2)
AssertionError: -1 != -2

=====
FAIL: test_insert (test_string_functions.TestStringFunctions)
-----
Traceback (most recent call last):
  File "C:\\Users\\ndunn\\OneDrive\\Documents\\Webucator\\Courseware\\c
    self.assertEqual(insert('weto', 'buca', 2), 'webucator')
AssertionError: 'webucato' != 'webucator'
- webucato
+ webucator
?          +

-----
Ran 5 tests in 0.007s

FAILED (failures=2)

```

This shows that five tests were run and two failed.

Exercise 46: Fixing Functions

10 to 15 minutes.

In this exercise, you will correct the functions that failed the unit tests

in our demos. The same files we used in the demos are in the testing-debugging/Exercises folder.

1. At the terminal, run the command to discover and run unit tests in the testing-debugging/Exercises folder.
2. Open testing-debugging/Exercises/math_functions.py.
 - A. Fix the function that failed the unittest.
3. Open testing-debugging/Exercises/string_functions.py.
 - A. Fix the function that failed the unittest.
4. Run the command to discover and run unit tests again.
5. If any unit tests failed, go back and fix the associated functions.

Challenge

Try writing your own simple function and an associated unit test.

Solution: testing-debugging/Solutions/string_functions.py

```
import random
import string
import re

def prepend(s,c):
    return c + s

def append(s,c):
    return s + c

def insert(s,c,pos):
    return s[0:pos] + c + s[pos:]
```

Solution: testing-debugging/Solutions/math_functions.py

```
import math

def round_down(f):
    return math.floor(f)

def round_up(f):
    return math.ceil(f)
```

Special unittest.TestCase Methods

The `unittest.TestCase` class includes special `setUp()` and `tearDown()` methods that run before and after each test. You can use them to:

1. Instantiate (and clean up) class instances to use in tests.
2. Open and close files, database connections, network connections, etc.
3. Start and/or stop any server processes needed to run the tests.

These methods (and others like them) create the working environment for the tests. This working environment is called a *fixture*. To see how these methods work:

1. Open [testing-debugging/Demos/beatles/beatles.py](#) in your editor and review the code.
2. Open [testing-debugging/Demos/beatles/test_beatles.py](#) in your editor. Note that the `TestBeatles` class contains two test methods:
 - A. `test_select()` – checks to make sure the `select()` method returns a list.
 - B. `test_select_one()` – checks to make sure the `select()` method returns a tuple.
3. The `TestBeatles` class also contains `setUp()` and `tearDown()` methods, which run before and after each test.
 - A. The `setUp()` method instantiates a `beatles` object and runs the

`create()` and `insert()` methods to create and populate the `beatles` table. It also prints “Setting up”.

- B. The `tearDown()` method runs the `close()` method to close the cursor and connection. It also prints “Tearing down”.

4. Run the `test_beatles.py` file. It should output the following:

```
Setting up
Tearing down
.Setting up
Tearing down
.
-----
Ran 2 tests in 0.003s

OK
```

You wouldn’t usually print “Setting up” and “Tearing down”. We do this only to show that the fixture methods get called once for each test.

Methods for Creating Fixtures

There are additional methods for creating fixtures:

- The `setUpClass()` and `tearDownClass()` methods are used to run code before any of the tests in a `unittest.TestCase` class begins and after they all end. These must be implemented as class methods.
- The `setUpModule()` and `tearDownModule()` methods are used to run code at the beginning and end of the module containing test cases. These are module-level functions that are not part of any `unittest.TestCase` class. They are usually defined at the top of the module containing one or more `unittest.TestCase` classes.

Assert Methods

The `assertEqual()` method we used in our examples is by far the most common, but there are many other assert methods available in `TestCase` classes. For full documentation, see <https://docs.python.org/3/library/unittest.html>.

Conclusion

In this lesson, you have learned to test the performance of different pieces of code and to create unit tests to test your Python code.

LESSON 17

Classes and Objects

Topics Covered

- Classes and objects in Python.
- Instance methods, class methods, and static methods.
- Properties.
- Decorators.
- Subclasses and inheritance.

*Every object she saw, the moment she crossed the threshold,
appeared to delight her...*

– Wuthering Heights, Emily Bronte

An object is something that has attributes and/or behaviors, meaning it *is* certain ways and *does* certain things. In the real world, everything could be considered an object. Some objects are tangible, like rocks, trees, tennis racquets, and tennis players. And some objects are intangible, like words, colors, tennis swings, and tennis matches. In this lesson, you will learn how to write object-oriented Python code.

Attributes

If you can say “x is y” or “x has y,” then x is an object, and y is an attribute of x. Some examples:

1. *The rock that he is holding is heavy.* Heavy is an attribute of the specific rock he is holding. More generally, rocks have weight.
2. *The apple tree in our backyard has four branches.* The four branches are attributes of that specific tree. More generally, trees have branches.
3. *Venus Williams’ swing is strong.* Strong is an attribute of Venus

Williams' swing. More generally, tennis swings have a strength.

4. *The final match* had three *sets*. The three sets are attributes of the specific match. More generally, matches have sets.
5. *Serena Williams* has a *first-serve percentage of 57.2%*. A 57.2% first-serve percentage is an attribute of Serena Williams. More generally, tennis players have a first-serve percentage.

Attributes are generally nouns (e.g., branches) or adjectives (e.g., heavy). In Python, we could write the statements above like this:

```
rock_he_holds.weight = 'heavy'  
backyard_apple_tree.branches = [branch1, branch2, branch3, branch4]  
venus.swing = 'strong'  
final_match.sets = [set1, set2, set3]  
serena.serve1 = .572
```

Behaviors

If you can say “x does” then does is a behavior of x. Some examples of behaviors:

1. *The rock falls* fast. Rocks can fall.
2. *The apple tree in our back yard* first *bore fruit* on August 23, 2002. Trees can bear fruit.
3. Venus Williams' *swing hit* the ball. Tennis swings can hit things.
4. The final *match ended* at 11:45 in the morning. Matches can end.
5. *Serena Williams served*. Tennis players can serve.

Behaviors are verbs and behaviors of objects are called *methods*, which are simply functions defined within a class definition. In Python, we could write the statements above like this:

```
rock_he_holds.fall('fast')  
backyard_apple_tree.bear_fruit(datetime.date(2002, 8, 23))  
venus.swing.hit(ball)  
final_match.end(datetime.time(11, 45))  
serena.serve()
```

Classes vs. Objects

A class is a template for an object. An object is an *instance* of a class. When we say Serena Williams is a tennis player, we are saying that Serena Williams is an object of the `TennisPlayer` class. There are other tennis players who have the same attributes and behaviors as Serena Williams, but not in the same way. For example, Serena has a winning percentage. Her sister Venus also has a winning percentage. So do Roger Federer and Rafael Nadal. But their winning percentages are all different. They also all have backhands, but they don't all have the same backhand. Roger Federer has a one-handed backhand, while the others all have two-handed backhands. If you needed to express that in code, you could do it this way:

```
serena.two_handed_backhand = True
venus.two_handed_backhand = True
roger.two_handed_backhand = False
rafa.two_handed_backhand = True
```

In programming, we use classes to define what attributes and behaviors an object has or can have. In Python, we do this with the `class` keyword, like this:

```
class Player:
    pass
```

We then create an instance of that class like this:

```
serena = Player()
```

Everything Is an Object

Before we go further with this, let's take a look at some of the classes and objects we have already worked with in Python. In Python, everything is an object. Strings are objects, lists are objects, integers are objects, functions are objects. Even classes themselves are

objects. Python includes two built-in functions that help identify the class of an object:

- `type(obj)` – Returns the object's type, which is essentially a synonym for class.
- `isinstance(obj, class_type)` – Returns `True` if `obj` is an instance of `class_type`. Otherwise, returns `False`.

Take a look at the following examples:

```
>>> type('Hello')
<class 'str'>
>>> isinstance('Hello', str)
True
>>> type(1)
<class 'int'>
>>> isinstance(1, int)
True
>>> type(['a','b','c'])
<class 'list'>
>>> isinstance(['a','b','c'], list)
True
>>> type((1,2,3))
<class 'tuple'>
>>> isinstance((1,2,3), tuple)
True
>>> type({'a':1, 'b':2})
<class 'dict'>
>>> isinstance({'a':1, 'b':2}, dict)
True
```

The `print()` function is an instance of the `builtin_function_or_method`:

```
>>> type(print)
<class 'builtin_function_or_method'>
```

There are some built-in functions that are really classes and not

functions. Examples are `tuple`, `list`, `dict`, and `range`. The following code shows that `age` is of type `range` and `range` is of type `type` (meaning that it is a class):

```
>>> age = range(0, 100)
>>> type(age)
<class 'range'>
>>> type(range)
<class 'type'>
```

Again, *type* is just a synonym for *class*.[60](#)

Creating Custom Classes

Now, let's return to our `Player` class:

```
class Player:
    pass
```

By convention, classes are named using upper camel case (e.g., `MyClass`).[61](#) Check out the following:

```
>>> class Player:
...     pass
...
>>> serena = Player()
>>> type(serena)
<class '__main__.Player'>
>>> type(Player)
<class 'type'>
>>> isinstance(serena, Player)
True
```

This shows:

1. That the `Player` class is of type `type`.
2. That the instance of `Player` is of type `__main__.Player`. The

`__main__` tells us that the class was defined at the top level. Don't worry about that for now.

3. That `serena` is an instance of `Player`.

Attributes and Methods

Let's see how to create a class that creates dice objects:

Demo 134: classes-objects/Demos/Die1.py

```
class Die:
    pass
```

Code Explanation

We could import this class and instantiate a new `Die` object like this:

```
import Die1

die = Die1.Die()
```

Or we could import the `Die` class directly:

```
from Die1 import Die

die = Die()
```

Try this yourself by opening a Python prompt at [classes-objects/Demos](#) and running the code:

```
>>> import Die1
>>> die = Die1.Die()
>>> die
<Die1.Die object at 0x00000212F7800AF0>
>>> from Die1 import Die
>>> die = Die()
>>> die
<Die1.Die object at 0x00000212F79DAB20>
```

This `Die` class doesn't currently define any attributes or methods, so it's not very useful. The appropriate attributes and methods would depend on what type of application we want to build. No matter what the application is though, we are going to want to include an *initialization method*. An initialization method is a special method in

Python, `__init__()`, that is automatically called when an object is instantiated (i.e., created). The purpose of the `__init__()` method is to set the initial attributes of the new object.

Double Underscores

Note that the double underscores surrounding `init` indicate that this is a special method (a *magic method*) in Python. You should never name your own methods in this way.

For now, let's initialize `Die` objects with a single attribute: `sides` with a default value of 6:

```
class Die:
    def __init__(self, sides=6):
        self.sides = sides
```

Code Explanation

The first parameter of every standard method defined in a class, including the `__init__()` method, is `self`,⁶² which is a reference to the object being created. Methods can have any number of additional parameters. Our `__init__()` method takes one additional parameter: `sides`, which it assigns to `self.sides`. When we create a `Die` object, we do not pass in anything for `self`; the object itself gets passed in automatically. We just pass in a value for `sides`. The result is that our new `Die` object has a `sides` attribute, which holds an integer.

Forgetting Your self

If you are new to creating your own classes, it may take you a while to get used to including `self` as the first parameter of your methods. If you forget to include it, you will get an error similar to the following when you try to call that method on your class instance:

```
xyz_method() takes 0 positional arguments but 1 was given
```

Look at the class definition for `Die` again and notice that in the `__init__()` method we assign `sides` to `self.sides`. Remember that `self` is the object created by the class. The value we pass in for `sides` is assigned to the `sides` attribute of `self` as the following test code demonstrates:

Demo 136: classes-objects/Demos/test_Die2.py

```
import unittest
from Die2 import Die

class TestDieFunctions(unittest.TestCase):

    def setUp(self):
        self.die1 = Die()
        self.die2 = Die(8)

    def test_init(self):
        self.assertEqual(self.die1.sides, 6)
        self.assertEqual(self.die2.sides, 8)

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

Code Explanation

Run this test file to see how it works:

```
PS ...\\classes-objects\\Demos> python test_Die2.py -v
test_init (__main__.TestDieFunctions) ... ok
```

```
-----
Ran 1 test in 0.000s
```

```
OK
```

Both of our asserts passed, meaning that `self.sides` contains what we expect it to.

Derived Attribute Values

It might not always be as straightforward as directly assigning a passed-in argument to an attribute. Consider this class definition for

Circle:

Demo 137: classes-objects/Demos/Circle1.py

```
import math

class Circle:
    def __init__(self, val, prop='r'):
        if prop == 'r': # radius
            self.radius = val
        elif prop == 'd': # diameter
            self.radius = val / 2
        elif prop == 'c': # circumference
            self.radius = val / (2 * math.pi)
        elif prop == 'a': # area
            self.radius = (val / math.pi) ** .5
        else:
            raise Exception('prop must be r, d, c, or a')

        self.diameter = self.radius * 2
        self.circumference = self.radius * 2 * math.pi
        self.area = self.radius ** 2 * math.pi
```

Code Explanation

In this case, a `Circle` object is initialized with two parameters: `val` and `prop`, neither of which is directly assigned to the object. Rather, we use the values passed in to determine four of the object's attributes:

1. `radius`
 2. `diameter`
 3. `circumference`
 4. `area`
-

Let's test the class with the following test code:

Demo 138: classes-objects/Demos/test_Circle1.py

```
import unittest
import math
from Circle1 import Circle

class TestCircleFunctions(unittest.TestCase):

    def setUp(self):
        self.circle_r = Circle(5, 'r')
        self.circle_d = Circle(10, 'd')
        self.circle_c = Circle(10 * math.pi, 'c')
        self.circle_a = Circle(25 * math.pi, 'a')

    # Test circle_r
    def test_circle_radius(self):
        self.assertEqual(self.circle_r.radius, 5)
        self.assertEqual(self.circle_r.diameter, 10)
        self.assertEqual(self.circle_r.circumference, 10 * math.pi)
        self.assertEqual(self.circle_r.area, 25 * math.pi)

    # Test circle_d
    def test_circle_diameter(self):
        self.assertEqual(self.circle_d.radius, 5)
        self.assertEqual(self.circle_d.diameter, 10)
        self.assertEqual(self.circle_d.circumference, 10 * math.pi)
        self.assertEqual(self.circle_d.area, 25 * math.pi)

    # Test circle_c
    def test_circle_circumference(self):
        self.assertEqual(self.circle_c.radius, 5)
        self.assertEqual(self.circle_c.diameter, 10)
        self.assertEqual(self.circle_c.circumference, 10 * math.pi)
        self.assertEqual(self.circle_c.area, 25 * math.pi)

    # Test circle_a
    def test_circle_area(self):
        self.assertEqual(self.circle_a.radius, 5)
        self.assertEqual(self.circle_a.diameter, 10)
```

```
        self.assertEqual(self.circle_a.circumference, 10 * math.pi)
        self.assertEqual(self.circle_a.area, 25 * math.pi)

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

Code Explanation

Running this file should return the following, indicating that the class calculates the attribute values correctly:

```
PS ...\\classes-objects\\Demos> python test_Circle1.py -v
test_circle_area (__main__.TestCircleFunctions) ... ok
test_circle_circumference (__main__.TestCircleFunctions) ... ok
test_circle_diameter (__main__.TestCircleFunctions) ... ok
test_circle_radius (__main__.TestCircleFunctions) ... ok

-----
Ran 4 tests in 0.000s

OK
```

Let's now add a `resize_by()` method to our `Circle` class:

Demo 139: classes-objects/Demos/Circle2.py

```
import math

class Circle:
    def __init__(self, val, prop='r'):
        if prop == 'r':
            self.radius = val
        elif prop == 'd':
            self.radius = val / 2
        elif prop == 'c':
            self.radius = val / (2 * math.pi)
        elif prop == 'a':
            self.radius = (val / math.pi) ** .5
        else:
            raise Exception('prop must be r, d, c, or a')

        self.diameter = self.radius * 2
        self.circumference = self.radius * 2 * math.pi
        self.area = self.radius ** 2 * math.pi

    def resize_by(self, amount):
        self.radius *= (1 + amount)
        self.diameter = self.radius * 2
        self.circumference = self.radius * 2 * math.pi
        self.area = self.radius ** 2 * math.pi
```

Code Explanation

Again, we pass `self` into the method. We also pass in an `amount`. We use the value of `amount` to change `radius` and then we change the other attributes based on the new `radius`.

Let's test the class with the following test code:

Demo 140: classes-objects/Demos/test_Circle2.py

```
import unittest
import math
from Circle2 import Circle

class TestCircleFunctions(unittest.TestCase):

    def setUp(self):
        -----Lines Omitted-----
        self.circle_resized = Circle(5, 'r')
        self.circle_resized.resize_by(.5) # grow by 50%
        -----Lines Omitted-----
    def test_circle_resized(self):
        self.assertEqual(self.circle_resized.radius, 7.5)
        self.assertEqual(self.circle_resized.diameter, 15)
        self.assertEqual(self.circle_resized.circumference, 15 * math.pi)
        self.assertEqual(self.circle_resized.area, 7.5 * 7.5 * math.pi)

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

Code Explanation

Running this file should return the following:

```
PS ...\\classes-objects\\Demos> python test_Circle2.py -v
test_circle_area (__main__.TestCircleFunctions) ... ok
test_circle_circumference (__main__.TestCircleFunctions) ... ok
test_circle_diameter (__main__.TestCircleFunctions) ... ok
test_circle_radius (__main__.TestCircleFunctions) ... ok
test_circle_resized (__main__.TestCircleFunctions) ... ok
```

```
-----
Ran 5 tests in 0.000s
```

```
OK
```

Exercise 47: Adding a roll() Method to Die

15 to 25 minutes.

Currently, we have a `Die` class to create die objects with some number of `sides`, but we have no way to roll the die. In this exercise, you will add a `roll()` method to the `Die` class.

1. Navigate to [classes-objects/Exercises](#) and open `Die.py` in your editor.
2. Add a `roll()` method that returns an integer between 1 and `sides`. You will need to import the `random` module.
3. Test your solution by creating an instance of `Die` and calling the `roll()` method several times.

Challenge

Write code to roll the die 100,000 times and then use a `Counter` object to create a list that shows how many times each side was rolled. It should output something like this:

```
[(1, 16422), (2, 16596), (3, 16567), (4, 16761), (5, 16951), (6,
```

Solution: classes-objects/Solutions/Die1.py

```
import random
```

```
class Die:
```

```
    def __init__(self, sides=6):  
        self.sides = sides
```

```
    def roll(self):
```

```
        roll = random.randint(1, self.sides)  
        return roll
```

Solution: classes-objects/Solutions/roll_die1.py

```
from Die import Die
```

```
die = Die()
```

```
roll = die.roll()
```

```
print(roll)
```

Challenge Solution: `classes-objects/Solutions/roll_die1_challenge.py`

```
from collections import Counter
from Die import Die

die = Die()

rolls = []
for i in range(100000):
    roll = die.roll()
    rolls.append(roll)

c = Counter(rolls)
c_sorted = sorted(c.items())

print(c_sorted)
```

Private Attributes

Many object-oriented programming languages have the concept of *private attributes* – attributes of the instance objects that can only be modified within the class definition. To understand the need for private attributes, consider what would happen if we set a value for the radius of our circle `c` like this:

```
c.radius = 25
```

That would change the value of `radius`, but would not change the values of `diameter`, `circumference`, and `area`, and so, the circle's `radius` would now be out of sync with its other attributes. In languages that allow for private attributes, you would explicitly mark those attributes private and only allow access to them through getter and setter methods, like this:

Demo 141: classes-objects/Demos/Circle3.py

```
import math
class Circle:
    def __init__(self, val, prop='r'):
        if prop == 'r':
            self.set_radius(val)
        elif prop == 'd':
            self.set_diameter(val)
        elif prop == 'c':
            self.set_circumference(val)
        elif prop == 'a':
            self.set_area(val)
        else:
            raise Exception('prop must be r, d, c, or a')

    def set_radius(self, r):
        self._radius = r
        self._diameter = r * 2
        self._circumference = r * 2 * math.pi
        self._area = r ** 2 * math.pi

    def get_radius(self):
        return self._radius

    def set_diameter(self, d):
        self.set_radius(d / 2)

    def get_diameter(self):
        return self._diameter

    def set_circumference(self, c):
        self.set_radius(c / (2 * math.pi))

    def get_circumference(self):
        return self._circumference

    def set_area(self, a):
        self.set_radius((a / math.pi) ** .5)
```

```
def get_area(self):
    return self._area

def resize_by(self, amount):
    r = self._radius * (1 + amount)
    self.set_radius(r)
```

Code Explanation

Notice that the attributes are now preceded by an underscore (e.g., `_radius`). That is a convention to indicate that the attribute is meant to be private (i.e., not accessed directly from outside of the class definition). Instead, each attribute should be retrieved with its *getter* and set with its *setter*. The `set_radius()` setter sets all the “private” attributes and the other setters (e.g., `set_diameter()`) just hand off the work to `set_radius()`.

To access the attributes’ values, we use the getters:

```
>>> from Circle3 import Circle
>>> c = Circle(10, 'd')
>>> a = c.get_area()
>>> a
78.53981633974483
>>> c.set_radius(8)
>>> a = c.get_area()
>>> a
201.06192982974676
```

Note that in Python there is no way to actually prevent developers from accessing the “private” attributes as the following example (continued from the preceding code) illustrates:

```
>>> c._radius = 5
>>> a = c.get_area()
>>> print(a)
```

```
201.06192982974676
```

The new area should have been set back to 78.53981633974483 to correspond to a radius of 5, but notice that setting `c._radius` did not cause the area of `c` to get updated. Neither did it result in an error, so the developer won't know that something went wrong. Python developers just need to know not to mess with attributes that begin with underscores.

Properties

While using `get_` and `set_` methods like we did in our previous example works, it is not the Pythonic way of creating getters and setters. Python developers prefer being able to access attributes directly:

Not Pythonic

```
c.set_area(25)
a = c.get_area()
```

Pythonic

```
c.area = 25
a = c.area
```

This is handled in Python through *properties*. Think of a property as an attribute with a defined getter and possibly a setter and a deleter as well.

There are two ways to create properties: with the `property()` function and with the `@property` *decorator*.^{[63](#)}

Creating Properties with the `property()` Function

```
class Circle:
    def __init__(self):
```

```
        self._radius = None

    def get_radius():
        return self._radius

    def set_radius(self, r):
        self._radius = r

    radius = property(get_radius, set_radius)
```

The method above makes use of `get_` and `set_` methods like many other object-oriented languages, but then it uses the built-in `property()` function to create a `radius` property, which allows `radius` to be got and set directly using `c.radius`.

Creating Properties using the `@property` Decorator

```
class Circle:
    def __init__(self):
        self._radius = None

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, r):
        self._radius = r
```

`@property` is a decorator, which turns the method directly following it into a getter function.

`@radius.setter` is also a decorator, which turns the method directly following it into a setter function for `radius`.

Here is our `circle` class with properties defined:

Demo 142: classes-objects/Demos/Circle4.py

-----Lines Omitted-----

```
def resize_by(self, amount):  
    r = self._radius * (1 + amount)  
    self.set_radius(r)
```

@property

```
def radius(self):  
    return self._radius
```

@radius.setter

```
def radius(self, r):  
    self._radius = r  
    self._diameter = r * 2  
    self._circumference = r * 2 * math.pi  
    self._area = r ** 2 * math.pi
```

@property

```
def diameter(self):  
    return self._diameter
```

@diameter.setter

```
def diameter(self, d):  
    self.radius = d / 2
```

@property

```
def circumference(self):  
    return self._circumference
```

@circumference.setter

```
def circumference(self, c):  
    self.radius = c / (2 * math.pi)
```

@property

```
def area(self):  
    return self._area
```

@area.setter

```
def area(self, a):  
    self.radius = (a / math.pi) ** .5
```

Forgetting the Underscore

A common error when using the `@property` decorator is to forget the underscore when attempting to return the attribute, like this:

```
class Foo:  
    @property  
    def name(self):  
        return self.name  
  
a = Foo()  
print(a.name)
```

This will cause an infinite loop and result in a “maximum recursion depth exceeded while calling a Python object” error, because the `return` statement is trying to return `self.name`, which directs Python back to the `name()` getter, which tries again to return `self.name`. This is bad. So, don’t forget your underscores.

Exercise 48: Properties

15 to 25 minutes.

In this exercise, you will convert `get_` methods to properties.

1. Open `classes-objects/Exercises/Simulation.py` in your editor.
2. Notice the `get_mean()`, `get_median()`, and `get_mode()` methods, which take advantage of the `statistics` module to calculate average results of a die rolled many times.
3. Run the code to see how it works. Here is a sample script you can run (using the `Die` class with the `roll` method assigned in an earlier exercise):

```
>>> from Die import Die
>>> from Simulation import Simulation
>>> die = Die()
>>> sim = Simulation(die.roll, 1000)
>>> sim.get_mean()
3.453
>>> sim.get_median()
3.0
>>> sim.get_mode()
1
```

4. Convert the three `get_` methods to properties and test your solution with the same code as above but replacing the `get_` methods statement with property references:

```
>>> from Die1 import Die
>>> from Simulation import Simulation
>>> die = Die()
>>> sim = Simulation(die.roll, 1000)
>>> sim.mean
3.453
>>> sim.median
3.0
>>> sim.mode
1
```

Solution: classes-objects/Solutions/Simulation.py

```
import statistics as stats

class Simulation:
    def __init__(self, fnct_to_run, iterations):
        self._fnct_to_run = fnct_to_run
        self._iterations = iterations
        self._results = []
        self.run()

    def run(self):
        for i in range(self._iterations):
            result = self._fnct_to_run()
            self._results.append(result)

    @property
    def mean(self):
        return stats.mean(self._results)

    @property
    def median(self):
        return stats.median(self._results)

    @property
    def mode(self):
        try:
            return stats.mode(self._results)
        except:
            return None
```

Objects that Track their Own History

In the `Simulation` class from the last exercise, we keep track of the rolls, but a die could keep track of its own history as well:

Demo 143: classes-objects/Demos/Die3.py

```
import random

class Die:
    def __init__(self, sides=6):
        if type(sides) != int or sides < 1:
            raise Exception('sides must be a positive integer.')
        self._sides = sides
        self._rolls = []

    @property
    def rolls(self):
        return self._rolls

    def roll(self):
        roll = random.randint(1, self._sides)
        self._rolls.append(roll)
        return roll
```

Code Explanation

Each time the `roll()` method is called the `die` instance will automatically append the result of the roll to its `_rolls` property.

Demo 144: classes-objects/Demos/roll_die3.py

```
from collections import Counter
from Die3 import Die

die = Die()

for i in range(100000):
    roll = die.roll()

rolls = die.rolls
c = Counter(rolls)
c_sorted = sorted(c.items())

print(c_sorted)
```

Code Explanation

Notice that we no longer have to keep a list of rolls as we did in the solution to the challenge in the [Adding a roll\(\) Method exercise](#), which contains this code:

```
rolls = []
for i in range(100000):
    roll = die.roll()
    rolls.append(roll)
```

Instead, we use the die's own `rolls` property:

```
rolls = die.rolls
```

Documenting Classes

One nice thing about using classes is the documentation you get for free. Check out the documentation for our `circle` class in [classes-objects/Demos/Circle4.py](#):

```
>>> import Circle4
>>> help(Circle4)
Help on module Circle4:
```

NAME

Circle4

CLASSES

builtins.object

Circle

```
class Circle(builtins.object)
```

```
| Circle(val, prop='r')
```

```
|
```

```
| Methods defined here:
```

```
|
```

```
| __init__(self, val, prop='r')
```

```
| Initialize self. See help(type(self)) for accurate signature
```

```
|
```

```
| resize_by(self, amount)
```

```
|
```

```
| -----
```

```
| Data descriptors defined here:
```

```
|
```

```
| __dict__
```

```
| dictionary for instance variables (if defined)
```

```
|
```

```
| __weakref__
```

```
| list of weak references to the object (if defined)
```

```
|
```

```
| area
```

```
|
```

```
| circumference
```

```
|
```

```
| diameter
```

```
|
```

```
| radius
```

This tells us that the `circle` class has two methods: `__init__()` and `resize_by()` and four properties: `area`, `circumference`, `diameter`, and `radius`.

Using docstrings

Assuming we named our attributes, methods, and properties well, we will get some pretty good free documentation, but we can make it a lot better by using *docstrings*. A docstring is just a string placed at the beginning of a module, function, class, or method definition. The string can be a single line (in single quotes) or multiple lines (in triple quotes). By convention, double quotation marks (" or "") are used for docstrings.

As a rule, all classes and their methods should include docstrings. Methods should include documentation on any keyword arguments.

Following are some docstrings for our `circle` class:

```
class Circle:
    "A circle"
    def __init__(self, val, prop='r'):
        """Create a circle based on a radius, diameter,
            circumference, or area

            Keyword arguments:
            val (float) -- the value of prop
            prop (str)
                -- 'r' : radius (default)
                -- 'd' : diameter
                -- 'c' : circumference
                -- 'a' : area
            """
        self._radius = None
        self._diameter = None
        self._circumference = None
        self._area = None
        if prop == 'r':
            self.radius = val
        elif prop == 'd':
            self.diameter = val
        elif prop == 'c':
            self.circumference = val
        elif prop == 'a':
            self.area = val
        else:
            raise Exception('prop must be r, d, c, or a')

    @property
    def radius(self):
        "radius of the circle object"
        return self._radius

    @radius.setter
    def radius(self, r):
        """sets _radius, _diameter, _circumference, and _area of
```

```

        circle object"""
    self._radius = r
    self._diameter = r * 2
    self._circumference = r * 2 * math.pi
    self._area = r ** 2 * math.pi

@property
def diameter(self):
    "diameter (2 x r) of the circle object"
    return self._diameter

@diameter.setter
def diameter(self, d):
    """uses diameter d to set radius, which then
    updates all related pseudo-private attributes"""
    self.radius = d / 2

@property
def circumference(self):
    "circumference (PI x d) of the circle object"
    return self._circumference

@circumference.setter
def circumference(self, c):
    """uses circumference c to set radius, which then update:
    all related pseudo-private attributes"""
    self.radius = c / (2 * math.pi)

@property
def area(self):
    "area (PI x r squared) of the circle object"
    return self._area

@area.setter
def area(self, a):
    """uses area a to set radius, which then updates all
    related pseudo-private attributes"""
    self.radius = (a / math.pi) ** .5

```

```

def resize_by(self, amount):
    """resizes radius, which then updates all related
       pseudo-private attributes

    Keyword arguments:
    amount (float) -- the amount by which to resize the radiu
                   -- a negative number shrinks the radius
    """
    self.radius = self.radius * (1 + amount)

```

Check out how this improves the help documentation:

```

>>> import Circle5
>>> help(Circle5)
Help on module Circle5:

NAME
    Circle5

CLASSES
    builtins.object
        Circle

class Circle(builtins.object)
|   Circle(val, prop='r')
|
|   A circle
|
|   Methods defined here:
|
|   __init__(self, val, prop='r')
|       Create a circle based on a radius, diameter,
|       circumference, or area
|
|       Keyword arguments:
|       val (float) -- the value of prop
|       prop (str)
|           -- 'r' : radius (default)
|           -- 'd' : diameter

```

```

|         -- 'c' : circumference
|         -- 'a' : area
|
|     resize_by(self, amount)
|         resizes radius, which then updates all related
|             pseudo-private attributes
|
|     Keyword arguments:
|     amount (float) -- the amount by which to resize the
|                     -- a negative number shrinks the radi
|

```

Notice the help does not include the setter documentation. If you want that documentation to show up in the help, you could include it in the getter, like this:

```

@property
def radius(self):
    """radius of the circle object
    setter: sets _radius, _diameter, _circumference,
    and _area of Circle object
    """

@property
def diameter(self):
    """diameter (2 x r) of the circle object
    setter: uses diameter d to set radius, which then updates
    all pseudo-private attributes
    """

```

Exercise 49: Documenting the Die Class

10 to 20 minutes.

In this exercise, you will add docstrings to the `Die` class.

1. Open [classes-objects/Exercises/Die.py](#) in your editor.

2. Add docstrings to the class so that `help(Die)` will output the following:

```
Help on class Die in module Die:
```

```
class Die(builtins.object)
|   A die
|
|   Methods defined here:
|
|   __init__(self, sides=6)
|       Creates a new standard die
|
|       Keyword arguments:
|       sides (int) -- number of die sides.
|
|   roll(self)
|       Returns a value between 1 and the number of die sides.
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
|
|   rolls
|       history of rolls
```

Solution: classes-objects/Solutions/Die2.py

```
import random

class Die:
    "A die"
    def __init__(self, sides=6):
        """Creates a new standard die

        Keyword arguments:
            sides (int) -- number of die sides."""
        if type(sides) != int or sides < 1:
            raise Exception('sides must be a positive integer.')
        self._sides = sides
        self._rolls = []

    @property
    def rolls(self):
        "history of rolls"
        return self._rolls

    def roll(self):
        "Returns a value between 1 and the number of die sides."
        roll = random.randint(1, self._sides)
        self._rolls.append(roll)
        return roll
```

Inheritance

Often you will find a class has a lot of the functionality you need, but is missing something. No worries. You can create your own class that inherits all the functionality of the other class and then you can make additions and modifications. The syntax for doing so is:

```
class A:
    pass

class B(A):
    pass
```

B is a *subclass* of A. A is a *superclass* of B.

Overriding a Class Method

Here is an example in which B overrides a method of A:

Demo 146: classes-objects/Demos/inheritance.py

```
class A:
    def __init__(self, name):
        self.name = name

    def intro(self):
        print('Hello, my name is {}'.format(self.name))

    def outro(self):
        print('Goodbye!')

class B(A):
    def intro(self):
        print('Hi, I am {}'.format(self.name))

a = A('George')
b = B('Ringo')

a.intro()
b.intro()
a.outro()
b.outro()
```

The output will be:

```
PS ...\\classes-objects\\Demos> python inheritance.py
Hello, my name is George.
Hi, I am Ringo.
Goodbye!
Goodbye!
```

Things to notice:

1. When we call `b.intro()`, it uses the `intro()` method defined in the `B` class.
2. When we call `b.outro()`, it uses the `outro()` method defined in the `A` class, because that method is not overwritten in the `B` class.

Extending a Class

The built-in `list` class has an `append()` method for appending new items to a `list`. It also has an `insert()` method for inserting new items at a specific index. However, it has no `prepend()` method. If you want to prepend an item to a list, the syntax is `mylist.insert(0, item)`. Let's create our own subclass of `list` that includes a `prepend()` method:

Demo 147: classes-objects/Demos/MyList.py

```
class MyList(list):
    "A subclass of list with additional functionality"
    def prepend(self, obj):
        """prepend obj to list

        Keyword arguments:
        obj -- obj to prepend"""
        self.insert(0, obj)
```

Now, let's test our new class:

```
>>> from MyList import MyList
>>> mylist = MyList(['a', 'b', 'c'])
>>> mylist.append('y')
>>> mylist.prepend('z')
>>> mylist
['z', 'a', 'b', 'c', 'y']
```

Notice that `mylist` has the `append()` method, which `MyList` inherited from `list` and it also has the new `prepend()` method.

Exercise 50: Extending the Die Class

15 to 25 minutes.

In this exercise, you will create a `WeightedDie` class that extends the `Die` class.

Exercise Code: classes-objects/Exercises/WeightedDie.py

```
import random
from collections import Counter
from Die2 import Die

class WeightedDie(Die):
    "A weighted die"
    def __init__(self, weights, sides=6):
        """Creates a new weighted die

        Keyword arguments:
        sides (int) -- number of die sides.
        weights (list) -- a list of integers holding the weights
            for each die side
        """
        if len(weights) != sides:
            raise Exception(f'weights must be a list of length {sides}')
        super().__init__(sides)
        self._weights = weights

    def roll(self):
        """Returns a value between 1 and the number of die sides

        # COMPLETE THIS CODE
```

1. Open classes-objects/Exercises/WeightedDie.py in your editor.
2. Review the `__init__()` method of the `WeightedDie` class. Notice that it creates a pseudo-private `_weights` attribute that holds a list of weights, each corresponding to a side of the die. A six-sided die with the following `_weights` should roll a 6 five of every ten rolls (on average):

```
[1, 1, 1, 1, 1, 5]
```

3. Complete the `roll()` method in the `WeightedDie` class. Again, the odds of returning a value should correlate to the weight in

`self._weights`. One way of doing this is to create a new list that contains each possible roll `n` times, where `n` is the associated weight. For example, for the `self._weights` above, the new list would look like this:

```
[1, 2, 3, 4, 5, 6, 6, 6, 6, 6]
```

Then use `random.choice()` to select a value from that list.

4. To test your solution, run the following code:

```
from WeightedDie import WeightedDie
from collections import Counter
die = WeightedDie(weights=[1, 1, 1, 1, 1, 5])

for i in range(100000):
    roll = die.roll()

c = Counter(die.rolls)
c_sorted = sorted(c.items())
c_sorted
```

The output should be something like this:

```
[(1, 9899), (2, 10012), (3, 10083), (4, 10133), (5, 10011), (
```

5. Notice that the instance of `WeightedDie` has a `rolls` property, which it inherited from the `Die` class.

Solution: classes-objects/Solutions/WeightedDie.py

```
import random

from Die import Die

class WeightedDie(Die):
    "A weighted die"
    def __init__(self, weights, sides=6):
        """Creates a new weighted die

        Keyword arguments:
        sides (int) -- number of die sides.
        weights (list) -- a list of integers holding the weights
            for each die side
        """
        if len(weights) != sides:
            raise Exception(f'weights must be a list of length {sides}')
        super().__init__(sides)
        self._weights = weights

    def roll(self):
        """Returns a value between 1 and the number of die sides
        options = []
        for i in range(self._sides):
            for j in range(self._weights[i]):
                options.append(i+1)
        roll = random.choice(options)
        self._rolls.append(roll)
        return roll
```

Extending a Class Method

Suppose a class you are extending has a method that does almost everything you want it to do, but you'd like to add something more. You may find you're able to extend the method rather than overwrite it. Here's a simple example:

Demo 148: classes-objects/Demos/extending_a_class_method.py

```
class A:
    def __init__(self, name):
        self.name = name

    def intro(self):
        print('Hello, my name is {}'.format(self.name))

    def outro(self):
        print('Goodbye!')

class B(A):
    def intro(self):
        super().intro()
        print('It\'s very nice to meet you.')

a = A('George')
b = B('Ringo')

a.intro()
print('-----')
b.intro()
print('-----')
a.outro()
b.outro()
```

This code produces the following output:

```
PS ...\\classes-objects\\Demos> python extending_a_class_method.py
Hello, my name is George.
-----
Hello, my name is Ringo.
It's very nice to meet you.
-----
Goodbye!
Goodbye!
```

Notice the `intro()` method of class B. It first calls `super().intro()`,

which calls the `intro()` method of the superclass. Then it extends the method by printing “It's very nice to meet you.”

Creating a Non-Negative Counter

You may remember from a [previous lesson](#) that when subtracting from a Counter, it is possible to end up with negative counts:

Subtracting with a Counter

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green'])
>>> c # Before subtraction:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})
>>> c.subtract(['red', 'yellow', 'yellow', 'purple'])
>>> c # After subtraction:
Counter({'blue': 3, 'green': 2, 'red': 0, 'yellow': -1, 'purple': 1})
```

Often, as with a product inventory, having a negative count doesn't make sense.

Counter objects have a special `__setitem__()` method that sets key values. We can override that method like this:

Demo 149: classes-objects/Demos/NonNegativeCounter.py

```
from collections import Counter

class NonNegativeCounter(Counter):
    'Counter that disallows negative values'
    def __setitem__(self, key, value):
        value = 0 if value < 0 else value
        super().__setitem__(key, value)
```

Code Explanation

On line 6, we set `value` to `0` if it is less than `0`. Otherwise, we set it to the value passed in to `__setitem__()`.

Then, on the next line, we pass `key` and `value` to `super().__setitem__()`.

`super()` refers to this class's *superclass*; that is, to `Counter`.

Let's see how it works:

```
>>> from NonNegativeCounter import NonNegativeCounter
>>> c = NonNegativeCounter(['green', 'blue', 'blue', 'red', 'yel
>>> c.subtract(['red', 'yellow', 'yellow', 'purple'])
>>> c
NonNegativeCounter({'blue': 3, 'green': 2, 'red': 0, 'yellow': 0
```

Notice that this time the values for the `yellow` and `purple` keys are both `0`.

Exercise 51: Extending the `roll()` Method

10 to 20 minutes.

In this exercise, you will create a `WeightedDie` class that extends the `WeightedDie` class. A `WeightedDie` starts with equal weights on each

side, but it becomes weighted by giving more weight to rolls it has rolled before. It does this by modifying the `_weights` attribute with each roll. Here is the initial code:

Exercise Code: classes-objects/Exercises/WeightingDie.py

```
import random
from collections import Counter
from WeightedDie import WeightedDie

class WeightingDie(WeightedDie):
    "A weighting die"
    def __init__(self, sides=6):
        """Creates a die that favors sides it has previously rolled.

        Keyword arguments:
        sides (int) -- number of die sides.
        """
        self._weights = [1] * sides
        super().__init__(self._weights, sides)

    def roll(self):
        """Returns a value between 1 and the number of die sides
        # COMPLETE THE CODE
```

Code Explanation

Review the `__init__()` method of the `weightingDie` class. Notice that it does not take a `weights` parameter like its superclass does. Rather, it sets all values in `_weights` to 1 using:

```
self._weights = [1] * sides
```

It then calls `super().__init__()` passing in `self._weights` and `sides`.

1. Open [classes-objects/Exercises/WeightingDie.py](#) in your editor.
2. Complete the `roll()` method so that it:
 - A. Calls the `roll()` method of the superclass and stores the result in a local variable.

- B. Modifies `self._weights` so that the weight for the roll just rolled is incremented by 1.
- C. Returns the roll.

3. To test your solution, run the following code:

```
from collections import Counter
from WeightingDie import WeightingDie
die = WeightingDie()

for i in range(10000):
    roll = die.roll()

c = Counter(die.rolls)
c_sorted = sorted(c.items())
c_sorted
```

The output should be something like this:

```
[(1, 473), (2, 70), (3, 828), (4, 4418), (5, 3359), (6, 852)]
```

Solution: classes-objects/Solutions/WeightingDie.py

```
import random
from WeightedDie import WeightedDie

class WeightingDie(WeightedDie):
    "A weighted die"
    def __init__(self, sides=6):
        """Creates a die that favors sides it has previously rolled.

        Keyword arguments:
        sides (int) -- number of die sides.
        """
        self._weights = [1] * sides
        super().__init__(self._weights, sides)

    def roll(self):
        """Returns a value between 1 and the number of die sides
        result = super().roll()
        self._weights[result-1] += 1
        return result
```

Static Methods

As we've discussed, when you call a regular method on an instance of a class, the instance itself is passed in as the first argument. Python classes can also have *static methods*, which are created by including `@staticmethod` immediately before the method. A static method can be called directly on the class or on an instance of the class. It does not take `self` as the first parameter. For instance:

```
class Planet:
    @staticmethod
    def greet():
        print('hello')

Planet.greet()
earth = Planet()
```



```
earth.greet()
```

And here is a more practical example:

Demo 150: classes-objects/Demos/Triangle.py

```
class Triangle:
    def __init__(self, sides):
        if not self.is_triangle(sides):
            raise Exception('Cannot make triangle with those sides')
        self._sides = sides

    @property
    def perimeter(self):
        return sum(self._sides)

    @property
    def area(self):
        p = self.perimeter/2
        a = self._sides[0]
        b = self._sides[1]
        c = self._sides[2]
        return ( p * (p-a) * (p-b) * (p-c) ) ** .5

    @staticmethod
    def is_triangle(sides):
        if len(sides) != 3:
            return False
        sides.sort()
        if sides[0] + sides[1] < sides[2]:
            return False
        return True
```

Things to notice:

1. The `is_triangle()` method is preceded by `@staticmethod`, indicating that it is a static method.
2. The `is_triangle()` does **not** take `self` as its first parameter.
3. On line 3, the `__init__()` method makes use of the `is_triangle()` method to check whether it is possible to make a triangle from the sides. It can call the method on `self` (i.e., the instance) but doesn't

pass self in. It just passes in sides.

Let's try out the Triangle class:

```
>>> from Triangle import Triangle
>>> good = [3,3,5]
>>> bad = [3,3,9]
>>> print( Triangle.is_triangle(good) )
True
>>> print( Triangle.is_triangle(bad) )
False
>>> t1 = Triangle(good)
>>> print(t1.area)
4.14578098794425
>>> t2 = Triangle(bad)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Webucator\Python\classes-objects\Demos\Triangle.py", .
    raise Exception('Cannot make triangle with those sides.')
Exception: Cannot make triangle with those sides.
```

Things to note:

1. The static `is_triangle()` method of the `Triangle` class returns `True` for a good triangle and `False` for a bad triangle.
2. We call the static method using the class name, `Triangle`. We do not need to instantiate a triangle object to call the method.
3. We can instantiate and print the area of a good triangle but if we attempt to instantiate a bad triangle, an exception is raised in the `__init__` method.

Class Attributes and Methods

Class Attributes

A class attribute is an attribute that is defined outside of a class method, like `foo` is in the following code:

```
class A:
    foo = 1
    def __init__(self):
        pass
```

Class attributes can be accessed by classes themselves and by their instance objects; however, you have to be careful, as instance attributes will take precedence over class attributes. The following code (from [classes-objects/Demos/class_attributes_and_methods.py](#)) illustrates this:

```
class A:
    foo = 1
    bar = 1
    def __init__(self):
        self.foo = 2

a = A()
print(a.foo, A.foo, a.bar, A.bar)
```

Output:
2 1 1 1

Notice that `a.foo` and `A.foo` return different values, but `a.bar` and `A.bar` both return the value of the class attribute.

However, if you use an instance variable to modify a mutable object stored as a class attribute, a new object will not be created. To see this, examine the following code:

```
class A:
    foo = ['a', 'b', 'c']
    def __init__(self):
        self.foo.append('d')

a = A()
a.foo, A.foo)
```

Output:

```
['a', 'b', 'c', 'd'] ['a', 'b', 'c', 'd']
```

In this case, `a.foo` and `A.foo` return the same values. In other words, they are both pointing to the same object.

Class Methods

Class methods are different from standard methods, which receive the instance object as their first parameter, and static methods, which do not receive any default arguments. Class methods receive the class itself as their first argument. To indicate that a method is a class method, precede it with `@classmethod`. Just like with a static method, a class method can be called directly on the class or on an instance of the class like this:

```
class A:
    foo = 1
    @classmethod
    def my_class_method(cls):
        cls.foo += 1
        return cls.foo

x = A.my_class_method()

a = A()
y = a.my_class_method()

print(x, y)
```

`x` will contain 2 and `y` will contain 3.

self and cls Parameters

The parameter names `self` and `cls` used in standard and class methods are used by convention. You can name them whatever you want, but it's best to stick with the convention.

So, when do you use class attributes and methods. Imagine you have a `Plane` class. Instances of `Plane` can take off and land. You want to keep track of all the `Plane` instances that have been created and also know how many are currently in the air. Here's a class for doing that:

Demo 151: classes-objects/Demos/Plane.py

```
class Plane:
    planes = []
    def __init__(self):
        self._in_air = False
        type(self).planes.append(self)

    def take_off(self):
        self._in_air = True

    def land(self):
        self._in_air = False

    @classmethod
    def num_planes(cls):
        return len(cls.planes)

    @classmethod
    def num_planes_in_air(cls):
        return len([plane for plane in cls.planes if plane._in_a:
```

Imagine that we test the `Plane` class with the following statements:

```
from Plane import Plane
p1 = Plane()
p2 = Plane()
p3 = Plane()
p1.take_off()
p2.take_off()
p1.land()
print(Plane.num_planes(), Plane.num_planes_in_air())
```

Here is the output:

```
3 1
```

Things to note:

1. When we append `self` to `self.planes` (line 5), the plane instance actually gets appended to the class attribute `planes`. That's because there is no instance attribute `planes`, so the instance looks to the class for the attribute.
2. `num_planes()` and `num_planes_in_air()` are both class methods, which is why we can call them on `Plane` in line 27.
3. After two planes take off and one lands, there is only one plane left in the air.

You Must Consider Subclasses

There is a hard-to-spot problem in our code though. It only reveals itself when we subclass `Plane`, like this :

Demo 152: classes-objects/Demos/Jet.py

```
from Plane import Plane

class Jet(Plane):
    def __init__(self):
        self.planes = []
        super().__init__()
```

If we run the following code,

```
from Plane import Plane
from Jet import Jet
p1 = Plane()
p2 = Plane()
p3 = Plane()
p1.take_off()
p2.take_off()
p1.land()
p4 = Jet()
p4.take_off()
print(Plane.num_planes(), Plane.num_planes_in_air())
```

the result is

```
3 1
```

Notice that the `__init__()` method of `Jet` defines a `self.planes` attribute. When we append `self` to `self.planes` in the `__init__()` method of the superclass, this plane instance gets appended to the instance attribute `planes`. But our class methods are looking at the class attribute `planes`, so our `Jet` objects won't be included, which is why the results still show a total of three planes with one in the air.

The solution is to change the `__init__()` method of the superclass. One possibility is to use `Plane` instead of `self` when appending the object to `planes`:

```
def __init__(self):
    self._in_air = False
    Plane.planes.append(self)
```

But it's better not to use the class name within the class definition. Instead, you can do this:

```
def __init__(self):
    self._in_air = False
    type(self).planes.append(self)
```

`type(self)` returns the class of the instance object, which is exactly what we want. Now, when `Jet` objects are initialized, they will be appended to the class attribute `planes` rather than the instance attribute `planes`.

If we test again then the output will include the `Jet` that took off:

```
4 2
```

Abstract Classes and Methods

An abstract class is a class that cannot be instantiated but is created for the purposes of subclassing. For example, imagine we're creating a game with a bunch of flying objects, including planes and birds. A few things to note:

1. Planes can only take off when the pilot is awake.
2. Planes can only land when they are over land.
3. Birds can only take off when their wings are healthy.
4. Birds can land anywhere.

Both birds and planes can take off and land, but they may do so in different ways and they may differ in other ways as well. So, we'll create a `FlyingObject` class for objects that can fly, but we don't want people to be able to instantiate `FlyingObject` objects. Rather, we want people to subclass `FlyingObject` to create more specific object types.

Here's our first stab at our `FlyingObject` class:

```
class FlyingObject:
    flyingobjects = []
    def __init__(self):
        self._in_air = False
        type(self).flyingobjects.append(self)

    def take_off(self):
        self._in_air = True

    def land(self):
        self._in_air = False

    @classmethod
    def num_objects(cls):
        return len(cls.flyingobjects)

    @classmethod
    def num_objects_in_air(cls):
        return len([fo for fo in cls.flyingobjects if fo._in_air])
```

Let's test our new class:

```
import FlyingObject from FlyingObject
ufo = FlyingObject()
print(FlyingObject.num_objects())
```

Output:

1

For the most part, this class serves our purposes, but it has one flaw; it can be instantiated, as the output from the script reveals above.

To prevent that, we need to explicitly specify that `FlyingObject` is an abstract class. The way to do that is to make any one of its methods abstract. Doing so, will:

1. Prevent people from instantiating `FlyingObject` objects.

2. Force subclasses to implement the methods marked abstract.

In Python, you create an abstract class, by

1. Importing the `abc` module.

2. Creating the class using `metaclass=abc.ABCMeta`.

3. Using `@abc.abstractmethod` decorators.

Here again is our `FlyingObject` class, now legitimately abstract:

Demo 153: classes-objects/Demos/FlyingObject.py

```
import abc

class FlyingObject(metaclass=abc.ABCMeta):
    flyingobjects = []
    def __init__(self):
        self._in_air = False
        type(self).flyingobjects.append(self)

    @abc.abstractmethod
    def take_off(self):
        self._in_air = True

    @abc.abstractmethod
    def land(self):
        self._in_air = False

    @classmethod
    def num_objects(cls):
        return len(cls.flyingobjects)

    @classmethod
    def num_objects_in_air(cls):
        return len([fo for fo in cls.flyingobjects if fo._in_air])
```

Now, we get an error when we attempt to create an instance of FlyingObject:

```
Traceback (most recent call last):
  File "FlyingObject_abstract.py", line 26, in <module>
    ufo = FlyingObject()
TypeError: Can't instantiate abstract class FlyingObject with ab
```

We also get an error if we attempt to create (and then instantiate) a subclass of FlyingObject without implementing all the abstract methods:

```
class Plane(FlyingObject):

    @property
    def pilot_awake(self):
        return True
    def take_off(self):
        if self.pilot_awake:
            super.take_off()
        self._in_air = True
```

We can try to instantiate a Plane object:

```
from Plane import Plane
plane = Plane()
```

But the attempt to create a Plane object fails and causes the following error message:

```
TypeError: Can't instantiate abstract class Plane with abstract m
```

But when we do implement the abstract methods, we can instantiate with no problem:

```
class Plane(FlyingObject):

    @property
    def pilot_awake(self):
        return True
    @property
    def over_land(self):
        return True
    def take_off(self):
        if self.pilot_awake:
            super.take_off()
        self._in_air = True
    def land(self):
        if self.over_land:
            super.land()
```

Now, we can successfully instantiate a `Plane` object:

```
from FlyingObject import FlyingObject
from Plane import Plane
plane = Plane()
print(FlyingObject.num_objects())
```

The output will be:

```
1
```

And, just to be complete, here's our `Bird` class:

Demo 154: classes-objects/Demos/Bird.py

```
from FlyingObject import FlyingObject

class Bird(FlyingObject):

    @property
    def healthy_wings(self):
        return True
    def take_off(self):
        if self.healthy_wings:
            super().take_off()
        self._in_air = True
    def land(self):
        super().land()
```

We can test the Bird class as follows:

```
from FlyingObject import FlyingObject
from Bird import Bird
bird = Bird()
bird.take_off()
print(FlyingObject.num_objects(), FlyingObject.num_objects_in_ai
```

And here is the output:

```
1 1
```

Understanding Decorators

Class Files Examples

Examples from this section are in <classes-objects/Demos/Decorators.py>.

We have mentioned several *decorators* in this lesson but have not explained what a decorator is. Decorators are functions that add functionality to (i.e., “decorate”) other functions. They take a function as an argument and return a different function. Remember that

functions are objects and objects can be passed from function to function. The following code illustrates this:

```
def foo(f):
    print(f)
    def foo_inner():
        pass
    return foo_inner

def bar():
    pass

print(bar)
bar = foo(bar)
print(bar)
```

Here is the output:

```
<function bar at 0x00000206D86A8EA0>
<function bar at 0x00000206D86A8EA0>
<function foo.<locals>.foo_inner at 0x00000206D86A8E18>
```

1. On line 10, we print `bar` and see that it is a function object. The long number, `0x00...`, is the object's unique memory address.
2. On line 11, we call `foo()` and pass it the `bar` function object. The `foo()` function prints out the object (line 2). And we can see that it prints exactly the same thing, meaning that the local variable `f` is pointing to the `bar()` function defined on lines 7 and 8.
3. On lines 3 and 4, we define `foo_inner()`, which is a function local to the `foo()` function, meaning it can only be called from within the `foo()` function, unless `foo()` returns it, which it does.
4. Back on line 11, we overwrite the `bar` variable with whatever `foo()` returns, which is the `foo_inner()` function.
5. On line 12, we print `bar` and see that it now contains a different function, the local `foo_inner()` function. However, because `bar` is global, this function can now be called from anywhere.

The main takeaway from this is that functions can be passed around just like any other object.

Now, let's take a look at an example of how we can decorate a function with a decorator:

```
from datetime import datetime

def format_report(f):
    def inner(text):
        print('MY REPORT')
        print('-' * 50)
        f(text)
        print('-' * 50)
        print('Report completed: {}'.format(datetime.now()))
    return inner

def report(text):
    print(text)

report = format_report(report)

report('I have created my first decorator.')
```

Here is the output:

```
MY REPORT
-----
I have created my first decorator.
-----
Report completed: 2018-10-09 18:13:57.690144.
```

1. Look at the `report()` function on lines 12 and 13. All it does is print the text that is passed in.
2. On line 15, the `report` function is passed to `format_report()`, which defines and later returns an `inner()` function. This `inner()` function:

- A. Prints a couple of lines of text.
 - B. Runs the passed-in function.
 - C. Prints another couple of lines of text.
3. Back on line 15, we overwrite the `report` variable with the function object returned by `format_report()`.
 4. Now, on line 17, when we call `report()`, it runs the `inner()` function returned by `format_report()`.

Can you see how `format_report()` is decorating (i.e., adding functionality to) the `report()` function? Of course, `format_report()` doesn't do anything terribly exciting, but a decorator can do anything you want it to do. For example, it could keep an event log or send an email.

There is a special decorator syntax, which you have already seen when creating properties and static, class and abstract methods. Instead of explicitly overwriting a function variable with the function returned by a decorator (e.g., `report = format_report(report)`), you indicate that the function will be decorated like this:

```
@foo
def bar():
    pass
```

This is the same as:

```
def bar():
    pass

bar = foo(bar)
```

Using this syntax, we can decorate the `report()` function like this:

```
from datetime import datetime

def format_report(f):
```

```
def inner(text):
    print('MY REPORT')
    print('-' * 50)
    f(text)
    print('-' * 50)
    print('Report completed: {}'.format(datetime.now()))
    return inner

@format_report
def report(text):
    print(text)

report('I have created my second decorator.')
```

And here is the output:

```
MY REPORT
-----
I have created my second decorator.
-----
Report completed: 2018-10-09 19:35:42.681604.
```

As you can see, it works in the same way.

This should give you a better understanding of how the `@property`, `@staticmethod`, `@classmethod`, and `@abc.abstractmethod` are working behind the scenes.

Conclusion

In this lesson, you have learned how to create Python classes and write object-oriented code.

LESSON 18

What Now?

Topics Covered

- Where do you go from here?

In three words I can sum up everything I've learned about life: it goes on.

– Robert Frost

Where to Go from Here

You: It has been fun getting to know you, Python! Challenging, but fun! Now, I'm ready to start working with you. What should I do first?

Python: Well, if you have some real projects you can work on, that would be the best thing. Just start looking for places where I can help you out. For example, you could build a script to search a folder using regular expressions.

You: Can't you do that with Visual Studio Code?

Visual Studio Code does a great job of that, but one thing it doesn't do is make it easy for you to search both file names and content at the same time. You could write a script that does that.

You: That's a good idea. Any other ideas?

Python: Scraping the web is always fun. You could build a tool that gets data from an HTML table and uses it to create a list of lists.

You: A list of lists? That would be like a matrix, right? OK, I'll give it a shot. What else you got for me?

Python: Do you like games?

You: Sure, I like games.

Python: Well then, how about learning to work with Pygame⁶⁴? You use it to create video games in Python (that's me!).

You: Very cool! What else?

Python: The possibilities are endless.

You: Okay, it sounds like I have some great ways to continue working with you!

Python: Yup. Have fun with it (with *me*)! If you enjoyed the book, please consider writing a review. In any case, good luck and happy coding!

Notes

[← 1].

<https://www.webucator.com/class-files/index.cfm?versionId=4734>

[← 2].

<https://pypl.github.io/PYPL.html>

[← 3].

The generic term for the various terminals is *command line shell* ([https://en.wikipedia.org/wiki/Shell_\(computing\)](https://en.wikipedia.org/wiki/Shell_(computing))). Visual Studio Code will select an appropriate terminal for you. On Windows, that will most likely be **Command Prompt** or **PowerShell**. On a Mac, it will most likely be **bash** or **Zsh**. The names “prompt,” “command prompt,” “shell,” and “terminal” are used interchangeably.

[← 4].

The `ls` command works in Windows PowerShell as well.

[← 5].

<https://twitter.com/gvanrossum/status/112670605505077248?lang=en>

[← 6].

Unicode is a 16-bit character set that can handle text from most of the world’s languages.

[← 7].

<http://www.pythontutor.com/visualize.html>

[← 8].

Although only a convention, the name “`main()`” was not chosen arbitrarily. It is used because modules refer to themselves as “`__main__`”, so it seems fitting to get them started with a

corresponding “main()” function.

[← 9].

Absolute paths start from the top of the file system and work their way downwards towards the referenced file. Relative paths start from the current location (the location of the file containing the path) and work their way to the referenced file from that location.

[← 10].

The lists of names come from

<https://www.ssa.gov/oact/babynames/>.

[← 11].

<https://www.python.org/dev/peps/pep-0008/#global-variable-names>

[← 12].

<https://docs.python.org/3/tutorial/modules.html#standard-modules>

[← 13].

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

[← 14].

The `min()` and `max()` functions can also compare strings.

[← 15].

To get a full list of the `math` module’s methods, import `math` and then type `help(math)` in the Python shell or visit

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

[← 16].

https://en.wikipedia.org/wiki/Natural_logarithm

[← 17].

To get a full list of the `random` module’s methods, import `random` and then type `help(random)` in the Python shell or visit

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

[← 18].

<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

[← 19].

Note that there is no “character” type in Python. A single character is just a string of length 1. So, when you index a string, you get multiple one-character strings (or strings of length 1).

[← 20].

Three single quotes will work as well, but double quotes are recommended. More information at

<https://www.python.org/dev/peps/pep-0008/#string-quotes>.

[← 21].

<https://docs.python.org/3/library/string.html#formatspec>

[← 22].

This is actually a slightly simplified version of the format specification. For the full format specification, see

<https://docs.python.org/3/library/string.html#formatspec>.

[← 23].

If you are a mathematician or scientist, you can see all the different available types at

<https://docs.python.org/3/library/string.html#formatspec>.

[← 24].

<https://www.python.org/dev/peps/pep-0008/#maximum-line-length>

[← 25].

f-strings were introduced in Python 3.6.

[← 26].

As we will see later, the `len()` function can also take objects other than strings.

[← 27].

The `min()` and `max()` functions can also compare numbers.

[← 28].

In versions prior to 3.7, an arbitrary item was removed.

[← 29].

You can name this directory whatever want, but it is commonly called “.venv” (notice the prepended dot) to indicate that it is a special directory for holding a virtual environment.

[← 30].

<https://pypi.org/project/playsound/>

[← 31].

The

https://commons.wikimedia.org/wiki/File:World_Time_Zones_Map.pr
image is used under the terms of Public Domain
(https://en.wikipedia.org/wiki/Copyright_status_of_works_by_the_fed

[← 32].

The time of the epoch varies across computing systems, but in Python it is pretty much guaranteed to by January 1, 1970. For a discussion on this, see <https://grokbase.com/t/python/python-dev/086gxjdb5a/epoch-and-platform>.

[← 33].

Documentation on the `time` module:

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

[← 34].

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

[← 35].

Documentation on `date` and `time` formatting directives is available at <https://docs.python.org/3/library/datetime.html?highlight=strptime#strptime-and-strptime-format-codes>.

[← 36].

Documentation on the `datetime` module:

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

[← 37].

<https://www.theatlantic.com/family/archive/2018/09/girls-names-for-baby-boys/569962/>

[← 38].

<https://www.python.org/dev/peps/pep-0008/>

[← 39].

In our class files, we use only a single blank line to separate functions to keep files shorter for printing purposes.

[← 40].

<https://mail.python.org/pipermail/python-dev/2006-February/060415.html>

[← 41].

`sys.path` contains a list of strings specifying the search path for modules. The list is os-dependent. To see your list, import `sys`, and then output `sys.path`.

[← 42].

Groups can also be named through a Python extension to regular expressions. For more information, see <https://docs.python.org/3/howto/regex.html#non-capturing-and-named-groups>.

[← 43].

<https://docs.python.org/3/howto/regex.html>

[← 44].

<https://www.python.org/dev/peps/pep-0249/>

[← 45].

<http://www.seanlahman.com/baseball-archive/statistics/>

[← 46].

<https://pypi.org/project/mysql-connector-python/>

[← 47].

PEP stands for Python Enhancement Proposal.

[← 48].

Check out the `pprint` library at

<https://docs.python.org/3/library/pprint.html> to see how you can pretty print data results. We have an example at [working-with-data/Demos/pretty_print.py](#).

[← 49].

https://cx-oracle.readthedocs.io/en/latest/user_guide/bind.html

[← 50].

See <https://docs.python.org/3/library/sqlite3.html> for documentation on Python's `sqlite3` library and <https://www.sqlite.org> for documentation on SQLite itself.

[← 51].

The state populations are based on data from

https://en.wikipedia.org/wiki/List_of_U.S._states_and_territories_by_2020_numbers are extrapolated.

[← 52].

The population data is from

<https://www.census.gov/data/tables/time-series/demo/popest/2010s-state-total.html>. The 2020 numbers are extrapolated.

[← 53].

<https://requests.readthedocs.io/en/master/>

[← 54].

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

[← 55].

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/#specifyin-the-parser-to-use>

[← 56].

<https://json.org>

[← 57].

<https://api.openweathermap.org/data/2.5/forecast/daily>

[← 58].

For a list of many JSON APIs, see

https://www.programmableweb.com/category/all/apis?data_format=21173.

[← 59].

In Python 3.6 and earlier, the default value for `repeat` was 3.

[← 60].

Historically, there were some differences, but those differences are largely academic.

[← 61].

The `tuple`, `list`, `dict`, and `range` classes are exceptions because they are usually used like functions rather than classes.

[← 62].

You could name the first parameter by a different name, but you really shouldn't. It would just be self-punishment.

[← 63].

Decorators are functions that add functionality to (i.e., “decorate”) other functions.

[← 64].

<https://www.pygame.org/docs/>