Introduction to Programming with Python®

Introduction to Programming with Python®

Part Number: 094021
Course Edition: 1.01

Acknowledgements

PROJECT TEAM

Authors	*Media Designer*	*Content Editor*
Jason Nufryk Brian S. Wilson	Brian Sullivan	Peter Bauer

Notices

DISCLAIMER

While Logical Operations, Inc. takes care to ensure the accuracy and quality of these materials, we cannot guarantee their accuracy, and all materials are provided without any warranty whatsoever, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. The name used in the data files for this course is that of a fictitious company. Any resemblance to current or future companies is purely coincidental. We do not believe we have used anyone's name in creating this course, but if we have, please notify us and we will change the name in the next revision of the course. Logical Operations is an independent provider of integrated training solutions for individuals, businesses, educational institutions, and government agencies. The use of screenshots, photographs of another entity's products, or another entity's product name or service in this book is for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the book by nor any affiliation of such entity with Logical Operations. This courseware may contain links to sites on the Internet that are owned and operated by third parties (the "External Sites"). Logical Operations is not responsible for the availability of, or the content located on or through, any External Site. Please contact Logical Operations if you have any concerns regarding such links or External Sites.

TRADEMARK NOTICES

Introduction to Programming with Python®

About This Course

Though Python® has been in use for nearly thirty years, it has become one of the most popular languages for software development, particularly within the fields of data science, machine learning, artificial intelligence, and web development—all areas in which Python is widely used. Whether you're relatively new to programming, or have experience in other programming languages, this course will provide you with a comprehensive first exposure to the Python programming language that can provide you with a quick start in Python, or as the foundation for further learning.

You will learn elements of the Python 3 language and development strategies by creating a complete program that performs a wide range of operations on a variety of data types, structures, and objects, implements program logic through conditional statements and loops, structures code for reusability through functions, classes, and modules, reads and writes files, and handles error conditions.

Course Description

Target Student

This course is designed for people who want to learn the Python programming language in preparation for using Python to develop software for a wide range of applications, such as data science, machine learning, artificial intelligence, and web development.

Course Prerequisites

Some experience programming in an object-oriented language is helpful, but even if you don't have such experience, this course can be useful to those that are new to programming.

To ensure your success in the course, you should have at least a foundational knowledge of personal computer use. You can obtain this level of skills and knowledge by taking a course such as one of the following Logical Operations courses:

- *Using Microsoft® Windows® 10*
- *Microsoft® Windows® 10: Transition from Windows® 7*

Course Objectives

In this course, you will develop simple command-line programs in Python.

You will:

- Set up Python and develop a simple application.
- Declare and perform operations on simple data types, including strings, numbers, and dates.
- Declare and perform operations on data structures, including lists, ranges, tuples, dictionaries, and sets.

- Write conditional statements and loops.
- Define and use functions, classes, and modules.
- Manage files and directories through code.
- Deal with exceptions.

The CHOICE Home Screen

Logon and access information for your CHOICE environment will be provided with your class experience. The CHOICE platform is your entry point to the CHOICE learning experience, of which this course manual is only one part.

On the CHOICE Home screen, you can access the CHOICE Course screens for your specific courses. Visit the CHOICE Course screen both during and after class to make use of the world of support and instructional resources that make up the CHOICE experience.

Each CHOICE Course screen will give you access to the following resources:

- **Classroom**: A link to your training provider's classroom environment.
- **eBook**: An interactive electronic version of the printed book for your course.
- **Files**: Any course files available to download.
- **Checklists**: Step-by-step procedures and general guidelines you can use as a reference during and after class.
- **LearnTOs**: Brief animated videos that enhance and extend the classroom learning experience.
- **Assessment**: A course assessment for your self-assessment of the course content.
- Social media resources that enable you to collaborate with others in the learning community using professional communications sites such as LinkedIn or microblogging tools such as Twitter.

Depending on the nature of your course and the components chosen by your learning provider, the CHOICE Course screen may also include access to elements such as:

- LogicalLABS, a virtual technical environment for your course.
- Various partner resources related to the courseware.
- Related certifications or credentials.
- A link to your training provider's website.
- Notices from the CHOICE administrator.
- Newsletters and other communications from your learning provider.
- Mentoring services.

Visit your CHOICE Home screen often to connect, communicate, and extend your learning experience!

How to Use This Book

As You Learn

This book is divided into lessons and topics, covering a subject or a set of related subjects. In most cases, lessons are arranged in order of increasing proficiency.

The results-oriented topics include relevant and supporting information you need to master the content. Each topic has various types of activities designed to enable you to solidify your understanding of the informational material presented in the course. Information is provided for reference and reflection to facilitate understanding and practice.

Data files for various activities as well as other supporting files for the course are available by download from the CHOICE Course screen. In addition to sample data for the course exercises, the course files may contain media components to enhance your learning and additional reference materials for use both during and after the course.

Checklists of procedures and guidelines can be used during class and as after-class references when you're back on the job and need to refresh your understanding.

At the back of the book, you will find a glossary of the definitions of the terms and concepts used throughout the course. You will also find an index to assist in locating information within the instructional components of the book. In many electronic versions of the book, you can click links on key words in the content to move to the associated glossary definition, and on page references in the index to move to that term in the content. To return to the previous location in the document after clicking a link, use the appropriate functionality in your PDF viewing software.

As You Review

Any method of instruction is only as effective as the time and effort you, the student, are willing to invest in it. In addition, some of the information that you learn in class may not be important to you immediately, but it may become important later. For this reason, we encourage you to spend some time reviewing the content of the course after your time in the classroom.

As a Reference

The organization and layout of this book make it an easy-to-use resource for future reference. Taking advantage of the glossary, index, and table of contents, you can use this book as a first source of definitions, background information, and summaries.

Course Icons

Watch throughout the material for the following visual cues.

Icon	*Description*
	A **Note** provides additional information, guidance, or hints about a topic or task.
	A **Caution** note makes you aware of places where you need to be particularly careful with your actions, settings, or decisions so that you can be sure to get the desired results of an activity or task.
	**LearnTO** notes show you where an associated LearnTO is particularly relevant to the content. Access LearnTOs from your CHOICE Course screen.
	**Checklists** provide job aids you can use after class as a reference to perform skills back on the job. Access checklists from your CHOICE Course screen.
	**Social** notes remind you to check your CHOICE Course screen for opportunities to interact with the CHOICE community using social media.

1 Setting Up Python and Developing a Simple Application

Lesson Time: 3 hours

Lesson Introduction

Before you start developing programming projects in Python®, you must set up a computer with the tools you need. Once you have done so, you can create a new programming project, and start writing and testing your code.

Lesson Objectives

In this lesson, you will:

- Set up a Python development environment.
- Write Python statements.
- Create a Python application.
- Prevent errors.

TOPIC A

Set Up the Development Environment

In this topic, you will acquaint yourself with the major versions of Python, as well the various development environments available to you. With a development environment set up from the start, you'll be able to write code more easily and efficiently.

Types of Programs Developed in Python

Python is a general-purpose programming language and can be used to develop many different kinds of applications. Examples of the types of applications developed in Python include:

- Data analysis and forecasting tools
- Artificial intelligence
- Web apps
- Video games
- Audio/video apps
- Mobile apps
- Networking apps
- Scientific/mathematical apps
- Security utilities
- Educational apps

Python is one of the most popular programming languages in the world and many well-known organizations use Python in developing their software. Google™, National Aeronautics and Space Administration (NASA), and many others have used Python. Following is a list of apps you may be familiar with that have been developed wholly or in part using Python.

- Instagram®, a popular photo/video sharing social media app
- Spotify®, a popular music streaming service
- Dropbox®, a cloud-based file hosting service
- PyCharm®, a programming tool (used in this course)
- Django®, an open source web framework
- Blender™, a 3D art and animation app
- ERP5, powerful open source business software used in aerospace, banking, and government

History of Python

Python was created by Guido van Rossum in the late 1980s. He wanted to design a language that was easy to read and required minimal lines of code as compared to other languages. Early versions of the language were released over the next few years, but version 1.0 was not released until January of 1994.

Python 2 was the first major update to the Python programming language, released in October 2000. Python 2 improved upon the original Python by adding features like garbage collection, augmented assignment, list comprehension, and support for Unicode strings.

As of April 2020, the most recent version of Python 2 was Python 2.7.18. Although it was superseded by Python 3 several years ago, for various reasons some developers have stayed with Python 2. This is especially true in enterprise environments that rely on specific libraries that were not ported to Python 3.

Python 3 is the most recent major version of the language. Python 3.9.0 was released in late 2020 and is being continually updated.

Interpreters and Compilers

In computer programming, an ***interpreter*** is software that executes written code so that it runs on the operating system. This is similar to the role of a ***compiler*** in other high-level programming languages like C++ and C#.

Compilers and interpreters do a similar job. They must translate the ***source code*** (code written by a programmer in a programming language such as Python) into machine code that can run on the computer. However, a compiler will compile the code into machine-executable code (saved as a .exe file, for example) before the program runs. A user must then launch the executable file to run the program. Because the executable file is essentially ready to run directly on the machine, compiled programs tend to run very quickly. Interpreters, on the other hand, convert code into machine-executable code each time the user runs the program. Because of these differences, there are some tradeoffs between using compilers and interpreters.

There are some advantages in using a compiler. For example, because of the extra processing overhead needed at runtime, interpreted programs tend to run more slowly than compiled code. Also, in situations where code runs on an interpreter, errors that prevent code from being translated correctly into machine-executable code may be visible to the user when the program is run, whereas code processed on a compiler must be free of such errors before an executable will be created by the compiler. In other words, in order to provide an executable file to an end user, the source code must successfully compile. Of course, other types of errors may exist in a program that successfully compiles.

There are, however, advantages to using an interpreter. Interpreters are generally easier to use than compilers, particularly for those new to programming. Interpreted programs are generally more "portable" than compiled programs. For example, the same Python script might run on a Windows® or Linux® computer, whereas a compiled program would typically need to recompiled from the source code in order to run it on a different operating system. Interpreted programs may be more convenient to maintain and use in environments where software behavior must be frequently revised, such as web servers, process automation, data analysis, and so forth.

Python Implementations

Over the years, numerous interpreters and compilers have been developed for Python. Here is just a handful of the many examples in use today.

- **CPython** - The most widely used interpreter, with the interpreter itself developed using the C programming language.
- **Stackless Python** - A fork of CPython that allows you to run hundreds of thousands of small tasks concurrently.
- **PyPy** - An interpreter that translates code at runtime rather than just before the program is executed, making it faster than CPython in some implementations.
- **Jython** - An implementation of the Python language that runs on the Java Virtual Machine, enabling seamless use of third-party Java libraries and other Java-based applications.
- **ActivePython** - A distribution of Python bundled with many supporting packages, hardened for cybersecurity, and with other features to support enterprise application development.
- **IronPython** - An implementation of the Python language developed in C# and based on Microsoft's .NET framework, enabling it to interact with .NET objects.

CPython

The default interpreter implemented in Python is ***CPython***, written in the C programming language. CPython translates your Python code into ***bytecode***, and this bytecode is executed by the CPython virtual machine. CPython is compatible with most major operating systems, including Windows, OS X, and Linux distributions.

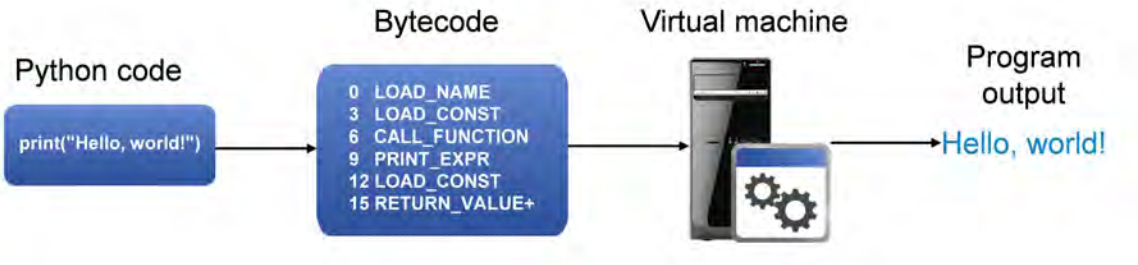


Figure 1-1: How CPython works.

IDLE

IDLE is the default cross-platform ***integrated development environment (IDE)*** that comes with Python.

The IDLE source code editor is very basic and is rarely used for serious coding. However, it does include the following features:

- Code completion
- Smart indentation
- Syntax highlighting

Along with a source code editor, IDLE also comes with a ***command shell*** for quick code execution and a ***debugger*** for finding and testing errors.

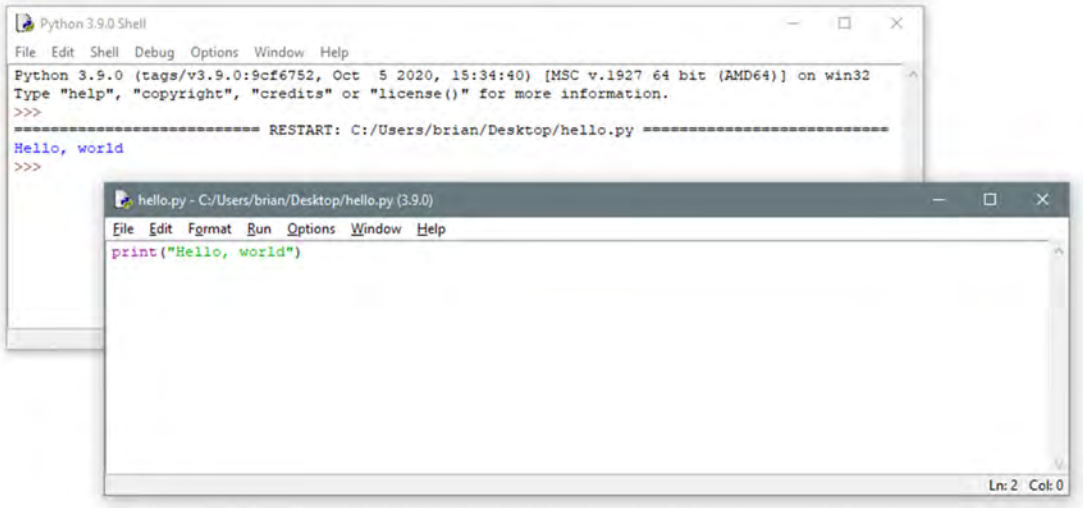


Figure 1-2: The IDLE shell and source code editor.

Other Python IDEs

There are several IDEs for Python that are more powerful than IDLE. Most of these IDEs share the following features:

- Code completion
- Smart indentation
- Code refactoring
- Syntax highlighting
- Error indicators
- Project navigation

- Customizable user interface
- Debugger

Following are various popular Python development environments.

- **PyCharm** - A cross-platform IDE delivered in a free Community Edition and commercial Professional Edition. Includes an intelligent code editor with features such as debugging, testing, profiling, deployments, remote development, database tools, and Python Console.
- **Jupyter Notebook** - A cross-platform, free web application that enables you to create and share documents that contain live code, equations, visualizations, and narrative text. This application is geared toward data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and education.
- **Python Tools for Visual Studio For Windows** - A free plugin for Visual Studio®, developed by Microsoft® to provide rich support for all actively supported versions of Python. The tool supports Visual Studio's extensive feature set, including IntelliSense®, debugging, code navigation, formatting, cleaning, and refactoring, variable explorer, test explorer, code snippets, and working with Jupyter notebooks.
- **Komodo IDE** - An advanced cross-platform, proprietary editor developed by ActiveState®. It provides a rich feature set including tracked changes, multiple selections, bookmarking, code folding, smart language detection and code intelligence, debugging, split views, variable highlighting, code snippets, macros, and user scripts.
- **Wing IDE** - A comprehensive cross-platform, proprietary Python IDE, including customizable code editor, code intelligence, debugging, unit testing, and project management.
- **PyDev** - A cross-platform, free third-party plugin for Eclipse. Similar to Python Tools for Visual Studio, it adds Python development capabilities to a feature-rich, general-purpose IDE (Eclipse), so Python developers can take advantage of the rich feature set provided by Eclipse.
- **Spyder** - A cross-platform, free, development environment designed primarily for scientific programming. It provides advanced features for editing and debugging Python code, and exploration of datasets, and is designed to be customized and extended.

IDE Configuration

As a Python developer, you might have to work with programs written in various versions of Python, and would therefore have multiple versions of Python installed on your development computer. You also have a choice when it comes to which IDE you will use.

Python interpreters and IDEs are typically separate software products, installed separately. For this reason, when you are setting up a Python development environment, it is often necessary to configure the IDE to be able to find the various interpreters installed on the computer.

Python Projects

Programming projects written in Python can become large and complex and may contain multiple files. To keep things organized, they are often stored in a project folder. IDEs such as PyCharm typically include features that enable you to automatically create a new project folder when starting a programming project.

In addition to keeping things organized, the ***project folder*** has another purpose—to store support files that enable you to configure the programming project with important information, such as which version of Python should be used to interpret your source code.

Code written for one version of Python will not necessarily run on a different version of Python. To enable your software development environment to know which Python interpreter to use, when you create a new software project, you need to configure it to use a particular interpreter. This ensures that the Python source code you are writing will be processed by the interpreter (and version of the Python language) that you intended to use.

Note: The PyCharm IDE automatically creates a subdirectory named *.idea* within a new project folder. This subdirectory contains various files (automatically maintained by PyCharm) that store information that you configure, such as which interpreter should be used to compile your program.

ACTIVITY 1-1
Running a Python Script

Before You Begin
Python 3.9.0 and the course data files have been installed on your computer.

Scenario
The Fuller & Ackerman (F&A) publishing firm publishes a wide variety of books in various formats. The Editing department has decided to use computer programs to perform initial analysis of manuscripts submitted by authors. Eventually, a wide range of analyses will be performed, but some software requirements have been identified for a prototype application that you will develop in Python.

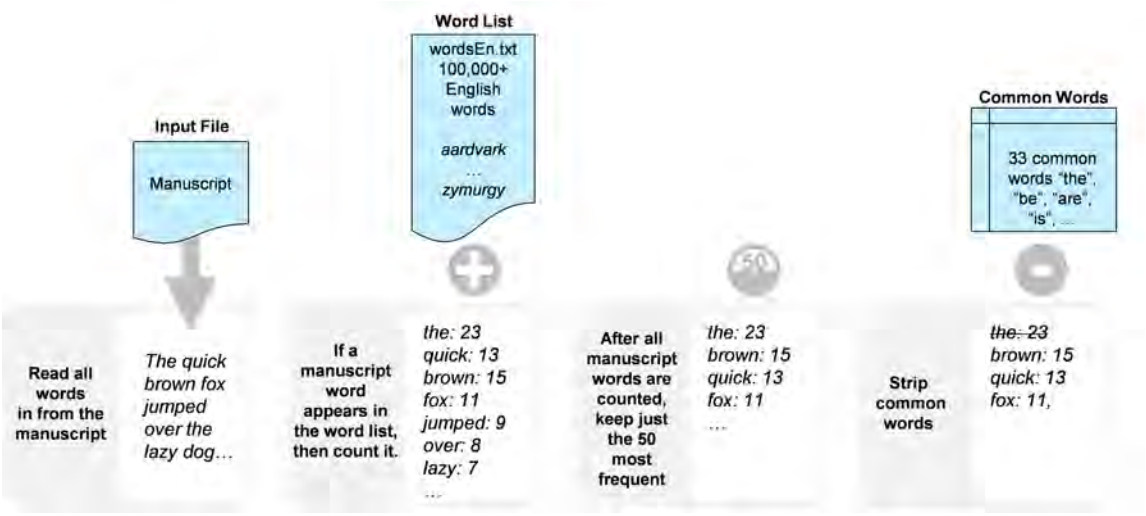


Figure 1-3: Data resources used in the WordCount program.

As shown in this figure, the program will use three primary data resources to perform the analysis.

- **Input File** - Text file from which the manuscript is read into the program for analysis.
- **Word List** - Text file containing more than 100,000 English words. If any word in the manuscript is also in the word list, that word will be counted. When words in the entire manuscript have been counted, the top 50 words will be pulled into a report.
- **Common Words** - A list of commonly used words. The user has the option of further culling the list by stripping the most common words from the top 50.

Some initial analysis has been done, and a list of requirements has been produced, along with a flowchart showing how the program might be implemented.

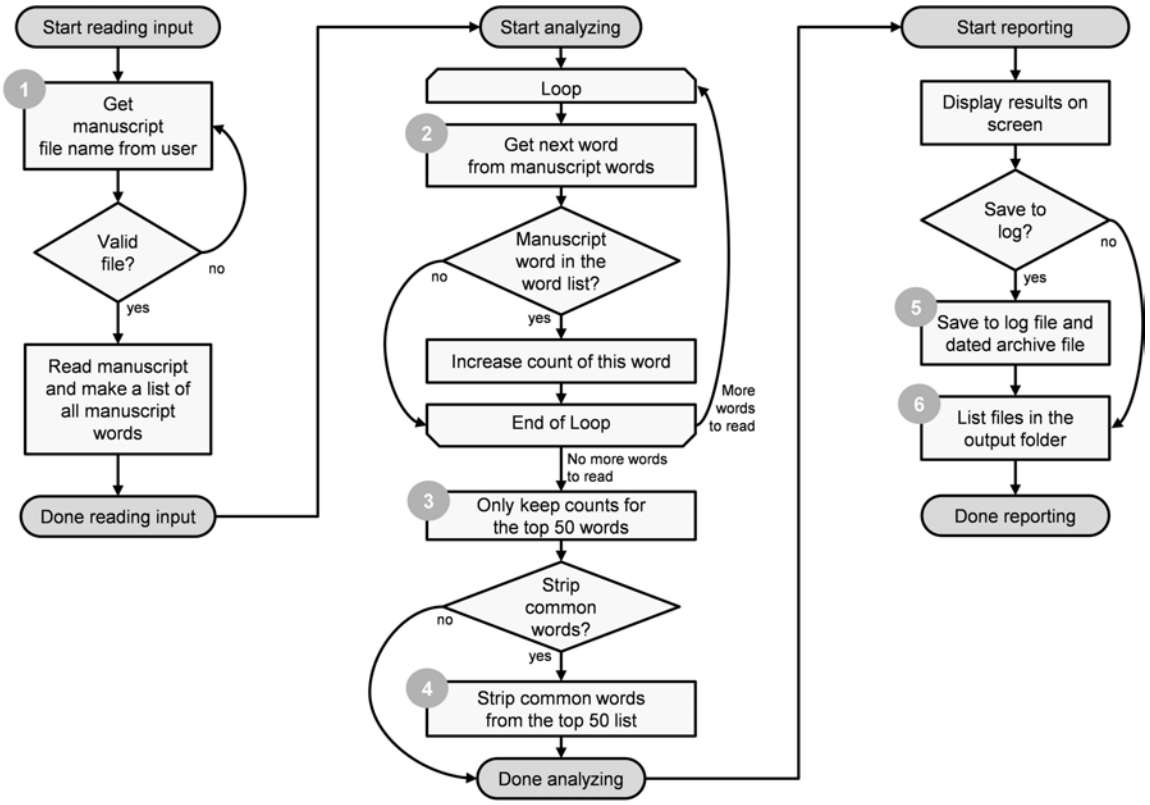


Figure 1-4: Flowchart of the WordCount program.

Program Requirements

1. **Prompts the user to specify an input file that contains the manuscript to be processed.**
   - The file can be specified in any location by entering the path and file name.
   - If no path is entered, the file is assumed to be in the WordCount program directory.
2. **Reads the manuscript file and counts how many times that each word appears.**
   - Manuscript words are counted only if they appear within an approved word list.
   - The word list is stored in the program directory in the wordsEn.txt text file.
3. **Identifies the top 50 words in the manuscript.**
   - Only the 50 words that appear most frequently are included in the report.
   - The report shows each word and the number of times it appears in the manuscript.
4. **Enables the user to strip common words from the results.**
   - The user has the option of excluding common words from the top 50 results.
   - The common words include: the, be, are, is, were, was, am, been, being, to, of, and, a, in, that, have, had, has, having, for, not, on, with, as, do, does, did, doing, done, at, but, by, from.
5. **Enables the user to save the results.**
   - Once the results are shown on screen, the user is given the choice of saving the results.
   - Results are saved as a text file in a desktop directory named "Wordcount Output".
   - The file name is based on the original file—e.g., ms_results.txt for the ms.txt manuscript.
   - The log file is copied to a dated archive file—for example, ms_archive_2020-08-07_15.42.05.txt.
6. **Displays a list of all files contained in the Wordcount Output folder.**
   - A directory listing of files in the output folder is shown to the user after the log file and archive file are saved.

To see an example of a working Python program and to get familiar with the application you will create in this course, you will start by running a copy of the finished program.

1. Start a Windows command prompt console.

a) Select the **Start** button to show the Windows **Start** menu.
b) Type ***cmd*** to search for the **Command Prompt** program.
c) Select the **Command Prompt** tile to load the Windows Command Prompt window.

2. Examine the project files.

a) In the **Command Prompt** console, type **`cd \094021Data`** and press **Enter**.

This changes the current working directory to the location where the course data files are located. If they have been set up in a different location, your instructor will guide you through the steps to navigate to the right location.

b) Type **`dir`** and press **Enter** to see a directory listing of this location.

```
Microsoft Windows [Version 10.0.18363.959]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\brian>cd \094021Data

C:\094021Data>dir
 Volume in drive C has no label.
 Volume Serial Number is 7034-8D4C

 Directory of C:\094021Data

08/11/2020  10:03 AM    <DIR>          .
08/11/2020  10:03 AM    <DIR>          ..
08/10/2020  05:31 PM    <DIR>          Dealing with Exceptions
08/10/2020  05:31 PM    <DIR>          Processing Data Structures
08/10/2020  05:31 PM    <DIR>          Processing Simple Data Types
08/11/2020  10:04 AM    <DIR>          Setting Up Python and Developing a Simple Application
08/10/2020  05:32 PM    <DIR>          Snippets
08/10/2020  05:32 PM    <DIR>          Structuring Code for Reuse
08/10/2020  05:32 PM    <DIR>          Writing Code to Process Files and Directories
08/10/2020  05:32 PM    <DIR>          Writing Conditional Statements and Loops in Python
               0 File(s)              0 bytes
              10 Dir(s)  419,385,597,952 bytes free

C:\094021Data>
```

This lists the files and subdirectories contained in the current working directory. Each directory contains the files for a lesson in the course. The project that contains the script you want to run is located within the "Setting Up Python and Developing a Simple Application Folder". This is a long name for a directory, but to reduce some typing you can use a shortened version of that name to navigate into it.

c) Type **`dir /x`** and press **Enter**.

```
C:\094021Data>dir /x
 Volume in drive C has no label.
 Volume Serial Number is 7034-8D4C

 Directory of C:\094021Data

08/11/2020  10:03 AM    <DIR>                       .
08/11/2020  10:03 AM    <DIR>                       ..
08/10/2020  05:31 PM    <DIR>          DEALIN~1     Dealing with Exceptions
08/10/2020  05:31 PM    <DIR>          PROCES~1     Processing Data Structures
08/10/2020  05:31 PM    <DIR>          PROCES~2     Processing Simple Data Types
08/11/2020  10:04 AM    <DIR>          SETTIN~1     Setting Up Python and Developing a Simple Application
08/10/2020  05:32 PM    <DIR>                       Snippets
08/10/2020  05:32 PM    <DIR>          STRUCT~1     Structuring Code for Reuse
08/10/2020  05:32 PM    <DIR>          WRITIN~1     Writing Code to Process Files and Directories
08/10/2020  05:32 PM    <DIR>          WRITIN~2     Writing Conditional Statements and Loops in Python
               0 File(s)              0 bytes
              10 Dir(s)  419,382,202,368 bytes free
```

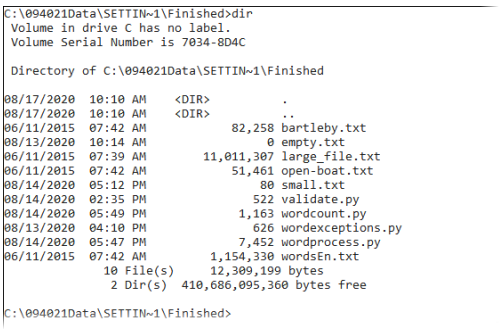
Shortened alternatives to the long directory names are now shown. The short name for the directory you want to navigate into is "SETTIN~1".

d) Type **`cd settin~1`** and press **Enter**.
e) Type **`dir`** and press **Enter**.

There are two subdirectories, Errors and Finished.

f) Type **`cd finished`** and press **Enter**.

g) Type **dir** and press **Enter**.

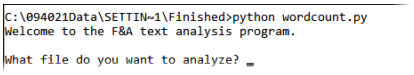


This directory contains a finished version of the project you will create in this course. There are various text files and two Python scripts.

- **bartleby.txt**, **large_file.txt**, **open-boat.txt**, and **small.txt** are text files containing various book manuscripts that you can pass in to the WordCount program for analysis.
- **wordcount.py** is a Python script that contains the actual program you will run.
- **wordexceptions.py** and **wordprocess.py** are supporting files that contain additional Python code used by the program in wordcount.py.
- **wordsEn.txt** is a text file that contains a list of over 100,000 common English words that will be searched for within the manuscript when performing an analysis.

3. Run a completed Python script.

a) Type **python wordcount.py** and press **Enter**.



- This issues the `python` command, which loads the Python interpreter.
- Because you passed in the name of the script `wordcount.py`, that script is loaded into the Python interpreter, translated into machine code, and run.
- The program displays a welcome message, then prompts you for the name of the file you want to analyze.

b) Type **bartleby.txt** and press **Enter**.

Progress messages are shown as the program loads the file and analyzes it. A prompt is displayed, asking whether you would like to exclude common words from the results.

c) Type **y** and press **Enter**.

- After some processing, the word counts are displayed in the console window, with some output possibly scrolling out of view.
- You are asked whether you want to output the results to a file.

```
C:\094021Data\SETTIN~1\Finished>python wordcount.py
Welcome to the F&A text analysis program.

What file do you want to analyze? bartleby.txt
  Reading file, one moment...
  File read successfully.

Strip common words from the results? (Y/N) y
  Compiling results, one moment ... read 14361 words from manuscript file.

Results for bartleby.txt
  his: 220 times
  he: 213 times
  my: 196 times
  it: 156 times
  me: 138 times
  him: 137 times
  you: 105 times
  would: 96 times
  this: 68 times
  upon: 66 times
  all: 64 times
  said: 57 times
  no: 52 times
  so: 52 times
  one: 51 times
  an: 48 times
  any: 47 times
  what: 45 times
  which: 45 times
  or: 44 times
  if: 43 times
  some: 42 times
  there: 42 times
  then: 42 times
  will: 41 times
  now: 36 times
  man: 35 times
  office: 35 times

Would you like to output these results to a file? (Y/N)
```

In a moment, you will respond to the input prompt and have the program produce an output file, which will be stored in the **Wordcount Output** folder. If that folder doesn't already exist on the desktop, then the program will create it for you.

d) Before you finish running the program, position the Command Prompt window where it will not obstruct your view of the left side of your Windows Desktop (the area where This PC, Recycle Bin, and other icons are normally located).

e) Type **y** and press **Enter**.

```
  now: 36 times
  man: 35 times
  office: 35 times

Would you like to output these results to a file? (Y/N) y
  Log file saved to bartleby_results.txt
  Archive file saved to bartleby_results_backup_2020-08-17_10.59.53.txt

The C:\Users\brian\Desktop\Wordcount Output output folder now contains:
  bartleby_results.txt
  bartleby_results_backup_2020-08-17_10.59.53.txt

C:\094021Data\SETTIN~1\Finished>
```

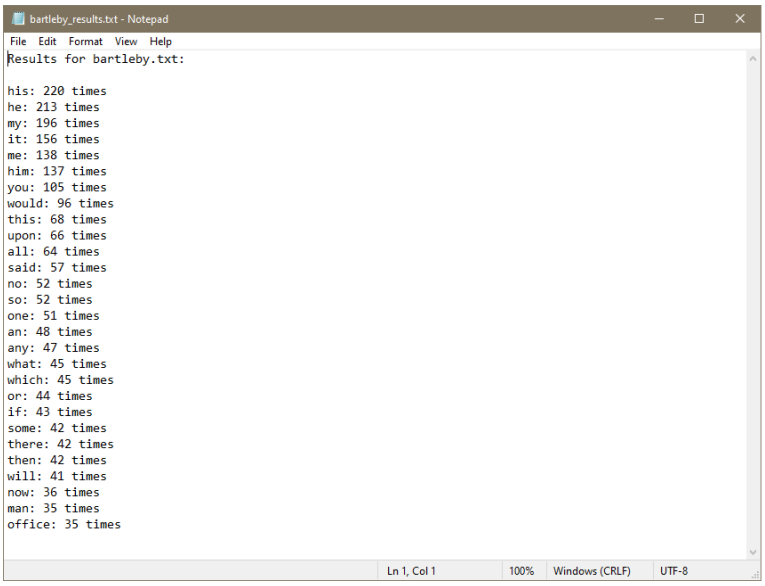
The **Wordcount Output** folder is automatically created on the desktop, and the program lists the files now contained in that folder.

f) In the Command Prompt window, type **exit** and press **Enter**.

4. Examine the output files.

a) On the desktop, open the **Wordcount Output** folder.

b) Double-click **bartleby_results.txt** and examine its contents.



The word counts have been saved in the file.

c) Exit the text editor.

d) Double-click the dated backup file and examine its contents.

- This file is an exact copy of the results file.
- If you were to run wordcount.py again, the new results would overwrite the contents of the bartleby_results.txt file. However, the new backup file would have a different time in its file name, so the previous dated backup file would remain intact.

e) Close the editor where you are viewing the backup file and the File Explorer window where you viewed the contents of the **Wordcount Output** folder.

f) On the desktop, delete the **Wordcount Output** folder.

ACTIVITY 1-2

Setting Up the Python Development Environment

Before You Begin

Python 3.9.0 and PyCharm Community Edition are installed on your computer.

Scenario

As you have seen, you can run a Python program directly from the Windows Command Prompt console. With a simple text editor like Windows Notepad, you can create your own Python applications and run them from the command line. However, there are benefits to using a Python integrated development environment like PyCharm, particularly when you are first learning Python.

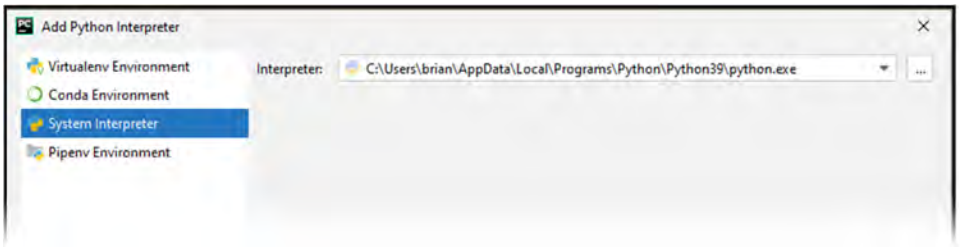
The Python 3.9.0 interpreter and the PyCharm IDE have both been installed on your computer. Before you start developing any project, you need to configure the IDE with the directory paths to any Python interpreters that you plan to use for development. Then when you create a new project, you can identify the specific Python interpreter you will use to develop your program.

Note: If PyCharm displays a message informing you that Windows Defender may be slowing down PyCharm's performance, select **Actions→Don't show again** to allow Windows Defender to continue scanning your Python projects. (Performance of the small projects used in this course will not be a problem.)

1. Set up the default Python interpreter.
   a) From your Windows desktop, double-click the **PyCharm Community Edition 2020.2.3 x64** shortcut.
   b) Select **Configure**, and from the menu that is shown, select **Settings**.
   c) In the left pane, select **Python Interpreter**.
   d) In the right pane, select the **Settings** button.

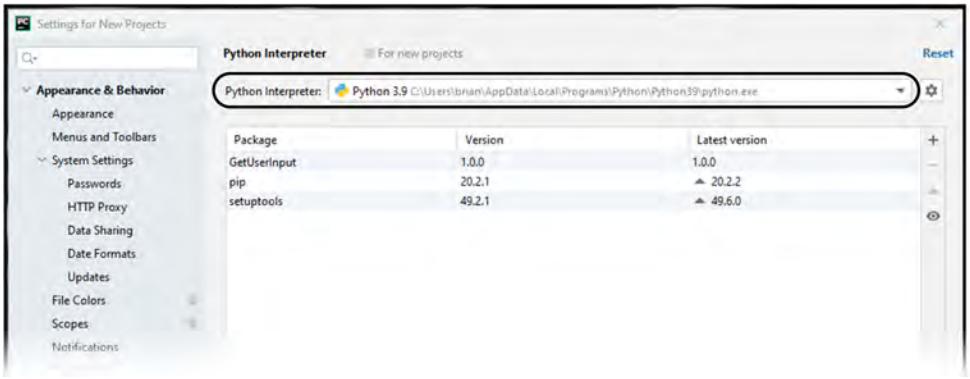
   e) Select **Add**.
   f) In the left pane, select **System Interpreter**.

g) If the Python 3.9 interpreter installed on your computer is not already shown in the **Interpreter** text box, follow your instructor's directions to navigate to it and select it.



h) Select **OK** to return to the **Settings for New Projects** dialog box.

i) Select the **Python interpreter** drop-down list, and select the Python interpreter that you just added.



This specifies the default interpreter that will be assigned to new projects.

j) Select **OK** to return to the **Welcome to PyCharm** dialog box.

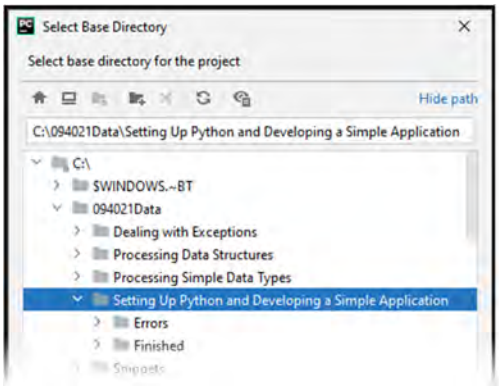
2. Create a new project.

a) In the **Welcome to PyCharm** window, select **New Project**.

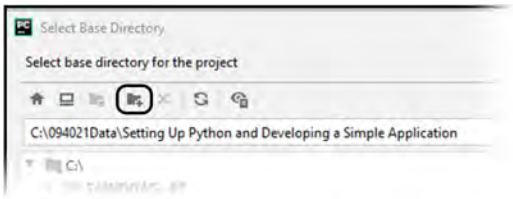
b) In the **New Project** dialog box, select the browse button on the right end of the **Location** field.

The **Select Base Directory** dialog box is shown. You can use this dialog box to identify where you will save the new project.

c) Select the **C:\094021Data\Setting Up Python and Developing a Simple Application** directory as shown.



d) Select the **New Folder** button.



You are prompted to enter a name for the new folder.

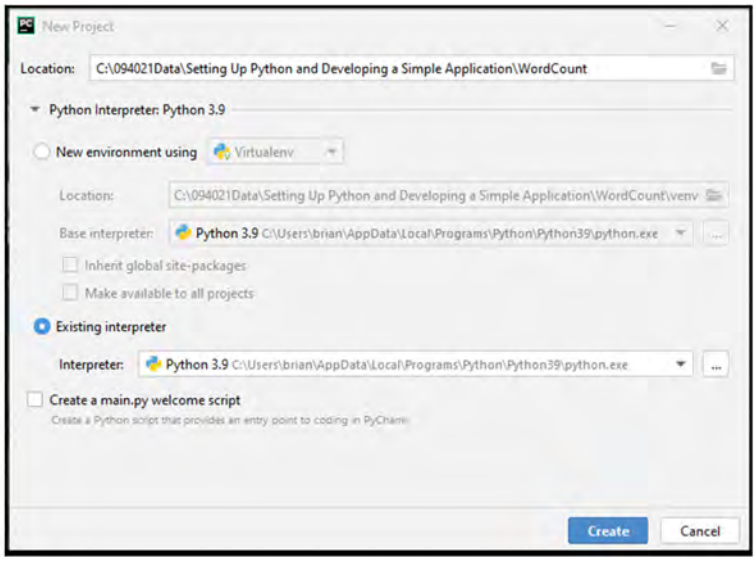
e) For the new folder name, type *WordCount* and select **OK**.
The **WordCount** folder is created and automatically selected.

f) Select **OK** to specify the **WordCount** folder you just created as the base directory for the project.
The location is shown in the **New Project** dialog box.

g) Select **Existing interpreter**.

h) Uncheck the option to **Create a main.py welcome script**.

i) Confirm the location where the project will be saved, and the interpreter that will be used to run it.

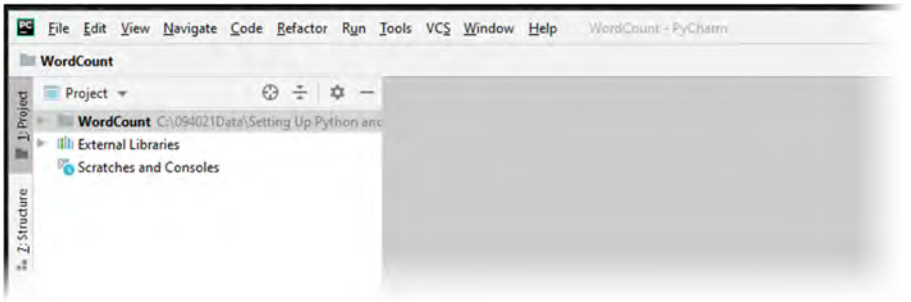


- The project will be saved in the new directory you created.
- The default interpreter you specified earlier will be used.

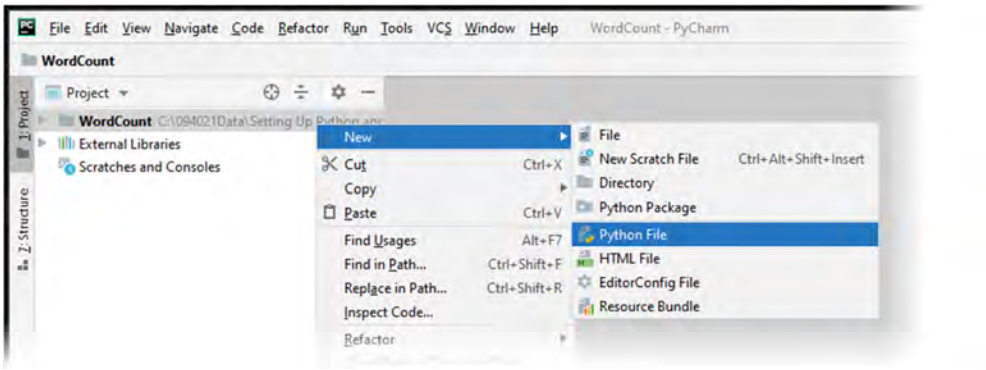
j) Select **Create** to create the project.
k) If the **Tip of the Day** window is shown, check the **Don't show tips** check box and select **Close**.
l) If the **Windows Defender might impact performance** message is shown, select **Actions** and **Don't show again**.

3. Create a Python file.

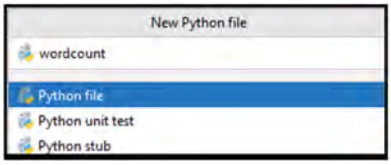
a) Verify that **WordCount** is listed in the **Project** pane on the left side of the PyCharm program.



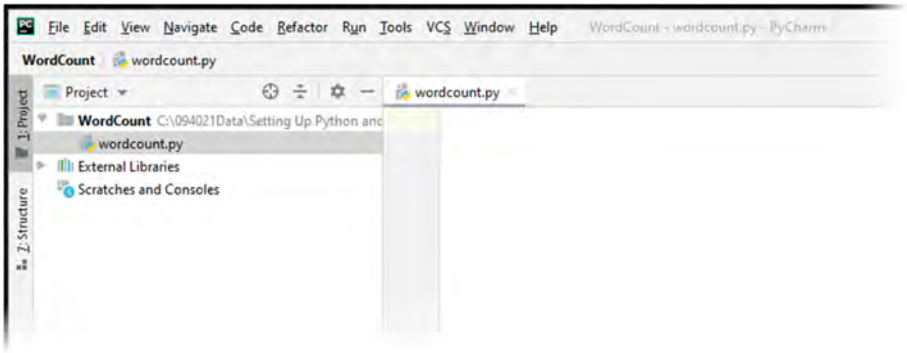
b) Right-click **WordCount** and select **New→Python File**.



c) In the **New Python File** dialog box, in the **Name** text box, type ***wordcount*** and press **Enter**.



d) Verify that the editor pane is open to your newly created file, **wordcount.py**.



The wordcount.py file is currently empty. This is where you will begin typing your Python code in the activities to follow.

TOPIC B

Write Python Statements

Now that you have your development environment set up and ready to go, you can begin writing Python code. Before you tackle a full-fledged application, however, you should become familiar with how Python works by executing basic and common statements.

Interactive Mode

Most programs are written in a source code editor and saved to a file. The Python interpreter then executes the code in this file all at once. This is considered the normal mode of operation. However, you can also invoke the interpreter in ***interactive mode***. The interpreter is in interactive mode when it presents a command prompt/shell to the user. The user is able to type statements directly into the interpreter and receive immediate feedback for each statement. This is true of statements that wouldn't otherwise provide feedback if run in normal mode. For example, running `2 + 2` in the interactive shell will return `4` as a result. Executing that same statement and nothing else in normal mode will not return anything, despite it running the computation.

Interactive mode uses >>> to signify that it is ready to receive a command from the user. After typing a statement and pressing **Enter**, it will return a value, if applicable. Either way, it will offer another >>> prompt to the user. Interactive mode also allows you to type in multi-line statements; the . . . symbol signifies a prompt that is continued from the previous one.

Interactive mode is useful as a way to test out certain commands without having to create an entirely new file for the interpreter to run. The feedback that the interactive shell provides is also beneficial for debugging snippets of code.

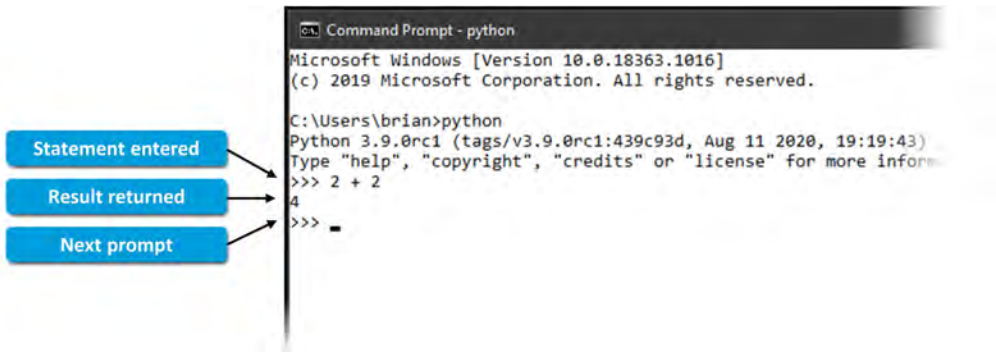


Figure 1-5: Python running in interactive mode.

Help

While in interactive mode, you can invoke the `help()` function in order to search Python's documentation. This is particularly useful if there's a topic you want more information on, or if there's an object in your code you want to learn more about. To begin an interactive help session, simply enter `help()`. You'll then be greeted by the help utility and from here you can search more about specific objects and topics.

You can also directly access help information about objects that you've defined. To do this, enter `help(object)` at the interactive shell, replacing `object` with whatever object you want more information on.

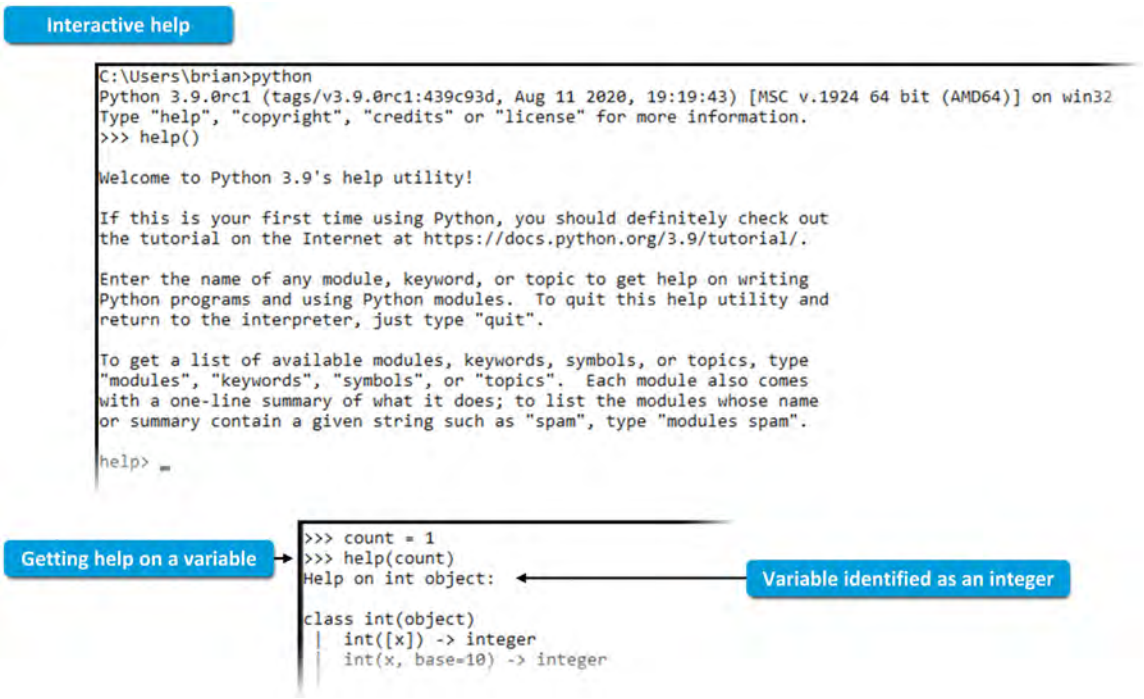


Figure 1-6: The interactive help utility.

Python Syntax

All programming languages have a ***syntax***, or the rules that define how you write the code. Each language has its own syntax, but many share a few commonalities. Python's syntax has a lot in common with languages like C, Java™, and Perl. Nevertheless, there are differences. For example, Perl's design philosophy suggests that there should be more than one way to write code to accomplish the same task. Python's philosophy is that there should be only one obvious way to do something.

Another core principle of Python is that it should be easily readable. Whereas many languages use curly braces `{}` to block off code in multiple lines, Python makes explicit use of indentation. In Python, you are required to indent certain blocks of code. The amount of spaces or tabs you use to create the indent can vary, but you must be consistent in each block. For example, observe the indentation in the following code:

```
if True:
    print("True")
else:
    print("False")
```

Another way that Python attempts to improve readability is by using English words where other languages would use punctuation. Python's simplistic syntax makes it an excellent first programming language for beginners.

Everything Is an Object

Some object-oriented programming languages treat only certain elements as discrete objects. In Python, however, everything is an object. This means that everything from string literals to functions can be assigned to a variable or passed in as an argument.

Variables and Assignment

A ***variable*** is any value that is stored in memory and given a name or an ***identifier***. In your code, you can assign values to these variables.

Many programming languages, like C, require you to define the type of variable before you assign it to a value. Examples of types include integers, floats, strings, and more. Essentially, these types define exactly what kind of information the variable holds. However, in Python, you don't have to declare variable types. Instead, once you assign a value to a variable, that type is defined automatically.

To assign a value to a variable, you use an equals sign (=). The element to the left of the = is the identifier or name of the variable, and the element to the right of the = is its value. Take a look at the following code:

```
count = 1
```

The variable is named `count` and it is assigned a value of `1`. Because 1 is an integer, Python knows to consider `count` an integer type of variable.

Formatting Variable Names

Based on Python's style guide, you should always define variables in lowercase format. If necessary, you can improve readability by separating words with an underscore.

- **Correct**: `my_variable = 1`
- **Incorrect**: `MyVariable = 1`

It's good practice to give your variables meaningful names in order to avoid ambiguity. You also cannot begin variable names with a number. Python will produce a syntax error.

Constants

Constants are identifiers with values that should not be changed. In Python, create a constant the same way you create a variable, but signify that it should be treated as a constant value by using uppercase letters.

- **Correct**: `PI_CONSTANT = 3.14159`
- **Incorrect**: `pi_constant = 3.14159`

Note: Creating a variable with an uppercase name does not prevent its value from being changed. It is simply a reminder to the programmer. You may be familiar with other programming languages that enable you to define a constant value that cannot be changed. In Python, you store such values in a variable, and you must be careful not to inadvertently change the value once you've initialized it.

Reserved Words

Although you can typically use whatever words you like to identify your variables and other objects, some words are reserved by Python. If you try to use any of these ***reserved words*** to define your own objects, Python will give you an error and your code will fail to execute. The following table lists the reserved words.

False	elif	lambda
None	else	nonlocal
True	except	not
and	finally	or
as	for	pass
assert	from	raise
break	global	return
class	if	try
continue	import	while

def	in	with
del	is	yield

These words are case sensitive. You could conceivably create a variable named `true` or `For`, but this is a poor coding practice.

Functions

A ***function*** is a block of code that you can reuse to perform a specific task. This is a vital part of writing efficient code, as calling a function can save you from having to write out the same or similar code over and over. You can define your own functions and Python has a number of built-in functions that you can call at any time.

Like variables, you define a function with a unique identifier. After this identifier, you must place open and close parentheses `()`. For example, the help utility mentioned earlier, `help()`, is a function.

Note: Defining and calling functions will be discussed in a later lesson.

Note: For a complete list of built-in functions, navigate to **https://docs.python.org/3.9/library/functions.html**.

Formatting Function Names

Like variable names, functions should be formatted in lowercase, with underscores between words.

- **Correct**: `read_input_file()`
- **Incorrect**: `READ_INPUT_FILE()`

Arithmetic Operators

Operators are objects that can evaluate expressions in a variety of ways. The values that are being operated on are called the ***operands***. A simple example is in the expression `2 + 4`. The `+` symbol is the operator, while `2` and `4` are the operands.

Operators can be grouped in several different ways. One such group is ***arithmetic operators***.

Operator	*Definition*	*Example*
`+`	Adds operands together.	`2 + 4` will return `6`.
`-`	Subtracts the operand to the right from the operand to the left.	`4 - 2` will return `2`.
`*`	Multiplies operands together.	`2 * 4` will return `8`.
`/`	Divides the left operand by the right operand.	`10 / 2` will return `5`.
`%`	Divides the left operand by the right operand and returns the remainder. This is called ***modulo***.	`13 % 4` will return `1`.
`**`	Performs exponential calculation using the left operand as a base and the right operand as an exponent.	`2 ** 4` will return `16`.

Note: This is not an exhaustive list of arithmetic operators. For a list of Python's built-in data types and the operations that can be performed on them, navigate to **https://docs.python.org/3/library/stdtypes.html**.

Order of Operations

When programming languages like Python evaluate expressions, they don't always do so from left to right. Instead, certain operators are executed before others. This ***order of operations*** determines what parts of a complex expression are acted on first, second, third, and so on. For example, without a clear order of operations, this expression:

```
2 + 4 * 8
```

can be evaluated one of two ways:

- Evaluated left to right, the answer is `48`.
- Evaluated with multiplication first, the answer is `34`.

Python has the following order of operations, from first to last:

- `**`
- `* / %`
- `+ -`

As you can see, some operators are on the same level as others. In deciding between these, Python evaluates from left to right. For example, `4 / 2 * 6` will result in `12`. This is because both multiplication and division are at the same order, so Python is evaluating `4 / 2` first, then multiplying that result by 6.

When you chain multiple exponents together, such as in `4 ** 3 ** 2` (262,144), Python actually evaluates these from right to left.

In these types of instances, you can also force Python to evaluate certain chunks of an expression over others. You can do this by wrapping the chunk in parentheses. For example, `4 / (2 * 6)` will now result in `.333` repeating because the parentheses are telling Python to evaluate `2 * 6` first.

A comprehensive summary of operator precedence is provided at **https://docs.python.org/3/reference/expressions.html#operator-summary**.

The print() Function

The `print()` function is a built-in function that outputs a given value to the command line. As it is a function, it must end with open and closed parentheses. Inside these parentheses, you can place whatever you wish to output. For example, `print("Hello, world!")` will output the text "Hello, world!". The `print()` function can also output other data types, as well as the value of variables. Take this code:

```
a = 1
print(a)
```

You are assigning variable `a` to an integer with a value of `1`. You then print the value of `a`, so the number `1` is output to the command line. The following is what it looks like running from interactive mode.

Figure 1-7: The print() function in interactive mode.

Note: In Python 2, `print` is a statement, not a function. It does not require parentheses. For example: `print a`.

Quotes and String Literals

A ***string literal*** is any value that is enclosed in single (') or double (") quotation marks. Which you choose is up to you, but you must be consistent within the string itself. For example:

```
a = "Hello"
b = 'Hello'
```

Both `a` and `b` are the same. `a = "Hello'` will cause an error because the string literal is not enclosed properly. However, you can still use both characters within a properly enclosed string literal. For example, `a = "Won't"` is acceptable because it is enclosed by consistent double quotation marks.

You can also use triple quotes of either variety (`'''` or `"""`) to enclose multiple lines of a string literal. For example:

```
a = """Hello,
world!"""

print(a)
```

This will print "Hello," and "world!" on separate lines.

Escape Codes

When creating string literals, there are times when you'll want to include characters that are difficult to represent. For example, say you want to include double quotes ("") within the string literal itself, while still wrapping it in double quotes:

```
a = "This is a "string literal.""
```

This will cause a syntax error because Python thinks the string literal has ended at the second instance of the double quotes, but the text continues. This is where ***escape codes***, also known as ***escape characters***, come in handy. Python interprets escape codes in a string literal as a certain command, and executes that command on the string. Using double quotes as an example:

```
a = "This is a \"string literal.\""
print(a)
```

The `\"` character is the escape code, and this particular escape code is telling Python to add a double quote at two different locations. The output would be:

```
This is a "string literal."
```

All escape codes begin with a backslash (\). The following table lists some of the escape codes in Python.

Escape Code	*Adds A...*
`\\`	Backslash character.
`\'`	Single quote character.
`\"`	Double quote character.
`\b`	Backspace character.
`\f`	Page break character.
`\n`	Line break character.
`\r`	Carriage return character.
`\t`	Horizontal tab character.

Escape Code	Adds A...
\v	Vertical tab character.

Note: For more Python escape codes, navigate to **https://docs.python.org/3.9/reference/lexical_analysis.html#string-and-bytes-literals**.

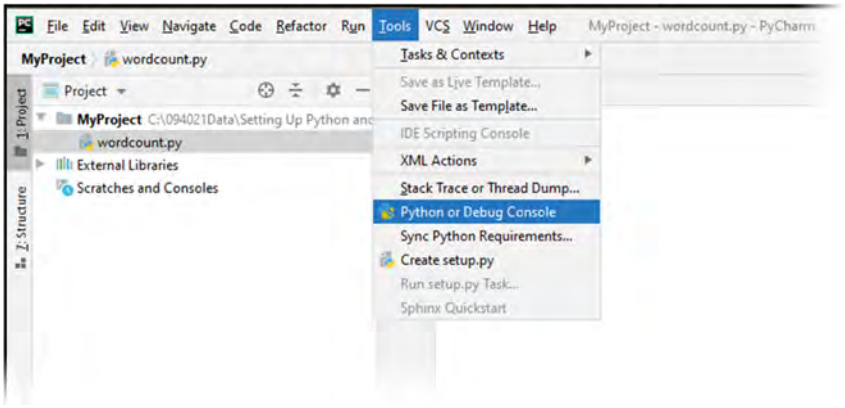
ACTIVITY 1-3
Writing Python Statements

Before You Begin
PyCharm is open. You will use the interactive shell for this activity.

Scenario
Now that you have your environment set up, you're ready to begin writing Python code. A good place to start practicing your code is from the interactive shell, as this will give you immediate feedback. In this activity, you'll work with some of the fundamentals of coding that all apps are built on—creating variables, assigning values, performing operations, and outputting text.

---

1. Open the interactive shell.
   a) From the PyCharm menu, select **Tools→Python or Debug Console**.



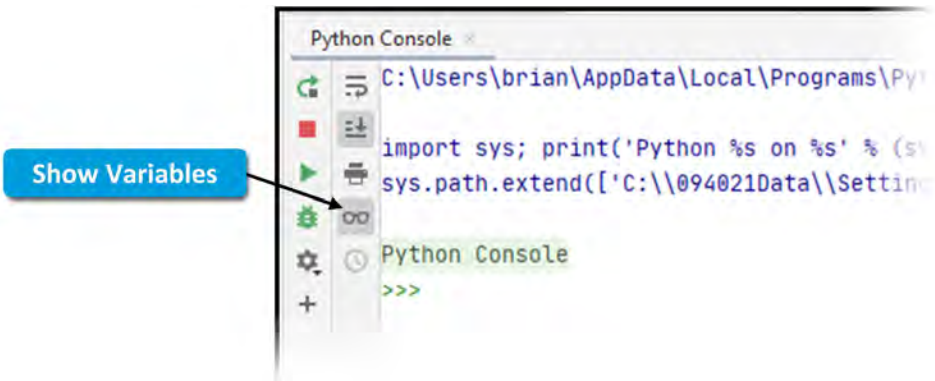
	Note: If you are prompted with a Windows Security Alert, select **Allow access** to allow PyCharm to access your network.

b) Examine the Python Console shown at the bottom of the PyCharm window.

- The Python Console provides an interactive shell, a text-based console in which you can type commands and see results immediately.
- The variable list is a debugging tool that provides status information about data values you are working with.

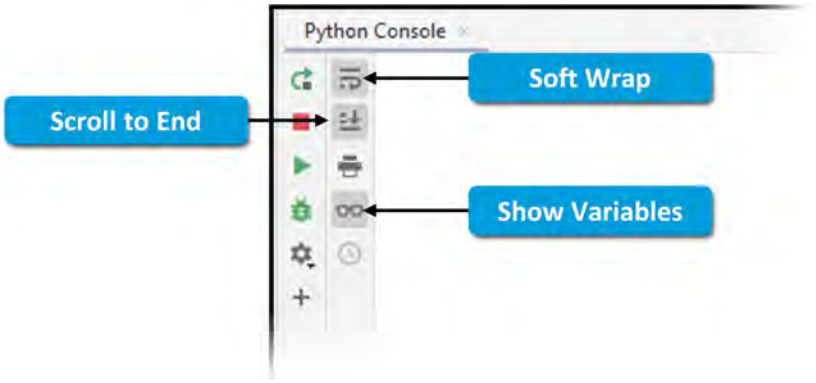
2. Configure the layout of the Python Console.

a) Select the **Show Variables** button a few times, and observe the effect it has on the layout of the Python Console.



This is a toggle button that changes the visibility of the variable list, like an on/off switch.

b) Make sure the three toggle buttons are selected as shown.



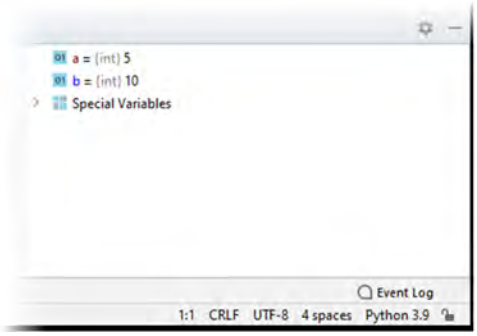
- Enabling the **Soft Wrap** setting causes lines of text in the Python Console to wrap around when it reaches the right edge. If this option were not selected, you might have to scroll horizontally to see text that has run past the right edge.
- Enabling the **Scroll to End** setting causes the Python Console to scroll automatically as new text is output to the console to ensure you're always looking at the most recent prompts and messages.
- Enabling the **Show Variables** setting displays the values of variables in your program as you run it, which can be helpful for testing and debugging.

3. Define variables and assign values to them.

a) At the command prompt, type `a = 5` and press **Enter**.

b) At the command prompt, type `b = 10` and press **Enter**.

c) Observe the current state of the variables.

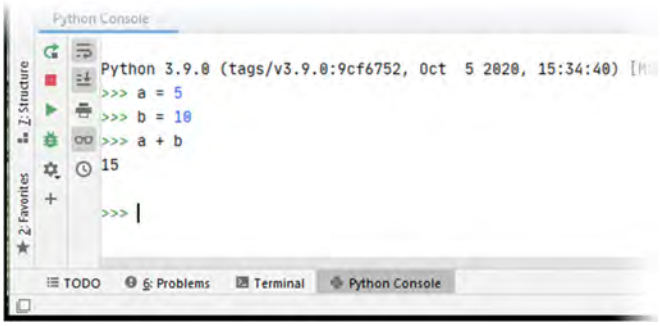


The current values of a and b are shown.

4. Perform arithmetic operations on your variables.

a) At the command prompt, enter `a + b`

b) Verify the value that Python returns.



The result of 15 is shown in the console.

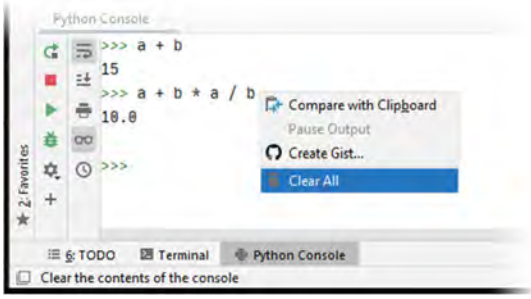
c) In the space provided, write your prediction of the value that should be returned by the statement `a + b * a / b`.

_____

d) Enter **`a + b * a / b`** and compare the result to the value you predicted.

5. Use the `print()` function to output messages to the console.

a) Right-click the last statement you wrote in the Python Console and select **Clear All**.



b) In the variable list, examine the values shown for a and b.

Even though you cleared the output, the values of a and b have been retained.

c) At the command prompt, enter **`print(a)`**

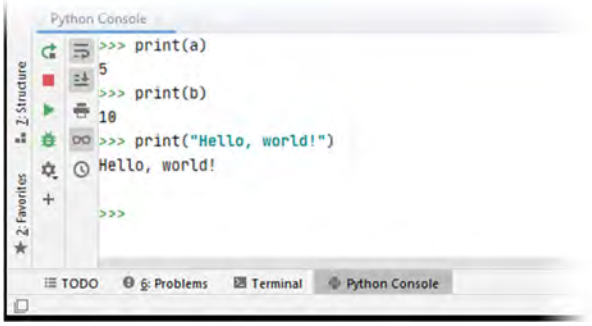
As you type the open parenthesis, the close parenthesis is automatically provided, so you don't have to type it.

d) Enter **`print(b)`**

e) Verify that Python prints **`5`** and **`10`** to the console.

f) Enter **`print("Hello, world!")`**

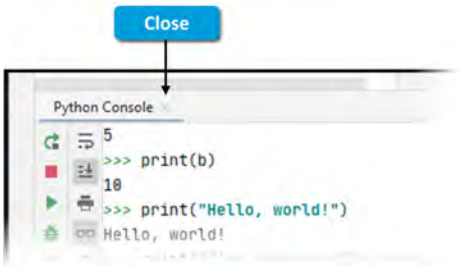
g) Verify that Python prints **`Hello, world!`** to the console.



h) Using escape codes, print to the console so that you receive the following output:

```
I've successfully used an "escape code."
```

6. Select the **Close** button on the **Python Console** tab.



TOPIC C

Create a Python Application

You're ready to move beyond simple statements and into the world of self-contained applications. In this topic, you'll write code that is common to nearly all Python applications.

Python Scripts and Files

Python's normal, non-interactive mode of operation is to save source code to a file. This file is also called a script and typically has the extension **.py**. The Python interpreter executes this script all at once and runs the program. In interactive mode, you typically stay inside the Python shell, but you can enter into a separate user interface by executing a script. This is how most developers create their Python applications.

Python source code can only be run if the Python interpreter is installed on the operating system. However, certain tools are available that allow you to convert your source code into OS-specific executables that don't require Python in order to run. For example, Windows does not come with Python; there are tools that can convert a **.py** to an **.exe**, allowing a Windows user without Python to run the application.

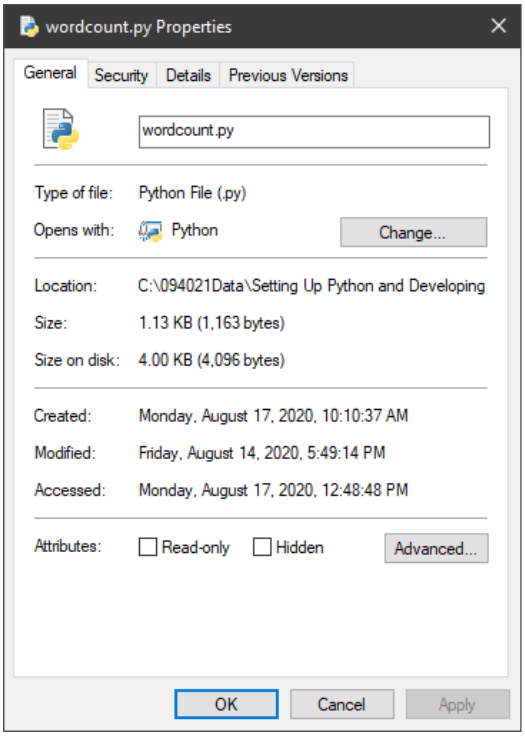


Figure 1-8: The file properties of a Python script.

Additional File Extensions

Python-related files may also come in the following extensions:

- .pyc (compiled source code)
- .pyo (optimized compiled source code)
- .pyd (dynamic module)
- .pyw (GUI-based source code)

Note: To learn how to create Windows executables from your Python scripts, check out the LearnTO **Make Your Python Program Usable on Any Windows Computer** presentation from the **LearnTO** tile on the CHOICE Course screen.

Comments

Comments are the programmer's way of annotating the source code that they are writing. The main purpose of comments is to make code easier to understand to human developers. The interpreter (or compiler) will ignore comments, and only someone with access to the source code will see them. There are several different ways programmers use comments:

- To clarify how or why a line of code does something.
- To clarify how or why a block of code does something or how it can be used elsewhere.
- To indicate any areas where code can be improved or expanded upon in the future.

Comments are, therefore, essential to every app, especially large, complex apps that will be handled by multiple developers. However, you should exercise careful judgment in using comments. Comments that state the obvious are a waste of space and do not add to the overall readability of code. For example, declaring a variable `eye_color = "blue"` is self-explanatory and would usually not warrant a comment.

In Python, you use the number sign (#) to declare that a line is going to contain comments for the interpreter to ignore. For example:

```
# User hasn't yet exited the window.
done = False
```

This is called a block comment. You can also place comments on lines that already contain interpretable code, called an inline comment. For example:

```
done = False  # User hasn't yet exited the window.
```

According to Python's style guide, block comments are preferable to inline comments.

Most IDEs will clearly differentiate the color of comment text from the color of normal code.

The Program Documentation String

Program documentation strings, or ***docstrings***, are similar to comments. From a technical standpoint, docstrings differ from comments in that they are not ignored by the interpreter. Instead, they are string literals that have actual value during runtime.

Functionally, docstrings are placed before large sections of code and reveal what that code will actually do. This means that docstrings are part of a program's documentation. Because they are formatted as string literals and exist at runtime, you can call upon this documentation. Docstrings are formatted with three single or double quotes at each end:

```
"""This program gets a user's input and returns a game board back to them."""
```

Docstrings can also span across multiple lines:

```
"""This program gets a user's input and returns a game board back to them.
It passes a user's input into a create_board() function that returns
a game board based on the user's difficulty selection."""
```

Note: To learn how to extract and format your program's docstrings, check out the LearnTO **Generate Documentation from Your Python Programs** presentation from the **LearnTO** tile on the CHOICE Course screen.

The input() Function

Most applications are designed to accept some type of user input. The `input()` function allows you to present the user with a command prompt and accept the value that they enter into that prompt. The following is an example of `input()` syntax:

```
name = input("What is your name?\n")
```

This code prints the question `What is your name?` and then opens up a command prompt for the user to type in. Whatever the user enters is passed into the variable `name`. By default, `name` will be a string. The following is what it looks like running from a script, with the user's name printed back to them.

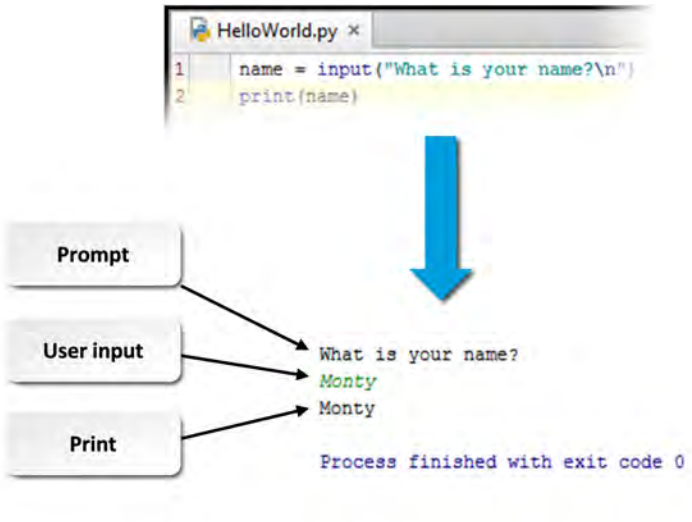


Figure 1-9: Accepting user input and printing it back to them.

Note: In Python 2, `raw_input()` is used instead of `input()`.

Multiple input() Functions

Multiple `input()` functions execute in sequential order as they appear in the source code. For example:

```
name = input("What is your name?\n")
quest = input("What is your quest?\n")
color = input("What is your favorite color?\n")
```

The program will first prompt the user for their name. After receiving the user's input, the program will then do the same for their quest and then their favorite color.

Command Line Arguments

You can execute a script directly, or you can run Python in interactive mode. You can also execute a script from your operating system's command line. Doing so allows you to provide arguments to your program. ***Command line arguments*** are a form of input that you specify before the script executes.

For example, say you want your program to take a file that the user specifies, make some calculations based on that file, then output the results to a new file. You would need two arguments: what file your program should take as input and what file it should produce as output. The user would have to specify both of these as command line arguments.

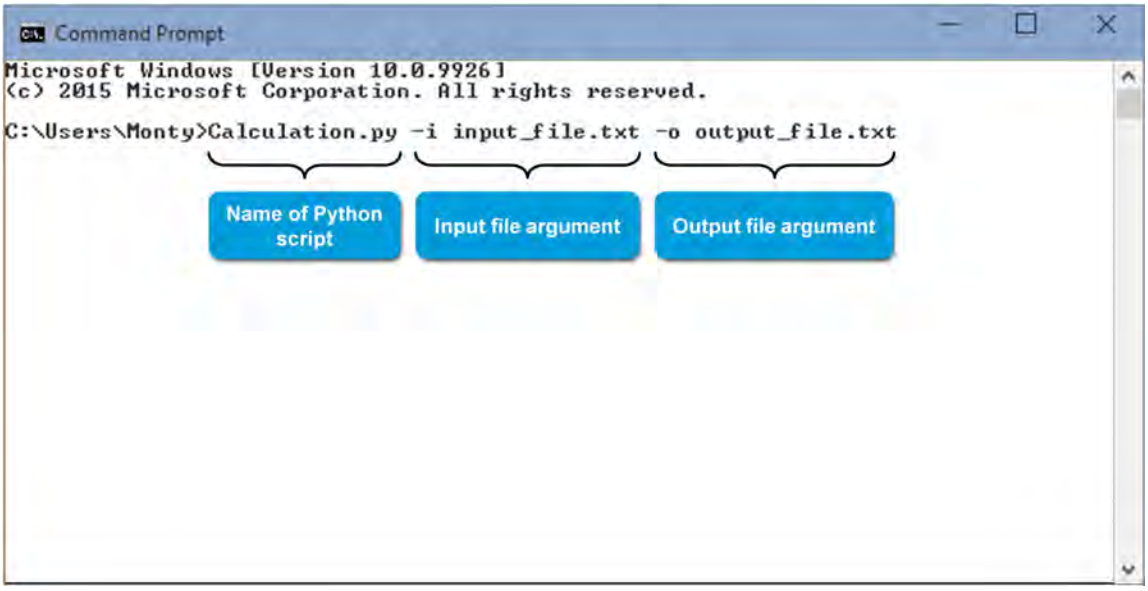


Figure 1-10: Running a Python script with two command line arguments: input file and output file.

The `-i` command indicates what type of argument the user is defining (in this case, input). `input_file.txt` is the name of an actual text file in this directory. The `-o` command indicates an output argument. `output_file.txt` is the name of the text file that the program will create after it performs its calculations.

The actual code you write to accept command line arguments will vary based on the nature of your program. A common way to accept command line arguments is by implementing the `argparse` module in your code.

	Note: Modules will be discussed in a later lesson.

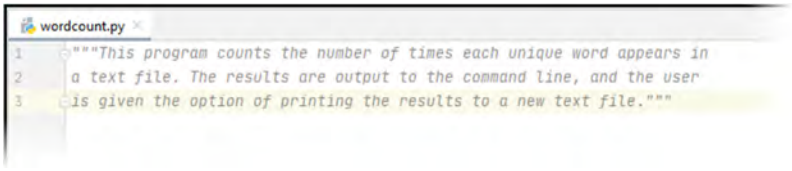
ACTIVITY 1-4
Creating a Python Application

Before You Begin
PyCharm is running, and an empty script **wordcount.py** script has been created.

Scenario
You've reviewed the requirements for the WordCount project, and have a good idea of the functionality it needs to provide. You've already created a Python project, including a file to contain the main script, where you'll start writing Python code to implement the WordCount program. You'll start by writing code to perform some simple input and output tasks.

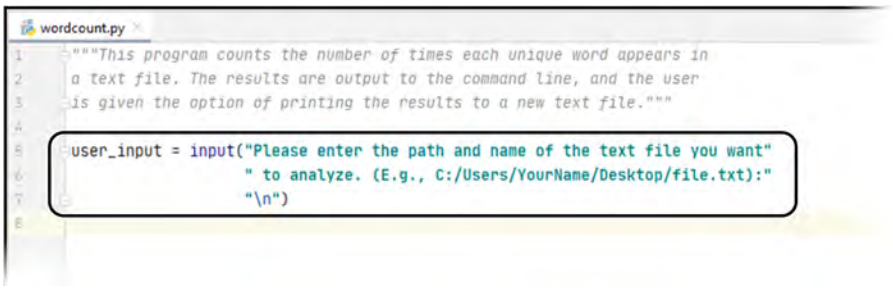
Note: You'll be developing this program all throughout the course.

1. Add a docstring that describes the program's function.
   a) In the editor, type the docstring as shown.



   - As you type the three opening double quotes, PyCharm automatically provides the closing double quotes.
   - You can press **Enter** to force a new line for formatting of your code, but the line break will not be considered to be part of the actual string.

2. Write code to take a user's input, then print it back to them.
   a) Press **Ctrl+End** to move to the end of the code, and press **Enter** until the insertion point is on line 5 as shown.
   b) Starting on line 5, add the following code.



   When the program runs, the user will type the name of the file to be analyzed. This statement assigns the variable `user_input` to whatever the user types in.

3. Add a print statement with a comment to explain the statement's purpose.

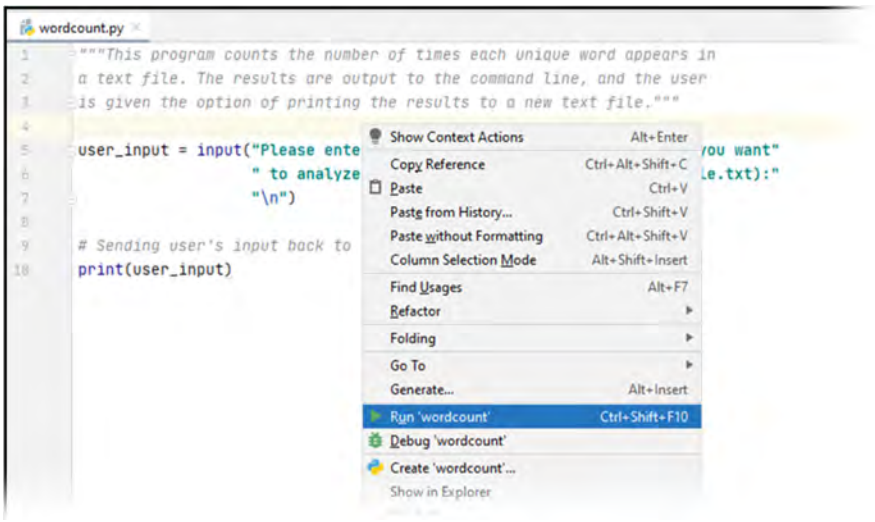
a) Press **Enter** to advance to line 9, and enter the two lines shown.

```
5   user_input = input("Please enter the path and name of the text file you want"
6                      " to analyze. (E.g., C:/Users/YourName/Desktop/file.txt):"
7                      "\n")
8
9   # Sending user's input back to them, for now.
10  print(user_input)
```

- The first line, a comment, explains what the second line does.
- The second line prints the user's input to the console.

4. Run your Python script.

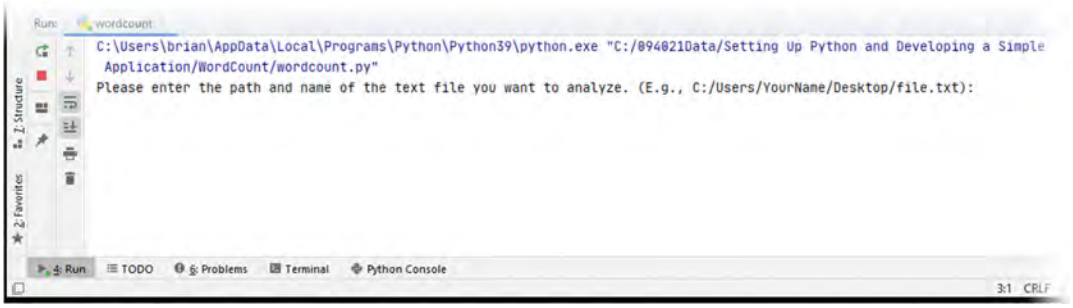
a) Within the wordcount.py code window, right-click and select **Run 'wordcount'**.



	Note: You don't need to manually save the source code yourself. PyCharm automatically does this after any change.

PyCharm opens the Run tool window at the bottom of the screen.

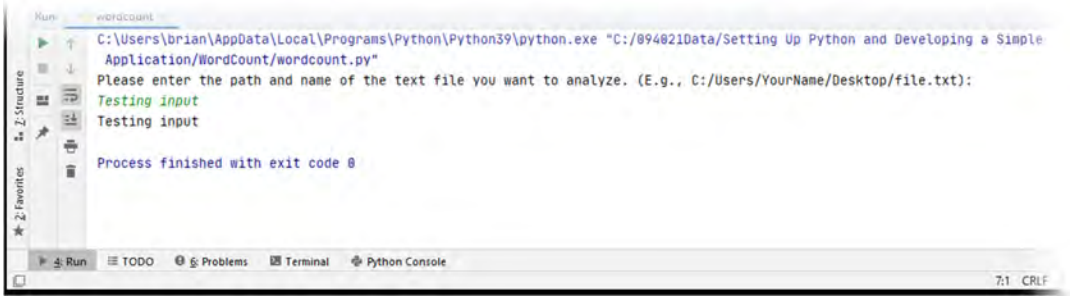
b) Examine the Run window.



Note: Many of the screen shots for this course were captured with PyCharm's **Soft-Wrap** setting enabled, to avoid truncating text. In this activity and others in this course, if text in the Run window is truncated on the right side of the screen, you might find it helpful to select the **Soft-Wrap** button to enable lines of output to wrap around automatically.

- In the Run window, you can see the command that PyCharm issued behind the scenes to run the Python interpreter (python.exe). The program file that you wanted to run (wordcount.py) was passed to Python.exe as a command-line parameter.
- Based on the source code you wrote, the Python interpreter created an executable program and ran it, and the running program has produced output in the Run window.
- The prompt you produced using the `input( )` function is displayed in the Run window.

c) Click in the Run window to select it for typing. Then type **`Testing input`** and press **Enter**.



d) Verify that Python returned your input back to you.

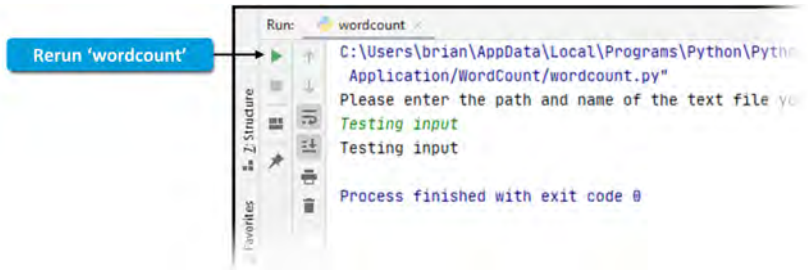
Note: **`Process finished with exit code 0`** means that Python was able to exit the program without any errors. Other exit codes like `1` or `-1` would indicate errors.

5. Create and run multiple, consecutive input statements.

a) In the editor pane, enter additional input and print statements in lines 12 through 18 as shown.

```
# Sending user's input back to them, for now.
print(user_input)

common_word = input("Would you like to strip common words from the results? (Y/N) ")

print(common_word)

user_output = input("\nWould you like to output these results to a file? (Y/N) ")

print(user_output)
```

b) In the Run window, select the **Rerun 'wordcount'** button.



The text in the Run window is cleared, and the new version of the program runs.

6. Test out the program.

a) At the first prompt, enter **`Testing input`**

```
C:\Users\brian\AppData\Local\Programs\Python\Python39\python.exe "C:/894021Data/Setting Up Python and Developing a Simple
 Application/WordCount/wordcount.py"
Please enter the path and name of the text file you want to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):
Testing input
Testing input
Would you like to strip common words from the results? (Y/N)
```

As before, the input is sent back to you. However, a new input prompt opens up directly after.

b) At the second prompt, enter **`Y`**

```
Would you like to strip common words from the results? (Y/N) Y
Y

Would you like to output these results to a file? (Y/N)
```

c) At the third prompt, enter **`N`**

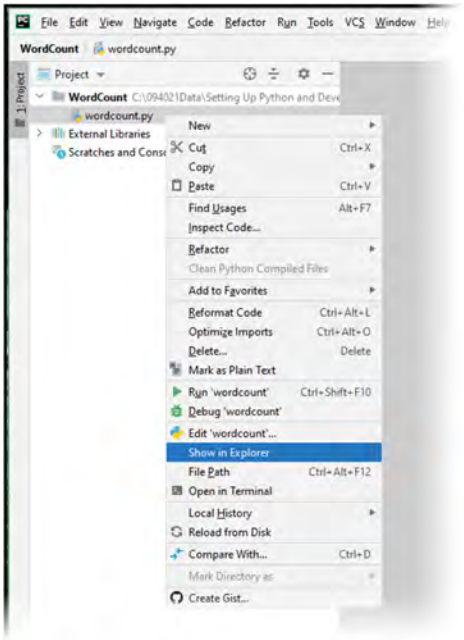
d) Verify that the program finishes with an exit code of zero.

e) In the Run window, close the **wordcount** tab.

f) Close the **wordcount.py** editor tab.
The editor for that file is closed, but the WordCount project remains open.

7. Examine the directory that contains the Python source code file.

a) In the **Project** pane, right-click **wordcount.py**, and select **Show in Explorer**.



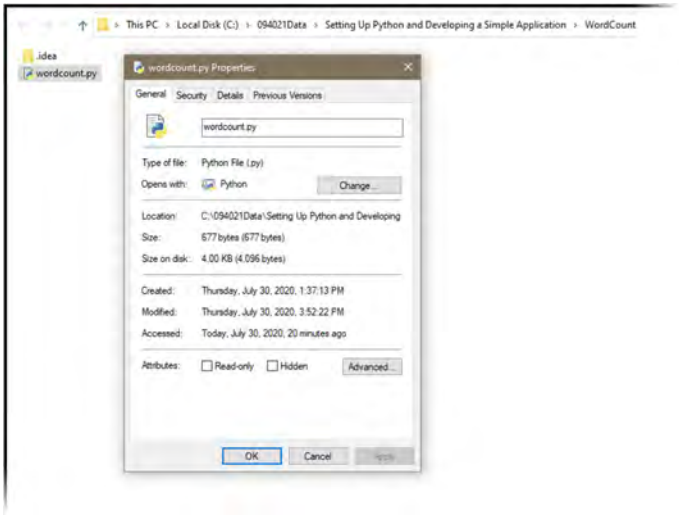
This command launches the Windows File Explorer program, and highlights the wordcount.py file within the project folder that you created.

b) Examine the contents of the project folder.

In addition to the Python file that contains your source code, the PyCharm editor automatically created a folder named .idea, which contains the project settings.

Note: Normally, you would not need to directly edit the files in the .idea folder, which are maintained automatically by the IDE.

c) In the File Explorer window, right-click **wordcount.py** and select **Properties**.



- This is the Python script you were just working on, saved as an individual file.
- It has the typical properties of a file, including the size, the creation date, and the default program it opens with.

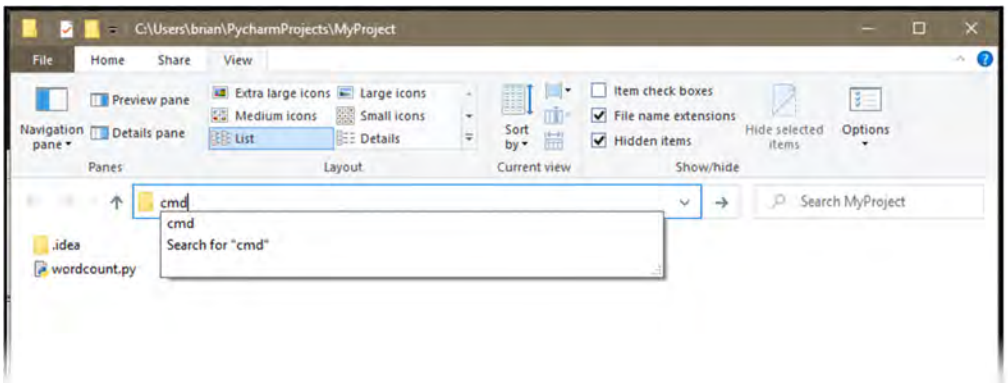
d) Close the **wordcount.py Properties** window.

8. Examine the project files from the command line.

a) In the File Explorer address bar, select the folder icon.

This selects the entire directory path.

b) Type ***cmd*** and press **Enter**.



This will run a Windows command prompt from the current directory location.

c) Examine the current directory in the command console.

d) Enter the **`dir`** command.

The wordcount.py file is shown in the list. If you were working outside of an editor like PyCharm, you might view the Python project files from the command line, like this.

9. Run the program directly from the command line.

a) Enter the **`python wordcount.py`** command.

This command uses the Python interpreter directly to run your script. The script runs, and the first prompt is displayed in the command console.

	**Note:** When Python was installed, the directory path for the Python installation was added to the computer's environment variables so you don't have to type the complete path every time you run a Python command.

b) At the various prompts, enter input as shown.

The program completes, and you are returned to the command line.

	**Note:** When running from the command line like this, the return code is not shown.

c) Enter the **`exit`** command to quit the command console.

d) Close the File Explorer window and return to the PyCharm IDE.

TOPIC D

Prevent Errors

Whether you're new to programming or have been coding for years, you're bound to make mistakes. In this topic, you'll learn about the different types of errors you'll encounter in Python. Then, you'll put good coding practices into use to stop errors from arising. As you develop your program throughout this course, you'll be better equipped to stop errors from becoming an obstacle to your progress.

Errors

In a program, an ***error*** refers to some sort of incorrect or unintended result after the execution or attempted execution of code. Errors are an unavoidable reality in programming, and Python is as susceptible to these pitfalls as any other language. In Python, errors are usually put into three categories:

- Syntax errors
- Logic errors
- Exceptions

Syntax Errors

A ***syntax error*** occurs when Python is unable to interpret some piece of code that you've written. These are relatively common, as human programmers often mistype or misspell statements, or they forget to add proper indentation and other formatting to their lines. Syntax errors almost always result in the program failing to execute and tend to be relatively obvious to spot and easy to correct. Thankfully, most modern IDEs will immediately anticipate syntax errors before you execute the program. Many will highlight the offending words or statements, usually in red, to signify that you need to change the code or else it will return an error.

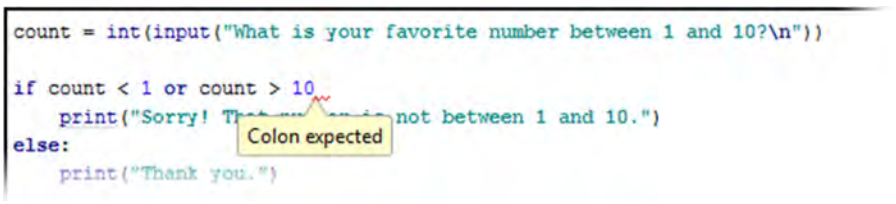


Figure 1-11: Syntax error (missing colon) identified in the IDE as the code is being written.

Logic Errors

A ***logic error*** is much more difficult to find and predict. These produce unintended results due to incorrectly implemented code. Even though the code is free of syntax errors and executes without fail, what the code actually does is wrong. Therefore, Python won't necessarily warn you that you have a problem that needs fixing. Take the following code as an example:

```
fav_number = input("What is your favorite number?")
fav_color = input("What is your favorite number?")

print(fav_number)
print(fav_color)
```

The code will execute without any syntax errors, but the `color` input is asking the wrong question. This will confuse the user and the program will fail to perform its intended function. Therefore, the

program is in error. These types of errors are why complex programs require extensive testing before they are finalized.

Guidelines for Preventing Errors

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following to help you prevent errors in your Python programs.

Prevent Syntax Errors

To prevent errors in syntax:

- Use an IDE with error indicators to detect syntax errors before execution.
- Use an IDE with code suggestion or completion to help you resolve syntax errors.
- Type carefully, and always remember to indent loops, conditional statements, function definitions, and other objects.
- Remember to add a colon at the end of the first line of most of these objects.
- Know Python's reserved words list so you don't accidentally try to use one as a variable.

Prevent Logic Errors

To prevent errors in logic:

- Use docstrings and comments to adequately explain what every block of code should do.
- Try to envision how a block of code could affect the whole program if it fails to execute as intended.
- Watch for loops that never initiate or terminate.
- Thoroughly test larger, more complex programs before releasing them to a wider audience.
- During testing, attempt to break your code; that is, test how the code handles unconventional or uncommon input.

ACTIVITY 1-5
Preventing Errors

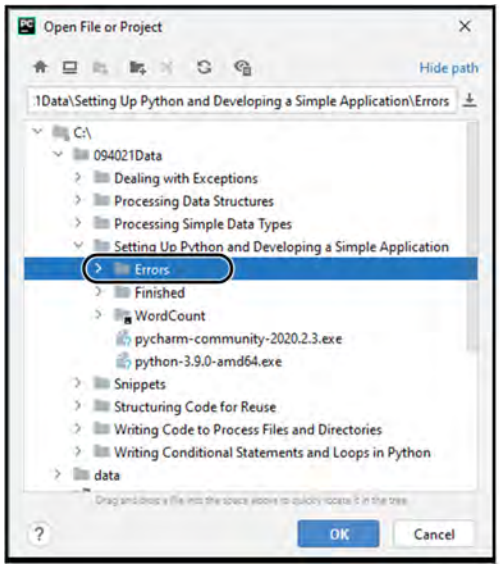
Data File
C:\094021Data\Setting Up Python and Developing a Simple Application\WordCount\wordcount.py

Scenario
Making mistakes is perfectly normal, and fortunately, IDEs like PyCharm are great for helping you spot and resolve these errors. In this activity, you'll open a slightly different version of wordcount.py that contains some errors. With PyCharm's help, you'll fix the errors and get the program running properly.

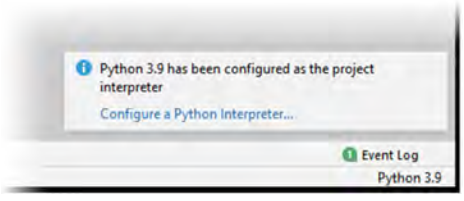
---

1. Open a version of the WordCount project that contains errors.
   a) In **PyCharm**, select **File→Close Project**.

      This closes the project you were working on, but leaves PyCharm running. You are returned to the Welcome to PyCharm window.
   b) Select **Open**, and beneath the **C:\094021Data\Setting Up Python and Developing a Simple Application** directory, select the **Errors** directory, as shown.



   c) Select **OK**.
   d) Confirm that the Python 3.9 interpreter will be used with this project.



e) In the **Project** pane, if **wordcount.py** is not showing beneath the **Errors** folder, select the triangle next to **Errors** to expand that folder.
f) Double-click **wordcount.py** to open the file for editing.

2. Attempt to run the program, and verify that Python produces a syntax error in the console.

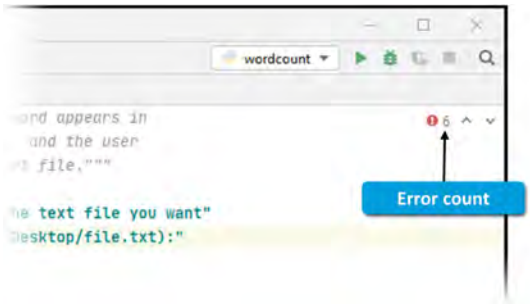
a) Within the **wordcount.py** code window, right-click and select **Run 'wordcount'**.

A syntax error is reported.

b) In the **Run** window, close the **wordcount** tab.

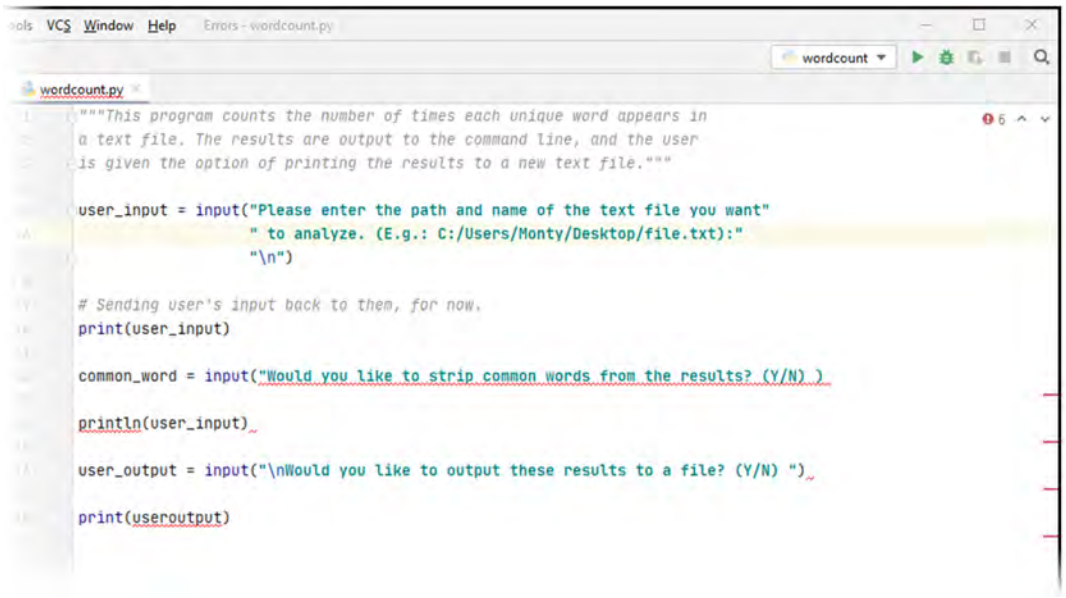
3. Debug the program.

a) Verify that PyCharm has detected several syntax errors in the code, denoted by red highlighting.
b) At the right of your source code, observe the error count.

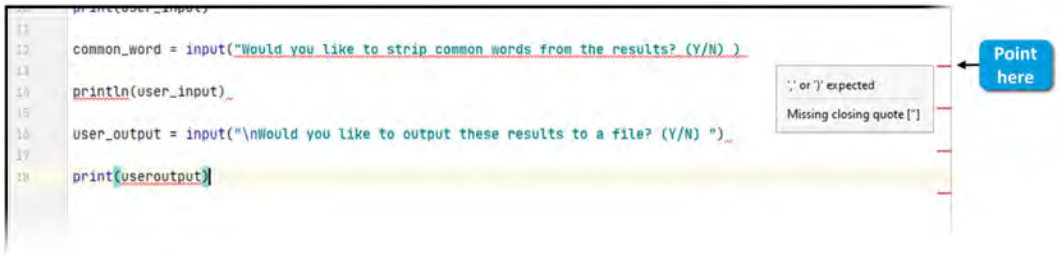


PyCharm has found numerous errors in the code.

c) Along the right edge, examine the markers showing lines that contain errors.



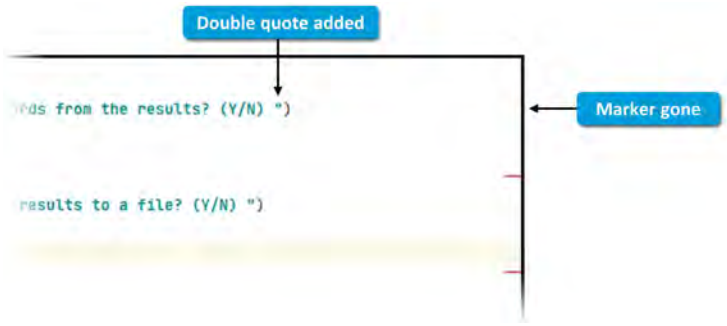
d) Point at the error marker at the end of line 12, and wait a moment.



Two possible errors have been identified in line 12.

4. Correct the missing closing quote.

a) Place your insertion point at the end of line 12, before the last closing parenthesis.
b) Type a double quote character.
c) Verify that line 12 is no longer marked as an error.



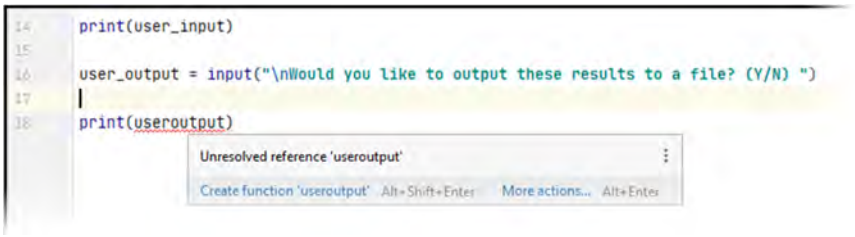
5. Fix the remaining syntax errors.

a) In line 14, point at the underlined word `println`.

Python's print statement is just `print`. There is no `println` statement in the language.

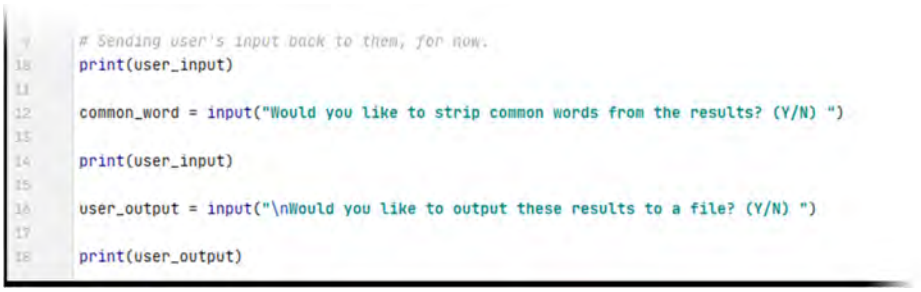
b) Fix the error by removing the letters `l` and `n` from `println`.

c) In line 18, point at the underlined word `useroutput`.



You didn't define `useroutput`, but you did define a `user_output` variable (notice the underscore). Mistakes like these are often the result of typos or forgetting the exact name of a variable. If the name isn't exact, Python won't be able to execute it.

d) Add the underscore to fix the print statement in line 18.

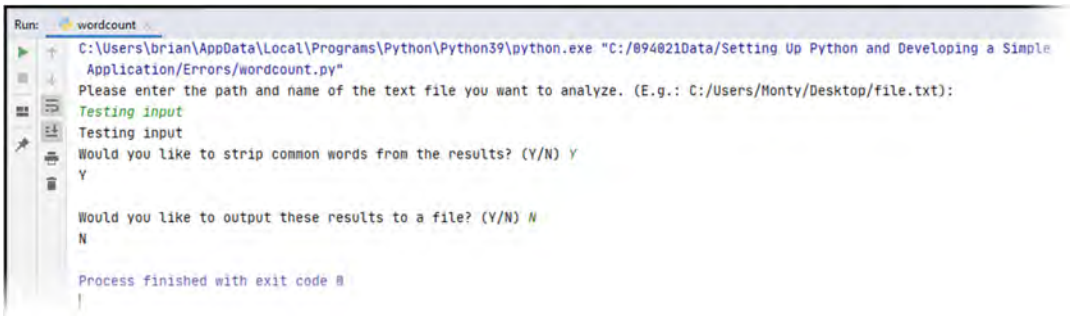


6. Test your program to make sure it runs properly.
   a) Run the program.
   b) Enter **`Testing input`** at the first prompt.
   c) When you're asked to strip common words, enter **`Y`**
   d) Observe that the resulting print line isn't repeating your **`Y`** answer, but says **`Testing input`**.
   e) In your source code, examine the print statement in line 14.

7. **Why does the program print the wrong value? How can you fix this statement to get the program to print the right value?**

8. **What kind of error is this? Why?**

9. Correct the error.
   a) In **wordcount.py**, correct the logic error in the code.
   b) Rerun the program and test that it works as intended.



10. Clean up the workspace.
   a) Select **File→Close Project**.
   b) Close the Welcome to PyCharm window.

Summary

In this lesson, you set up your Python environment to better prepare yourself for development. You then explored some fundamental variable assignments and math operations in the interactive shell. Finally, you used the code editor to save and run a Python script. These tasks are the foundation on which you will build your Python applications.

What about Python is different from the other programming languages you've used or have heard of?

When do you think you'll use interactive mode versus running a Python script normally?

	**Note:** To learn how to set up Python for the Windows command line, check out the LearnTO **Integrate Your Python Environment in Windows** presentation from the **LearnTO** tile on the CHOICE Course screen.
	**Note:** Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

2 Processing Simple Data Types

Lesson Time: 1 hour

Lesson Introduction

In any programming language, knowing how to work with data types is essential. Not all data types are the same, and being able to recognize what you can and can't do with each will help you make the most of your Python® code.

Lesson Objectives

In this lesson, you will:

- Process strings and integers.
- Process decimals, floats, and mixed number types.

TOPIC A

Process Strings and Integers

Both strings and integers are among the most common of simple data types. In this topic, you'll learn how to leverage the power of both in your Python® applications.

Data Types

Data types refer to how particular variables are classified and stored in memory. Classifying variables in different ways is useful because it allows programmers to set rules for what operations can be performed on those variables. For example, a variable that is of type integer can be used in arithmetic. A variable that is of type string can be used to display text to the screen. There are many uses for the different data types.

The following are the five most common data types in Python:

- Numbers
- Strings
- Sequences
- Dictionaries
- Sets

Unlike in many other programming languages, in Python, you do not need to explicitly define data types when you create variables. Python sets data types automatically based on the values you assign to variables. Python will interpret `a = 1` as an integer and `a = "1"` as a string.

Numbers

Numbers, as the name suggests, are variables with numeric values. When you explicitly define number variables in your code, you simply type the number to the right of the equals sign. No other symbol is required. The primary purpose of numbers is arithmetic. All of the arithmetic operators you learned previously can be performed on number variables to evaluate expressions and produce results.

Python categorizes numbers into additional subtypes. One of the most common subtypes is an ***integer***. Integers are either positive or negative whole numbers that do not contain a decimal point. All of the following variables are integers:

```
a = 56
b = -72
c = 0
d = 5893849
e = 2
f = -1
```

With some data types, like numbers, there are times when you'll need to convert one type or subtype to another. For example, say you have defined a string literal containing a series of digits, and want to convert it to an integer so you can perform arithmetic on it. To do this, you need to call Python's built-in `int()` function:

```
num_string = "635502"
new_num = num_string + 3           # Fails.
new_num = int(num_string) + 3      # Succeeds.
```

The `int()` function turns the string literal into an integer. Keep in mind that converting from one data type to another requires the value you're converting to be in the proper format. The code `int("Hello")` will not work, because alphabetical characters cannot be converted to integers.

Strings

A *string* is a data type that stores a sequence of characters in memory. A string value typically represents these characters and cannot be used as part of an arithmetic operation. Therefore, strings are distinct from numeric values like integers.

As you've seen before, string literals are variables with values enclosed in quotation marks. This is the way that strings typically appear in source code. While a programmer may write a string literal `a = "Hello"`, the Python interpreter will convert `a` into a string object at runtime. This happens behind the scenes, and for all intents and purposes, your use of strings will be confined to string literals.

Even though strings often represent alphabetical text, they can also represent other symbols and characters. For example:

```
a = "Count to 3."
```

Even though `3` is a number, here it is part of the string because it is enclosed within the quotation marks.

You can use the `str()` function to convert a value or variable into a string. For example:

```
num = 635502
print("The number is " + num)        # Fails.
print("The number is " + str(num))   # Succeeds.
```

The first print statement fails because a string can be added to a number.

The second print statement converts the number to a string before it is added to the string. Since the + operator in the second print statement is operating on two strings, the operation can be completed successfully. When you apply the + operator to two strings, they are concatenated. The second print statement would produce the following message.

```
The number is 635502
```

String Operators

Although strings are not subject to arithmetic operations, they do have their own special operators. These operators can manipulate strings in a number of ways. Assume the following is true for the proceeding table:

```
a = "Hello"
b = "World"
```

Operator	*Description*	*Example and Result*
`+`	Combines the left operand string with the right operand string. This process is called *concatenation*.	`a + b` `"HelloWorld"`
`*`	Concatenates a copy of the string itself the number of times defined by the right operand.	`b * 3` `"WorldWorldWorld"`
`[]`	Returns the character at the given position. This is called a *slice*.	`a[0]` `"H"`
`[:]`	Returns the group of characters that span the given positions. This is called a *range slice*. Note that the end position is exclusive; i.e., it is not included in the result.	`b[0:3]` `"Wor"`
`in`	Returns `True` if the given character(s) exist in the string. Returns `False` if they don't.	`"x" in a` `False`
`not in`	Returns `True` if the given character(s) do not exist in the string. Returns `False` if they do.	`"Hello" not in b` `True`

String Formatting

One additional way to process strings uses placeholder characters (`{}`). These are placeholders for variables or values that you define. The variables are defined at the end of the string, and the Python interpreter fills in the placeholders with their corresponding values at runtime. This process is called ***string interpolation***. String interpolation can make formatting complex strings easier.

When you use interpolation, you insert a placeholder within the string itself. Outside of your string literal, you'd type `.format()` and fill in the parentheses with whatever value or variable you want to inject into the string. For example:

```
>>> count = 2
>>> people = "There are {} people.".format(count)
>>> print(people)
There are 2 people.
```

Python uses `format()` to format the variable into a string, then passes this value into its placeholder field (signified by curly braces). You can also use multiple placeholders to interpolate multiple variables:

```
>>> count = 2
>>> name = "Terry"
>>> people = "There are {} people named {}.".format(count, name)
>>> print(people)
There are 2 people named Terry.
```

If you don't put anything inside these placeholder fields, Python will replace variables from left to right. Otherwise, you can manipulate the replacement values in several ways. For example:

```
people = "There are {1} people named {0}.".format(count, name)
```

Which outputs: `There are Terry people named 2.` The number you supply in the replacement field corresponds to the order of the variables in `format()`, starting with `0`.

Note: String interpolation is an alternative to concatenation and is especially useful since you'd otherwise need to convert any integer variables into strings before you can concatenate them.

Format Operator

Although using `{}` and `.format()` is the preferred method of string interpolation in Python 3, you can also interpolate with the `%` format operator:

```
>>> count = 2
>>> people = "There are %s people." % count
>>> print(people)
There are 2 people.
```

Notice the `s` character after the `%` character. This tells Python to format the placeholder as a string. You can also tell Python to format `count` as an integer with `%i`, which in this case will produce the same result. For multiple interpolated values:

```
>>> count = 2
>>> name = "Terry"
>>> people = "There are %s people named %s." % (count, name)
>>> print(people)
There are 2 people named Terry.
```

Guidelines for Processing Strings and Integers

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following to help you process strings and integers in your Python programs.

Process Strings and Integers

To help you process strings and integers in your Python programs:

- Don't worry about explicitly defining the data types of variables; Python does this for you.
- Use the conversion function `int()` to convert variables into the integer data type.
- Be sure that the values you're trying to convert to integers can actually be integers. Alphabetical characters cannot be converted to integers, for example.
- Convert variables to strings by using the `str()` function.
- Use the string operators `+` and `*` to concatenate strings.
- Use the string operators `[]` and `[ : ]` to find slices and range slices of a string, respectively.
- Use the `in` and `not in` operators to identify if a character or set of characters exists in a string.
- Use string interpolation (`{}`) to format complex strings that include variables, especially variables of non-string type.

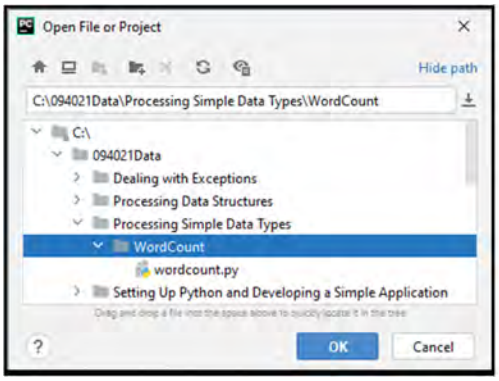
ACTIVITY 2-1
Processing Strings and Integers

Data File
C:\094021Data\Processing Simple Data Types\WordCount\wordcount.py

Scenario
Now that you've created the backbone for your user input processes, you'll want to start formatting that input. You'll eventually want to make sure that the input text file doesn't exceed a certain file size, as this could be too much for your system to handle or might simply take too long. Before you can make this rule strict, you'll ask the user to provide the file size themselves. You'll convert this size measurement (megabytes) into bytes, which will be of use later when you write code to reject files beyond a certain size.

---

1. Launch PyCharm and open the WordCount project.
   a) Launch **PyCharm** if you have not left it running from a previous session.
   b) In the Welcome to PyCharm window, select **Open**.
      The **Open File or Project** dialog box is shown.
   c) Beneath the **094021Data** folder, expand **Processing Simple Data Types**, and select the **WordCount** project folder.



   d) Select **OK**.
      The project is loaded and configured to use the Python 3.9 interpreter.
   e) In the **Project** pane, expand the **WordCount** folder if necessary, and double-click **wordcount.py** to show the source code in the editor.

2. Prompt the user to enter the size of the input file.
   a) Place the insertion point at the end of line 10 and press **Enter** twice to prepare to enter a new statement.

b) Write the code as shown to get the size of the input file from the user.

```
9   # Sending user's input back to them, for now.
10  print(user_input)
11
12  size_query = input("How big (in megabytes) is your input file? ")
13  size = int(size_query)
14  size_in_bytes = size * 1000000
15
16  common_word = input("Would you like to strip common words from the results? (Y/N) ")
```

- Line 12 creates a variable named **`size_query`** and assigns it to the input that the user enters in response to the prompt.
- Line 13 converts the user's string input to an integer so math operations can be performed on it.
- Line 14 multiplies the user's input to convert it to bytes (assuming one million bytes per megabyte).

3. Print the user's file size in bytes with string concatenation and interpolation.

a) Write the code as shown to print the file size in bytes.

```
12  size_query = input("How big (in megabytes) is your input file? ")
13  size = int(size_query)
14  size_in_bytes = size * 1000000
15  response = "Your file size of " + size_query + " megabyte is equal to {} bytes."
16  print(response.format(size_in_bytes))
17
18  common_word = input("Would you like to strip common words from the results? (Y/N) ")
```

- Line 15 creates a new `response` variable which contains a string literal concatenated with the value in `size_query`, concatenated with another string literal. A { } placeholder value is included in the string.
- Line 16 prints the `response` string, replacing the { } placeholder with the `size_in_bytes` value.

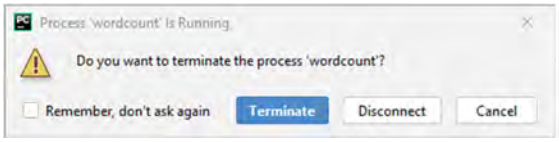
4. Why can you concatenate `size_query`, but must use interpolation for `size_in_bytes`?

5. Run the code to test your conversion operation and print statements.

a) Right-click the source code window and select **Run 'wordcount'**.
b) In the **Run wordcount** pane, enter any value at the first prompt.
c) When you're prompted to enter a file size in megabytes, enter **`1`**
d) Verify that your program converted your file size and formatted your string properly.

```
Run:  wordcount
C:\Users\brian\AppData\Local\Programs\Python\Python39\python.exe "C:/094021Data/Processing
Please enter the path and name of the text file you want to analyze. (E.g.: C:/Users/Monty
Testing input
Testing input
How big (in megabytes) is your input file? 1
Your file size of 1 megabyte is equal to 1000000 bytes.
Would you like to strip common words from the results? (Y/N)
```

e) In the Run window, close the **wordcount** tab.



f) Select **Terminate** to confirm you want to terminate the program.

TOPIC B

Process Decimals, Floats, and Mixed Number Types

Aside from integers, there are other number data types that you'll likely need to process in your applications.

Decimals

Besides integers, another type of number that Python can work with is decimals. Decimals are any non-whole numbers; that is, they contain a decimal point and a number following that decimal point. Decimals have a high degree of precision and can store numbers containing many digits. Because of this, decimals are most suitable in contexts where accuracy is absolutely vital; financial, accounting, and other fields that deal with monetary values which are best represented with decimals. An increase in precision means that decimals add processing overhead to a program.

The following is an example of creating a decimal in Python:

```
cost = decimal.Decimal("45.95")
```

In this case, `cost` holds the exact value of `45.95`.

Note: Actually using decimals requires that you import the decimal module. Importing a module is covered later in this course.

Note: To learn more about how to use decimal values, check out the LearnTO **Work with Decimals in Python** presentation from the **LearnTO** tile on the CHOICE Course screen.

Floats

As a tradeoff between performance and precision, ***floating point numbers*** or ***floats*** are used by programming languages like Python. Floats limit the precision of digits that trail the decimal point so less processing power is wasted in calculating numbers that don't really matter. For example, a carpenter measuring a piece of lumber doesn't need to know how thick in inches the wood is to the hundred thousandths decimal place (.00001). A float can help make this value less precise, but more practical to use.

Floats can use what is similar to scientific notation to represent a number. The number 123.456 could be represented as 1.23456×10^2. The decimal point in this notation can float to a different place, and its correct value can still be represented. For example, 1234.56×10^{-1} is the same value as before, but the decimal point has floated and the exponent has changed. This is advantageous because it allows programs to process numbers that have varying scales. If you want to multiply a very large number by a very small number, a float will maintain the accuracy of the result. For example, physicists may need to use astronomical values (e.g., the distance between stars) in the same operation as atomic values (for example, the distance between protons and electrons in an atom).

Defining floats in Python is very similar to defining integers. Simply by assigning a variable to a number with a decimal point, Python will define that variable as a float data type:

```
my_float = 123.456
```

Also, like integers, you can perform arithmetic operations on floats. In fact, in Python 3, dividing (/) any integers will result in a float, even if the remainder is zero:

```
>>> first_num = 6
>>> second_num = 2
```

```
>>> first_num / second_num
3.0
```

Numbers can also be converted to type float by using `float()`:

```
>>> my_integer = 5
>>> float(my_integer)
5.0
```

Note: Floats and decimals are different data types, despite their similarities.

Note: You should never use floats for monetary values, as these values require the high precision of a decimal.

Statements with Mixed Number Types

Python allows you to perform arithmetic on different types of numbers in the same operation. For example:

```
>>> my_integer = 8
>>> my_float = 4.0
>>> my_integer + my_float
```

See if you can guess the result of the operation before running it. When it comes to mixing integers with floats, the result of the arithmetic operation will always be a float. This helps preserve any fractional digits after the decimal point in a float. If you want to operate on integers and floats, but keep the result an integer, you can use the `int()` conversion object. Note that a simple conversion to an integer will always round down, no matter what the numbers are after the decimal. So, `int(12.7)` will become `12`.

When it comes to deciding which number type to conform to in mixed operations, Python uses the widest type. Floats are wider than integers, so in an operation with both, the integer must widen to the float.

Long Integers

In Python 2, there are two types of integers: integers and ***long integers***. Integers are precise out to 32 bits, whereas long integers have unlimited precision. The `long()` function converts numbers to the long integer type. In the sense of its width, a long integer is wider than an integer, but narrower than a float. Python 3 consolidated integers with long integers, so that `long()` is no longer a valid function.

String Formats for Float Precision

When you perform arithmetic on floats or mixed numbers, you'll often get a result with many digits after the decimal point. When it comes to outputting this in a string, too many numbers after the decimal point may not be ideal. You'll want the float to look neater—for example, you might not want a velocity value to display any more than two digits after the decimal point. So, you need a way to control the precision of your floats when you place them in strings.

The way to do this should be familiar, as it involves string interpolation:

```
>>> miles = 118.23
>>> hours_elapsed = 5
>>> mph = miles / hours_elapsed
>>> print("The air speed velocity is {:.2f} miles per hour.".format(mph))
The air speed velocity is 23.65 miles per hour.
```

The `:.2f` reference within the braces is a placeholder that alters the float's default formatting. It tells Python to truncate the float out to two decimal places. Then, `.format(mph)` passes the variable into

the placeholder. Using just `{}` would keep the float at `23.646`, which is one more digit than what you want in this case.

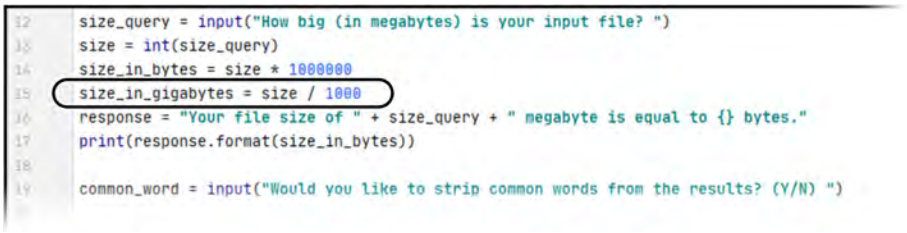
Note: You can also use the string format operator `%` for float precision in the same basic way.

ACTIVITY 2-2
Processing Mixed Number Types

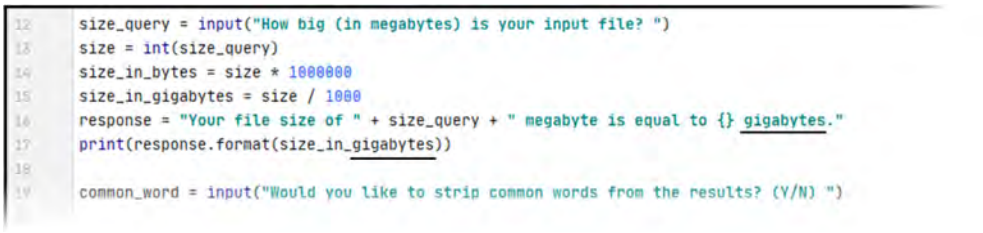
Scenario
Aside from converting megabytes to bytes, you'll also want to go the opposite direction and convert the user's file size to gigabytes. In doing so, you'll need to work with both floats and integers at the same time. You'll then exercise greater control over the precision of these floats by formatting them in strings you'll print as output.

---

1. Convert the user's input (megabytes) to gigabytes.
   a) Place your insertion point at the end of line 14 and press **Enter** to insert a new line.
   b) Write the code as shown to convert the size to gigabytes.



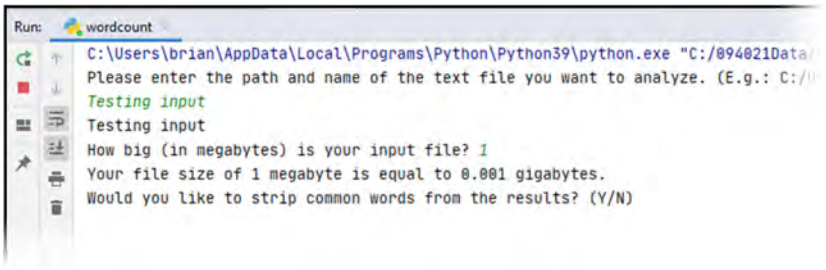
   Line 15 divides the user's input to convert it to gigabytes.
   c) Modify lines 16 and 17 as shown.



   The string text in line 16 and the variable reference in line 17 are changed. Now the program output will show a gigabyte value as calculated in line 15.

2. Run the program and provide input.
   a) Run the program and input any value on the path question.
   b) When you're asked about your file's size, enter `1`

c) Verify that you receive the following output:

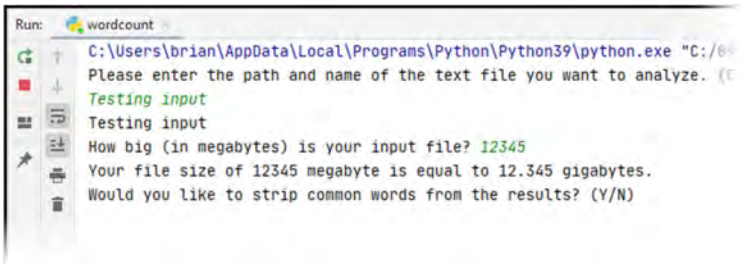


3. **Why did `size_in_gigabytes` turn into a float instead of an integer?**

4. **Assume you want to convert `size_in_gigabytes` back to megabytes by multiplying `size_in_gigabytes` by `1000`. What would the result of this operation be and why?**

5. Run your program with a more precise input value.

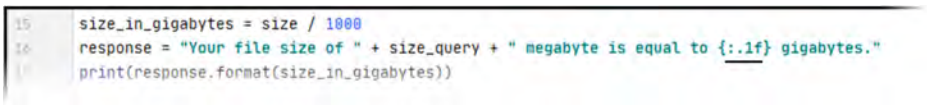
   a) In the **Run wordcount** pane, select the **Rerun 'wordcount'** button to run your program again. 
   This button appears if your program is still running.
   b) Input any value for the path and file name.
   c) When prompted for the file size, enter **`12345`**



   12.345 gigabytes is returned by the program.

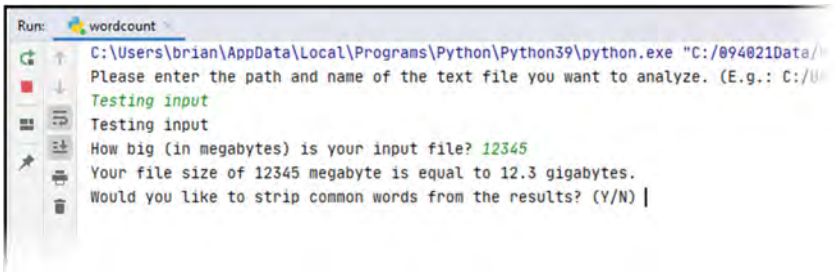
6. Format your output string to change your float's precision to a single decimal point.

   a) On line 16, within the string interpolation curly braces, type **`:.1f`**



   b) Rerun the program, and use **`12345`** as your file size input again.

c) Verify that your print line formatted the float to a single decimal point (`12.3`).



7. Clean up the workspace.
   a) Select **File→Close Project**.
   b) If you are prompted to terminate the running process, select **Terminate**.
   c) Close the Welcome to PyCharm window.

Summary

In this lesson, you processed simple data types like strings and numbers. Performing operations on these data types is essential to most programs.

What kinds of values might you want to format for float precision in your apps?

In what circumstances might you want to concatenate strings in your apps?

	**Note:** To learn how to manipulate date and time values, check out the LearnTO **Work with Date and Time in Python** presentation from the **LearnTO** tile on the CHOICE Course screen.
	**Note:** Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

3 Processing Data Structures

Lesson Time: 1 hour, 20 minutes

Lesson Introduction

You've processed simple data types, but Python® also supports more complex data structures. These data structures make almost any application, no matter how large and complex, much more powerful and easy to write.

Lesson Objectives

In this lesson, you will:

- Process ordered data structures.
- Process unordered data structures.

TOPIC A

Process Ordered Data Structures

In Python, data structures are often divided into two categories: ordered and unordered. You'll begin by processing ordered data structures, also called sequences.

Types of Sequences

In programming, a ***sequence*** variable is a collection of elements in which order matters. Because sequences are ordered, each element has its own index, or position, within the sequence. In Python, this index is numbered and starts with 0, increasing by one for each successive element.

There are three main sequence types in Python:

- Lists
- Ranges
- Tuples

Although they each involve an ordered collection of data, these sequence types store and process data differently.

Mutable vs. Immutable Objects

In programming, objects (including data structures) can be said to be either ***mutable*** or ***immutable***.

The values in mutable objects can be modified after that object has been defined. On the contrary, the values in immutable objects cannot be modified after those objects have been defined.

Although the difference may seem unimportant in practice, it can actually have some bearing on which data types you choose. Remember, a variable points to certain values in memory. When a mutable variable changes, it points to a different value in memory. Because you can't change the value of an immutable variable that's already been defined, you must create *another* variable in memory. Extrapolate this to programs that must constantly update many variables, and you'll see that using immutable variables in this situation will cost significantly more processing overhead.

Likewise, there's situations where using immutable objects is a better idea. For example, immutable objects can prevent conflicts in multi-threaded programs. If two or more threads run at the same time, but have different values for the same object due to some change, then this can corrupt the object. Even in single-threaded programs, having an immutable object is useful because it makes it easier to ensure that there won't be unwanted changes to your object somewhere in your code.

Although all objects in Python are either mutable or immutable, this property is most often associated with data structures.

The following data structures are **mutable**:

- Lists
- Dictionaries
- Sets

The following data structures are **immutable**:

- Ranges
- Tuples

List Type

A *list* in Python is a type of sequence that can hold different data types in one variable. Lists are mutable. Additionally, the values in a list do not need to be unique; they can repeat as many times as necessary. These qualities makes lists ideal for storing records of varying types and values, while still allowing you to update the lists when necessary.

Note: Lists are similar to arrays in other programming languages.

Let's say you want to work with a bunch of different year values in your program. You could define each year as its own separate integer, like this:

```
year1 = 1939
year2 = 1943
year3 = 1943
```

However, this can be very tedious and inefficient, especially with a large number of values. Lists are a better way of storing such values.

Like other variables, lists in Python are defined by assigning them to a variable using =. Python identifies a variable as a list when its value is enclosed in square brackets `[]` and each element inside those brackets is separated by commas. The following code defines a list of years:

```
years = [1939, 1943, 1943]
```

Notice that these are all integers. As stated above, you can mix any data type (even other lists) into a single list. The following list includes strings and integers:

```
years_and_names = [1939, 1943, 1943, "John", "Eric", "Michael"]
```

Lists have indices. Each element in a list, from left to right, is indexed, starting at `0`. In the preceding `years_and_names` list, index `0` is `1939`. You can retrieve and process specific indices in a list by appending square brackets to a variable and providing the index within those brackets. This is the same as "slicing" a string. For example, the following code retrieves `"John"`:

```
>>> years_and_names[3]
John
```

You may also use a range slice to retrieve indices:

```
>>> years_and_names[0:3]
[1939, 1943, 1943]
```

A range slice will give you all values between the indices you specify, the end position being exclusive. In the previous example, the range slice `[0:3]` will return indices `0`, `1`, and `2`. Being able to retrieve indices in a list is useful for when you need to process specific elements in that list.

Using indices, you can also update or overwrite elements in a list. The following code replaces the value at index `3` (`"John"`) with the string `"Terry"`:

```
years_and_names[3] = "Terry"
```

List Processing

Like other data types, you can perform a number of operations on lists. The following table provides examples of list operations using the lists `["A", "B", "C"]` and `["D", "E", "F"]`.

Expression and Result	*Description*
`["A", "B", "C"] + ["D", "E", "F"]` `["A","B","C","D","E","F"]`	Concatenates lists.
`["A", "B", "C"] * 3` `["A", "B", "C", "A", "B", "C", "A", "B", "C"]`	Repeats list values.

Expression and Result	Description
`"A" in ["A", "B", "C"]` `True`	Returns `True` if value exists in the list; `False` if it doesn't.

Other than finding indices and performing operations, you can manipulate lists in several other ways. The following table assumes a list named `my_list` that is equal to `[20, 10, 30, 10]`.

Approach	Example	Result	Description
`.append()`	`my_list.append(5)`	`[20, 10, 30, 10, 5]`	Adds values you specify to the end of a list.
`.insert()`	`my_list.insert(0, 5)`	`[5, 20, 10, 30, 10]`	At an index of a list you specify (first argument), inserts a value you specify (second argument).
`.remove()`	`my_list.remove(10)`	`[20, 30, 10]`	Removes first item in the list that matches the value you specify.
`.index()`	`my_list.index(30)`	`2`	Returns the index in a list of the value you specify.
`.count()`	`my_list.count(10)`	`2`	Returns the number of times a value you specify appears in a list.
`.sort()`	`my_list.sort()`	`[10, 10, 20, 30]`	Sorts items in a list in increasing order.
`.reverse()`	`my_list.reverse()`	`[10, 30, 10, 20]`	Reverses the order of values in a list.
`.clear()`	`my_list.clear()`	`[]`	Removes all items from a list.
`del`	`del my_list[2]`	`[20, 10, 10]`	Deletes an item from a list at the specified index. Values to the right of the deleted item are shifted to the left.
`len()`	`len(my_list)`	`4`	Retrieves the number of items in a list.
`max()`	`max(my_list)`	`30`	Identifies the highest value in a list. Returns an error if the list contains unsortable data, like another list or a mix of strings and numbers.
`min()`	`min(my_list)`	`10`	Identifies the lowest value in a list. Returns an error if the list contains unsortable data, like another list or a mix of strings and numbers.

.split()

The `.split()` method does not technically process lists, but actually creates a list from a string. By default, it will split a string by placing each character separated by a space into its own list index. For example:

```
>>> test = "Split this string"
>>> test.split()
["Split", "this", "string"]
```

In this example, each word is at its own index because each word is separated by a space.

Ranges

In Python, a ***range*** is an immutable sequence of integers, meaning its values cannot change. It is typically used as a way to loop or iterate through a process a number of times. This is much more efficient and allows you greater control than if you simply repeated the same code snippet over and over again.

The syntax of a range is `range(start, stop, step)`. The `start` argument tells the range where in the list to begin, and if left blank, will default to position `0`. The `stop` argument tells the range where to end and must be specified. The `step` argument defaults to `1` and will process the very next value in the range of a list, unless you specify a different step argument.

Consider the following code:

```
my_range = range(0, 50, 5)
```

This creates a range that starts at `0`, increments by `5`, and ends at `50`. Therefore, the range contains the following integers: `0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50`. Now consider this code snippet using the same `my_range`:

```
>>> my_range[3]
15
```

Like normal lists, you can find the index of a value in a range using square brackets. In this case, index `3` will return `15`.

Note: Printing an entire range won't format it like a list. For example, printing `my_range` will return `range(0, 50, 5)`. Nevertheless, the variable still holds all of the integers of the range.

Tuple Type

A ***tuple*** is very similar to a list, the main difference being that tuples are immutable. As stated before, immutability has its benefits, and tuples may help you ensure that your data structure does not change somewhere within your code. Tuples also process faster than lists, which can make a difference in larger programs.

Tuple syntax uses parentheses `()` to enclose its values. For example:

```
my_tuple = (1939, 1943, 1943, "John", "Eric", "Michael")
```

As you can see, tuples, like lists, can hold different data types. This can include other tuples.

As with other sequence types, you find an index in a tuple by using square brackets:

```
>>> my_tuple[0]
1939
```

And with a range slice:

```
>>> my_tuple[2:4]
(1943, "John")
```

Tuple Processing

Tuple processing is very similar to list processing. The main exceptions are that you cannot add, update, or delete values from a tuple, as this is not possible with an immutable data type. The following table provides examples of tuple operations using the tuple `("A", "B", "C")`.

Expression	Result	Description
`("A", "B", "C") + ("D", "E", "F")`	`("A","B","C","D","E","F")`	Concatenates tuples.
`("A", "B", "C") * 3`	`("A", "B", "C", "A", "B", "C", "A", "B", "C")`	Repeats values.
`"A" in ("A", "B", "C")`	`True`	Returns `True` if value exists; `False` if it doesn't.

The following table assumes a tuple named `my_tuple` that is equal to `(20, 10, 30, 10)`.

Approach	Example	Result	Description
`.index()`	`my_tuple.index(30)`	2	Returns the index in a tuple of the value you specify.
`.count()`	`my_tuple.count(10)`	2	Returns the number of times a value you specify appears in a tuple.
`len()`	`len(my_tuple)`	4	Retrieves the number of items in a tuple.
`max()`	`max(my_tuple)`	30	Identifies the highest value in a tuple. Returns an error if the tuple contains unsortable data, like another tuple or a mix of strings and numbers.
`min()`	`min(my_tuple)`	10	Identifies the lowest value in a tuple. Returns an error if the tuple contains unsortable data, like another tuple or a mix of strings and numbers.

Note: You can convert lists to tuples using the `tuple()` function, and vice-versa using `list()`. This can be useful if you need to change the structure's mutability on-the-fly.

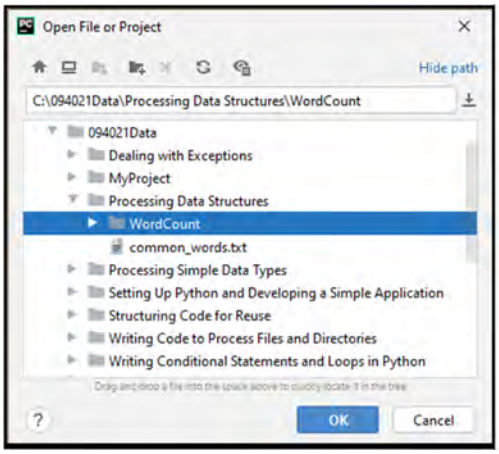
ACTIVITY 3-1
Processing Ordered Data Structures

Data File
C:\094021Data\Processing Data Structures\WordCount\wordcount.py

Scenario
You've decided to address file size later. For now, you'll move on to storing values in more complex types than numbers and strings. Particularly, you want to store the final results of the word count in some sort of structure. You need your results to be ordered by the number of times each word appears, and you also need the results to be mutable so you can continually add to the structure. This calls for a list. In this activity, you'll take input from the user and add the values in that input to a list. You'll take a file as input later, but for now, you'll ask the user to manually input words to the console. The program's specifications explain that some common words need to be suppressed in the final output, so you'll test out removing values from the list as well.

---

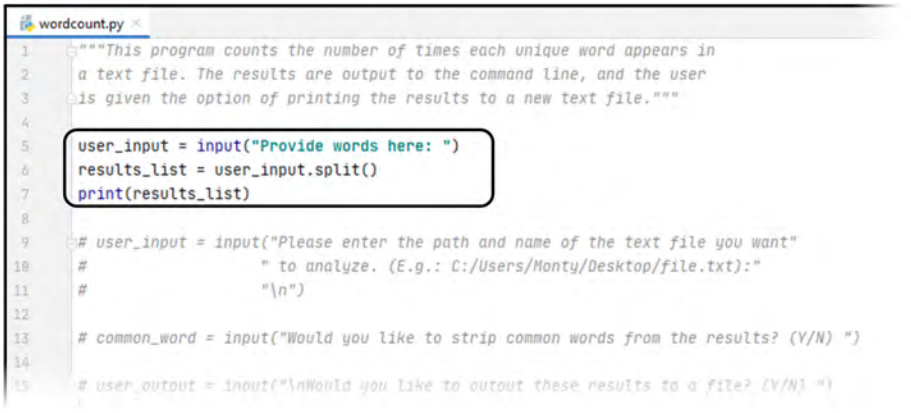
1. Launch PyCharm and open the WordCount project.
   a) Launch **PyCharm** if you have not left it running from a previous session.
   b) In the Welcome to PyCharm window, select **Open**.
      The **Open File or Project** dialog box is shown.
   c) Beneath the **094021Data** folder, expand **Processing Data Structures**, and select the **WordCount** project folder.



   d) Select **OK**.
      The project is loaded and configured to use the Python 3.9 interpreter.
   e) In the **Project** pane, expand the **WordCount** folder if necessary, and double-click **wordcount.py** to show the source code in the editor.
      Lines of code associated with getting the input file size have been removed, and lines with various input prompts have been commented out. The interpreter will essentially ignore these lines now, but it will be easy to enable them again later by removing the hash marks.

2. Write code to get input from the user and print the individual words the user entered.
   a) Position the insertion point in line 5.

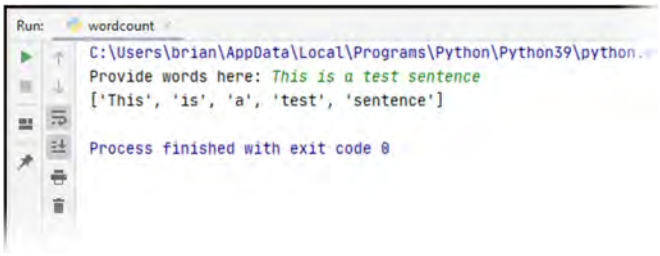
b) Write the code as shown.



- Line 5 stores input from the user in a variable named user_input.
- Line 6 uses the `split()` statement to split the input text wherever any spaces appear. Each item in the resulting list will be a single word.
- Line 7 prints the list.

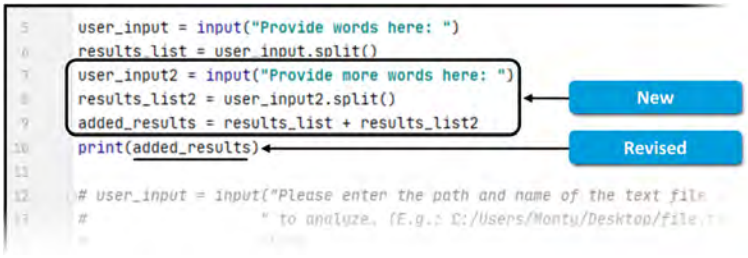
3. Run the program and test some input.

a) Run the program.
b) In the console, when prompted to provide words, type ***This is a test sentence*** and press **Enter**.
c) Verify that Python prints a list and that each item in the list is a string of a single word.



4. Prompt the user for more input to add to the existing list.

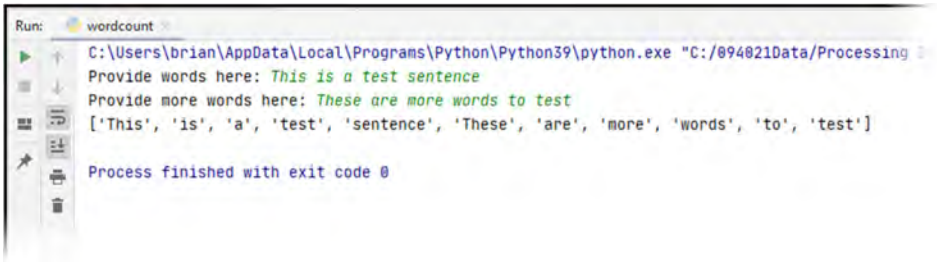
a) Position the insertion point at the end of line 6 and press **Enter**.
b) Write three new lines of code as shown, and revise the print statement to output added_results.



- Line 7 prompts the user to provide more words. Assign the input to the variable **`user_input2`**
- Line 8 uses the `split( )` function to create another list of words.
- Line 9 uses the + operator to combine the two lists.
- Line 10 now prints out the combined lists.

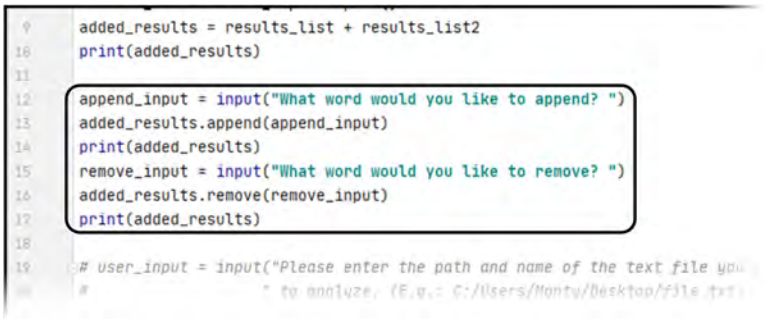
5. Test the list addition.

a) Run the program.
b) Enter ***This is a test sentence*** for the first input and ***These are more words to test*** for the second input.
c) Verify that your output list (`added_results`) combines the items from both lists.



6. Append and remove individual items to and from the list.

a) Position the insertion point at the end of line 10 and press **Enter** twice so that your insertion point is on a new blank line 12.
b) Write six new lines of code as shown.

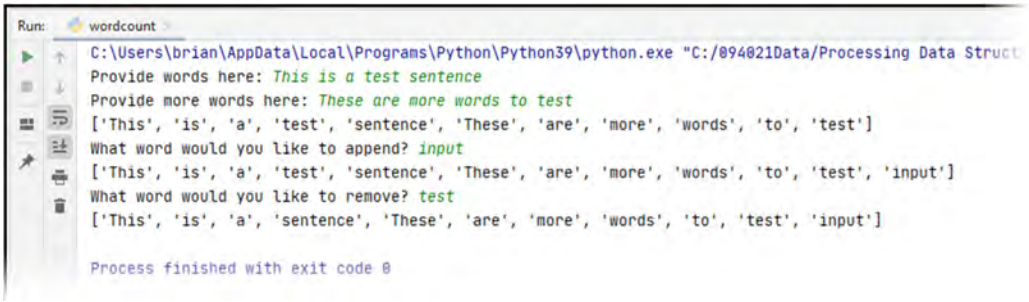


- Line 12 prompts the user for a word to append, storing the word in `append_input`.
- Line 13 appends the word in `append_input` to the `added_results` list.
- Line 14 prints the list up to this point.
- Line 15 prompts the user for a word to remove, storing the word in `remove_input`.
- Line 16 removes the word in `remove_input` from the `added_results` list.
- Line 17 prints the completed list.

7. Test out your append and remove operations.

a) Run the program.
b) Enter ***This is a test sentence*** for the first input and ***These are more words to test*** for the second input.
c) When asked to append a word, enter ***input***
d) Verify that your word was added to the list in the printed statement.
e) At the next prompt, type in ***test*** to provide a word that currently exists in the list.

f) Verify that the first instance of the word **test** was removed from the list.



g) In the Run window, close the **wordcount** tab.

8. **The word "test" still exists in the list. Why has it not been removed?**

TOPIC B

Process Unordered Data Structures

In addition to sequences, Python also has a couple of unordered data structure types. You'll process dictionaries and sets in this topic.

Dictionary Type

A ***dictionary*** is an unordered, mutable data structure. This means that, unlike lists, each element in a dictionary is not bound to a specific numbered index. What further separates dictionaries from other data types is that dictionaries have key-value pairs. The key acts as an index; calling that key returns its associated value. This makes dictionaries ideal for when you need your program to look up information using other information it's associated with. For example, in a phone registry, you'd map a person's name (the key) to their phone number (the value). As the name implies, you can also think of a dictionary as containing a term matched with the definition of that term.

Note: In other languages, a dictionary may be called an associative array or a hash table.

Dictionaries are defined by using curly braces (`{}`). Within the braces, a key is separated from its value by a colon (`:`), and any subsequent key-value pairs are separated by a comma. Observe the following code, in which a dictionary is defined:

```
dict = {"John": 5551234, "Terry": 5554321, "Eric": 5551234}
```

In this dictionary, there are three keys and three associated values. Each key must be unique, otherwise it cannot act as an index. However, the values for each key do not have to be unique (John and Eric have the same number).

Because dictionaries are indexed by key, you can access each value in the dictionary. The syntax for accessing a key's value is similar to accessing indices in a list:

```
>>> dict["Terry"]
5554321
```

Keep in mind that because values are not unique, you won't be able to do the reverse (look up a key from a value).

Like in lists, you can put any data type you want in a dictionary value. You can even nest dictionaries within a dictionary. However, the keys in a dictionary must be immutable. For example, a list cannot be a key.

Note: A space is not required between key and value, but including spaces increases readability and is the preferred practice according to Python's style guide.

Dictionary Processing

You can add, update, and delete key-value pairs in a dictionary easily. Consider the previous dictionary of names and phone numbers:

```
dict = {"John": 5551234, "Terry": 5554321, "Eric": 5551234}
```

To add Graham and his phone number to the dictionary, you'd type the following:

```
dict["Graham"] = 5556789
```

To change Terry's phone number to `5556789`, you'd type:

```
dict["Terry"] = 5556789
```

Note that this is essentially the same as adding a new entry, but Python detects that the key `"Terry"` already exists, so it simply updates the dictionary. To delete John and his phone number from the dictionary:

```
del dict["John"]
```

By simply calling the dictionary variable, you can see how it's changed. Remember that dictionaries are unordered, so the result will likely not be in the order you defined the key-value pairs:

```
>>> dict
{'Eric': 5551234, 'Graham': 5556789, 'Terry': 5556789}
```

There are many more ways you can process dictionaries. Assume that the following table uses the updated dictionary of names and phone numbers.

Approach	*Example*	*Result*	*Description*
`len()`	`len(dict)`	`3`	Returns the total number of key-value pairs in the dictionary.
`.items()`	`dict.items()`	`dict_items([("Eric", 5551234), ("Graham", 5556789), ("Terry", 5556789)])`	Returns each key-value pair in the dictionary.
`.keys()`	`dict.keys()`	`dict_keys(["Eric", "Graham", "Terry"])`	Returns the keys in a dictionary.
`.values()`	`dict.values()`	`dict_values([5551234, 5556789, 5556789])`	Returns the values in a dictionary.
`in`	`"John" in dict`	`False`	Returns `True` if specified value is a key in the dictionary, `False` if not.
`.clear()`	`dict.clear()`	`{}`	Removes all entries from a dictionary.

New in Python 3.9

Python 3.9 introduced additional ways to work with dictionaries.

The merge operator (|) enables you to easily combine dictionaries.

```
tv = {'Flying Circus':1969, 'Live at Aspen':1998}
film = {'Holy Grail': 1975, 'Life of Brian': 1979}
projects = tv | film
print(projects)
```

This output from this code would be `{'Flying Circus': 1969, 'Live at Aspen': 1998, 'Holy Grail': 1975, 'Life of Brian': 1979}`

The update operator (|=) updates the original dictionary.

```
projects = {'Flying Circus':1969, 'Live at Aspen':1998}
projects |= {'Holy Grail': 1975, 'Life of Brian': 1979}
print(projects)
```

This output from this code would be `{'Flying Circus': 1969, 'Live at Aspen': 1998, 'Holy Grail': 1975, 'Life of Brian': 1979}`

Note: If the dictionaries contain a common key, the key-value pair in the second dictionary will be used—essentially updating the value in the original dictionary.

Set Type

A ***set*** is an unordered, mutable data structure that cannot contain duplicate values. Unlike a dictionary, it does not have key-value pairs; there is only one value per entry, similar to a list. Sets can only contain immutable data types, like integers, floats, and strings. They cannot contain mutable data types, like lists, dictionaries, and other sets. The reason for using a set over a list is that processing large sets is considerably faster, which is useful when you need to continually update or confirm values in the set. For example, assume that you have a large database of files. Each file name in the database is unique, and its order does not matter. You want to check whether or not a certain file still exists to make sure it hasn't been deleted. Looking up this particular file will take less time if all of the file names are part of a set.

The syntax for defining a set is by using curly braces, like a dictionary:

```
my_set = {"Blue", "Yellow", "Arthur", "Robin", 24}
```

Since there is no key-value pair in a set, you'd only be separating values by commas. This set contains string and integer data types and, as required, each entry is unique. If you attempt to define a set with repeating values, only one of those values will be in the set when Python creates it.

You can also explicitly use `set()` when defining your variable. Within the `set()` arguments, you enclose the values in square brackets, like you would in a list:

```
my_set = set(["Blue", "Yellow", "Arthur", "Robin", 24])
```

Set Processing

Set processing has much in common with list processing. However, the unique values in a set allow you to do additional processing not possible in lists. For the following table, assume `my_set` is equal to `{"X", "Y", "Z"}` and `my_set2` is equal to `{"X", "Y", "A"}`.

Approach	*Example and Result*	*Description*
`in`	**`"X" in my_set`** `True`	Returns `True` if specified value is in the set, `False` if not.
`.add()`	**`my_set.add(1)`** `{"X", "Y", "Z", 1}`	Adds the specified value to the set.
`.remove()`	**`my_set.remove("X")`** `{"Y", "Z"}`	Removes the specified value from the set.
`.clear()`	**`my_set.clear()`** `set()`	Removes all values from the set.
`.difference()`	**`my_set.difference(my_set2)`** `{"Z"}`	Compares one set with another and returns the value(s) that are in the first set, but not also in the second set.
`.intersection()`	**`my_set.intersection(my_set2)`** `{"X", "Y"}`	Compares one set with another and returns the value(s) that appear in both sets.
`.union()`	**`my_set.union(my_set2)`** `{"X", "Y", "Z", "A"}`	Combines all of the unique values from two sets into one set.

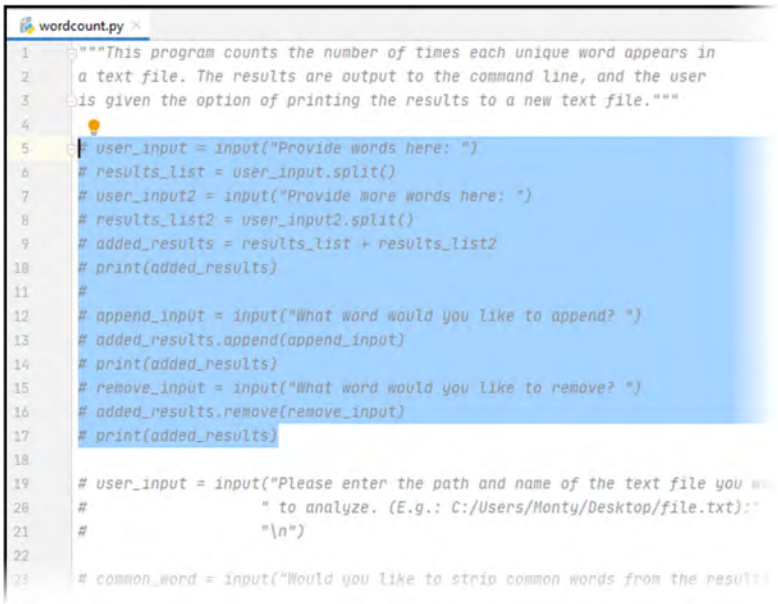
ACTIVITY 3-2
Processing Unordered Data Structures

Scenario
When it comes to actually processing how many times each word appears in a text file, you aren't necessarily concerned about the order of items. You do, however, need the structure to change as you continually add words and numbers to it. So, you'll use a dictionary for this purpose. In this dictionary, each word will be the key and each key's value will be the amount of times that key appears. Aside from populating the dictionary with your test data, you'll also manipulate the dictionary to both alter values and check to see if certain words exist.

The other unordered data structure you'll add to your program is a set. You learned before that the program will need the ability to suppress common words. These common words were provided by the Editing department and consist of the most common words used in English language writing, not including pronouns as they may be valuable for analyzing perspective. So, you'll define these words as a set constant to work with later.

---

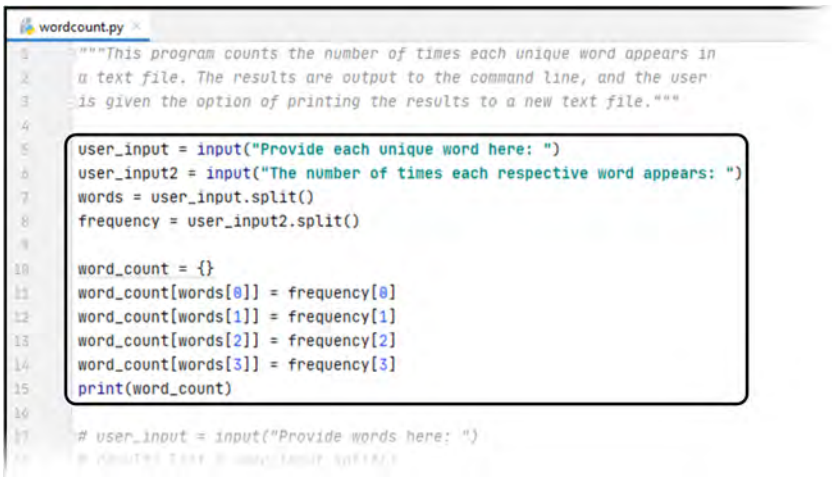
1. Comment out code for later use.
   a) Highlight lines 5 through 17, which hold your currently active code.
   b) From the menu, select **Code→Comment with Line Comment**.
   c) Confirm that everything below the docstring is now commented out.



	Note: You'll return to this code later. For now, it'll be easier to test your new code by itself.

2. Take new user input for each word and its frequency.
   a) Position the insertion point at the end of line 3 and press **Enter** twice.

b) Starting in line 5, enter the code as shown.



- Line 5 prompts the user to enter a list of unique words, to be stored in `user_input`.
- Line 6 prompts the user to enter a corresponding list of word counts, to be stored in `user_input2`.
- Lines 7 and 8 split the user inputs into list variables stored as `words` and `frequency`.
- Lines 10 through 14 create a dictionary from each input, using the values in `words` as keys, and the values in `frequency` as the word count values.

	Note: You'll write a more efficient way of adding items to the dictionary later. For now, you'll populate the dictionary manually.

3. Verify that the dictionary is being populated with your test input.

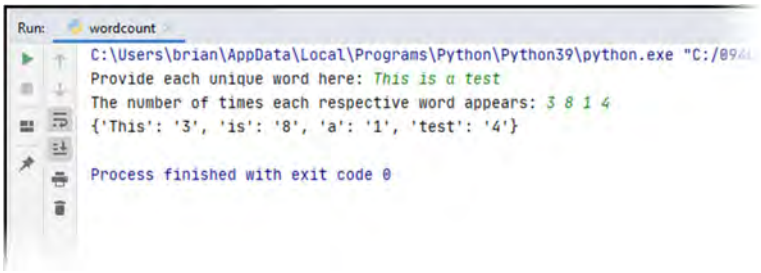
a) Run the program.

b) At the first prompt, type ***This is a test***

c) At the second prompt, type ***3 8 1 4***

	Note: This group of numbers is arbitrary. You can replace it with any other set of four numbers you want.

d) Verify that the dictionary prints to the screen, and that each word is a key and each key has its associated value.



4. Find the number of times the word "test" appears.

a) Change the print statement on line 15 as shown.

```
word_count[words[2]] = frequency[2]
word_count[words[3]] = frequency[3]
print("The word 'test' appears {} times.".format(word_count["test"]))
```

In this print statement, you are accessing the `word_count` dictionary, providing "test" as the key. The corresponding dictionary value stored under that key (4, if you use the same inputs as before) should be returned by the dictionary in the output.

b) Run the program using the same inputs you entered before (***This is a test*** and ***3 8 1 4***).

c) Verify that your print statement returns the value that was paired with the key "test".

```
Run: wordcount
C:\Users\brian\AppData\Local\Programs\Python\Python39\python.exe "C:/09402
Provide each unique word here: This is a test
The number of times each respective word appears: 3 8 1 4
The word 'test' appears 4 times.

Process finished with exit code 0
```

5. Check if a given word exists in the dictionary.

a) Click the line number for line 15 to select the whole line, and press **Delete** to remove the line.

b) Starting in line 15, write three new statements as shown.

```
word_count[words[2]] = frequency[2]
word_count[words[3]] = frequency[3]
check_word = input("Which word would you like to check? ")
word_exists = check_word in word_count
print("The word '{}' is in the dictionary: {}".format(check_word, word_exists))

# user_input = input("Provide words here: ")
# results_list = user_input.split()
```

- Line 15 prompts the user to identify which word should be checked, assigning it to the variable **`check_word`.**
- Line 16 assigns a value to the `word_exists` variable. The value will either be `True` (if the word to check is in the dictionary) or `False` (if it is not).
- The new print statement in line 17 outputs the result.

c) Run the program using the same inputs you entered before (***This is a test*** and ***3 8 1 4***).

d) When asked to check a word, provide any single word and view the outcome.

e) Verify that your code returns either `True` or `False`, depending on what word you checked.

```
Run: wordcount
C:\Users\brian\AppData\Local\Programs\Python\Python39\python.exe "C:/09
Provide each unique word here: This is a test
The number of times each respective word appears: 3 8 1 4
Which word would you like to check? test
The word 'test' is in the dictionary: True

Process finished with exit code 0
```

f) Close the program console.

6. Clean up the workspace.
   a) Select **File→Close Project**.
   b) Close the Welcome to PyCharm window.

Summary

In this lesson, you processed both ordered and unordered sequence data types. Being able to manipulate these data types will help you implement more complex and useful features into your applications.

Which sequence type do you think you'll use more often, lists or tuples? Why?

In what real-world situation might you want to use the set comparison methods `difference()` and `intersection()`?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

4 Writing Conditional Statements and Loops in Python

Lesson Time: 1 hour, 30 minutes

Lesson Introduction

Other than storing data, programming languages like Python® execute logical processes in order to determine what to do in a program. Conditional statements and loops are the primary ways that you can control the logical flow of a program, and they are found in nearly all Python-developed software. In this lesson, you'll take advantage of both to produce a more complex application.

Lesson Objectives

In this lesson, you will:

- Write conditional statements.
- Write loops.

TOPIC A

Write a Conditional Statement

In this topic, you'll leverage Python's ability to create objects that decide what to do based on certain conditions.

Conditional Statements

A ***conditional statement*** is an object that tells the program it must make a decision based on various factors. If the program evaluates these factors as true, it executes the block of code contained in the conditional statement. When that block of code has run, execution continues with the code following the conditional statement. If false, the program skips the block of conditional code and jumps directly to the code following the conditional statement.

Conditional statements are fundamental to most programs, as they help you control the flow of executed code. For example, if a user enters some input, you might want to process that input differently based on a number of factors. The user might press one navigation button in a web app and not another. Rather than sending them to both web pages, you'd only send the user to whichever they chose.

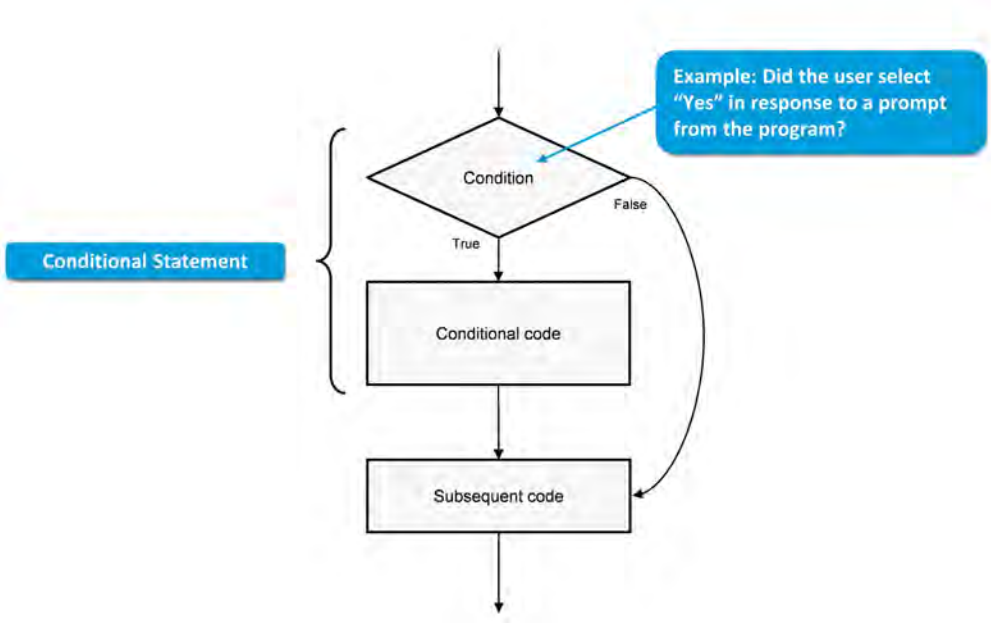


Figure 4-1: The basic flow of a conditional statement.

If Statements

Python uses the `if` statement as its conditional statement.

The syntax for `if` statements is as follows:

```
if condition:
    statement
```

If the condition is met, execute the statement below. That is the essential process of an `if` statement. Note that any code to be executed as part of the `if` statement must be indented. Also note the colon (`:`)—this is required after the condition of an `if` statement and before the actual statement to be executed. Now consider the syntax with real values:

```
a = 2
b = 3
c = 5

if a + b == c:
    print("Successful!")
```

Once the program evaluates this expression and determines it is true, `Successful!` is printed to the command line.

Note: Conditions can use many different types of operators and are not just limited to simple arithmetic.

Now consider what would happen if the condition evaluates to false:

```
if a + c == b:
    print("Successful!")
```

In fact, nothing happens. Python doesn't produce an error, as nothing about this code causes problems in execution. Python simply skips over the indented statement because the condition for it executing wasn't met.

But what if you want the false value of a conditional statement to execute some code, rather than just moving on? For that, you can modify the typical `if` statement by making it an `if...else` statement:

```
if a + c == b:
    print("Success!")
else:
    print("Failure!")
```

If Python does not evaluate the condition to true, it will execute everything in the `else` branch of the statement. This is useful when there are multiple conditions you need to evaluate, but only one requires a unique action, and the rest can simply be treated the same way.

On that note, what if there are multiple conditions and you want to treat each *differently*? In this case, you can essentially combine `if` and `else` into the `elif` branch. For example:

```
if a == b:
    print("A is B")
elif a == c:
    print("A is C")
elif b == c:
    print("B is C")
else:
    print("Failure!")
```

After it evaluates the first `if`, Python will go down each successive `elif` and determine its truth value. Python will stop once it finds a true condition and executes its code. As before, you can end the `if` and `elif` branches with `else` to capture every other condition that you didn't explicitly call out.

Nesting `if` Statements

You can also nest `if` statements within each other to have Python evaluate multiple layers of conditions. For example:

```
if a + b == c:
    if b + c == a:
        print("Success!")
    else:
        print("Failure!")
else:
    print("Failure!")
```

In this conditional statement, Python must evaluate `a + b == c` before it can even start evaluating `b + c == a`. There are several possible outcomes or "paths" involved in this statement.

Comparison Operators

Comparison operators test the relation between two values. These are most commonly used in conditional statements. There are several different ways to compare values in Python, some of which are described in the following table.

Operator	*Definition*	*Example*
`==`	Checks if both operands have an equal value. Evaluates to true or false.	`2 == 4` is false.
`!=`	Checks if operands do not have an equal value. Evaluates to true or false.	`2 != 4` is true.
`>`	Checks if left operand is greater in value than right operand. Evaluates to true or false.	`2 > 4` is false.
`<`	Checks if left operand is less in value than right operand. Evaluates to true or false.	`2 < 4` is true.
`>=`	Checks if left operand is greater in value or equal in value to right operand. Evaluates to true or false.	`2 >= 2` is true.
`<=`	Checks if left operand is less in value or equal in value to right operand. Evaluates to true or false.	`4 <= 2` is false.

Note: This is not an exhaustive list of comparison operators. For more information, navigate to **http://www.tutorialspoint.com/python/python_basic_operators.htm**.

Logical Operators

Another common operator in conditional statements is the ***logical operator***. Logical operators connect multiple values together so they can be evaluated. There are several different ways to connect multiple values in Python, which are described in the following table.

Operator	*Definition*	*Example*
`and`	Checks if both operands are true. Evaluates to true or false.	`2 > 3 and 4 > 3` is false.
`or`	Checks if at least one of the operands is true. Evaluates to true or false.	`2 > 3 or 4 > 3` is true.
`not`	Negates an operand. Evaluates to true or false.	`not(3) == 4` is true.

Identity Operators

Identity operators check to see if both operators point to the same location in memory. The following are the two identity operators.

Operator	*Definition*	*Example*
`is`	Returns true if the left operand points to the same memory address as the right operand.	`100 is 10` is false.

Operator	Definition	Example
`is not`	Returns true if the left operand does not point to the same memory address as the right operand.	`"Hello" is not "World"` is true.

At first glance, it may seem like these two operands are equivalent to == and `!=`. If you type `100 == 100` in the interactive console it will return `True`, as will `100 is 100`. However, this is because Python caches small integer objects in memory. Since identity operators test if an object is pointing to the same memory value as another object, small integers like `10` will point to the same address in memory. See what happens when you test a larger integer that Python doesn't cache in memory:

```
>>> 100 == 10**3
True
>>> 100 is 10**3
False
```

Although `10**3` is the same value as `100`, it does not point to the same memory address, so using an identity operator will return `False`.

Identity operators are often used when testing `True` or `False` values. For example:

```
x = False
if x is False:
    print("Success.")
```

Order of Operations

When you include arithmetic, comparison, logical, and identity operators, this is the order of operations:

- `**`
- `* / &`
- `+ -`
- `<= < > >=`
- `== !=`
- `is is not`
- `and or not`

Note: Like with arithmetic, you can control the order of logical, comparison, and identity operations by wrapping them in parentheses.

Guidelines for Writing Conditional Statements

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following to help you write conditional statements in your Python programs.

Write Conditional Statements

When writing conditional statements:

- Use an `if` statement to test for a simple condition.
- Add an `else` statement at the end of a conditional to account for all other conditions.
- Use successive `elif` statements to test multiple conditions and execute different code for each.
- Make use of comparison operators like == and < in conditional statements to test how values and variables relate to one another.

- Make use of logical operators like `or`, `and`, and `is not` to determine how values are connected together in a condition.
- Use identity operators `is` and `is not` to check `True` and `False` values.
- Place a colon (`:`) at the end of the condition in an `if` and `elif` statement, as well as right after an `else` statement.
- On the line below the condition or `else` statement, indent the code you want the statement to execute.
- Keep the order of operations in mind when writing conditions for an `if` statement.
- For more complex processing, nest `if` statements within each other.

ACTIVITY 4-1
Writing Conditional Statements

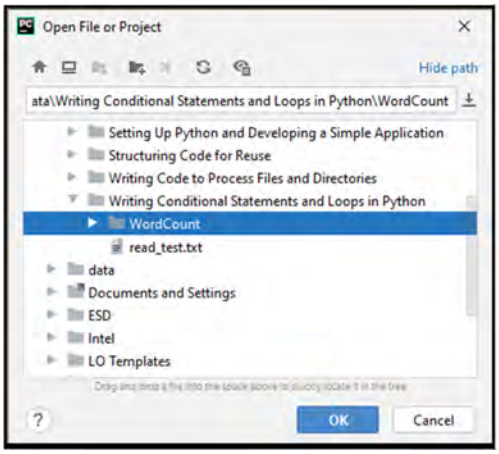
Data File
C:\094021Data\Writing Conditional Statements and Loops in Python\WordCount\wordcount.py

Scenario
Your program won't function very well unless it can make decisions based on a number of factors—especially user input. To account for these decisions, you'll need to write conditional `if` statements to anticipate the various ways your program can execute. You'll start by taking the three main questions you ask the user—which input file they want to analyze, whether they want to strip common words from the results, and whether they want the results output to a file—and execute certain code based on the user's input. For each respective question, these are the conditions you'll check for:

1. Does the file in the specified path exist?
2. Has the user selected yes or no to stripping common words from the results?
3. Has the user selected yes or no to outputting the text to a file?

---

1. Launch PyCharm and open the WordCount project.
   a) Launch PyCharm if you have not left it running from a previous session.
   b) In the Welcome to PyCharm window, select **Open**.
      The **Open File or Project** dialog box is shown.
   c) Beneath the **094021Data** folder, expand **Writing Conditional Statements and Loops in Python**, and select the **WordCount** project folder.



   d) Select **OK**.
      The project is loaded and configured to use the Python 3.9 interpreter.

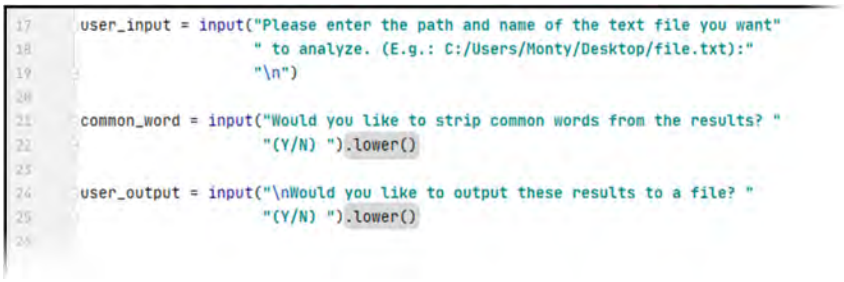
e) In the **Project** pane, expand the **WordCount** folder if necessary, and double-click **wordcount.py** to show the source code in the editor.

```
5   COMMON_WORDS = {"the", "be", "are", "is", "were", "was", "am",
6                   "been", "being", "to", "of", "and", "a", "in",
7                   "that", "have", "had", "has", "having", "for",
8                   "not", "on", "with", "as", "do", "does", "did",
9                   "doing", "done", "at", "but", "by", "from"}
10
11  read_input = ["This", "is", "a", "test!", "this", "is,", "another", "test.",
12                "TesT", "value", "this", "!test", "is?"]
13  read_wordlist = ["this", "is", "a", "test"]
14
15  word_count = {}
16
17  user_input = input("Please enter the path and name of the text file you want"
18                     " to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):"
19                     "\n")
20
21  common_word = input("Would you like to strip common words from the results? "
22                      "(Y/N) ")
23
24  user_output = input("\nWould you like to output these results to a file? "
25                      "(Y/N) ")
```

- Experimental code that you added previously has been removed.
- In lines 5 through 9, a set constant containing a list of common words has been added to the source code.
- In lines 11 through 13, two lists have been added to provide some temporary data for testing. Eventually, these lists will hold real values that you'll get from two text files, but for now they have been initialized with data you can use to test your loops. Remember that `read_input` will hold the written work that your program analyzes, and `read_wordlist` will be the list of English words it gets compared to.
- Line 15 initializes an empty dictionary, which you will use to hold key-value pairs containing word counts.
- In lines 17 through 25, the three user input questions you had added earlier are ready for you to expand on.

2. Force the user's answers to lowercase.

a) In lines 22 and 25, add two calls to the `String` object's `lower( )` method as shown.



```
17  user_input = input("Please enter the path and name of the text file you want"
18                     " to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):"
19                     "\n")
20
21  common_word = input("Would you like to strip common words from the results? "
22                      "(Y/N) ").lower()
23
24  user_output = input("\nWould you like to output these results to a file? "
25                      "(Y/N) ").lower()
26
```

Now, any letters that the user types in uppercase will be immediately converted to lowercase. This makes processing the upcoming conditional statements easier.

3. Create a conditional statement that handles bad file path input.

a) Position the insertion point in line 20 and press **Enter**.

b) Type two new lines as shown.

```
17  user_input = input("Please enter the path and name of the text file you want"
18                     " to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):"
19                     "\n")
20
21  if user_input is False:
22      print("The file you specified does not exist.\n")
23  common_word = input("Would you like to strip common words from the results? "
24                      "(Y/N) ").lower()
```

c) Position the insertion point at the end of line 22.
d) Press **Enter** to go to the next line and press **Backspace** to remove the automatic indentation.
e) Type **`else:`** and press **Enter**.
f) Examine the code up to this point.

```
17  user_input = input("Please enter the path and name of the text file you want"
18                     " to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):"
19                     "\n")
20
21  if user_input is False:
22      print("The file you specified does not exist.\n")
23  else:
24      
25  common_word = input("Would you like to strip common words from the results? "
26                      "(Y/N) ").lower()
27
28  user_output = input("\nWould you like to output these results to a file? "
29                      "(Y/N) ").lower()
```

- Line 21 checks to see if the input value is `False`.
- If this is the case, then you can't process the file, so line 22 prints a message indicating that the requested file doesn't exist.
- Line 23 starts the `else` block. Every statement indented under this section will execute if the file actually does exist.
- PyCharm has automatically indented the insertion point so you can start typing code for the *else* branch, but you already have two existing statements (in lines 25 through 29) that should be included in this block, so these statements need to be indented.

Note: Later, you'll write code to actually look for the file and return `False` if the file doesn't exist. For now, you'll temporarily get this `True` or `False` value from user input.

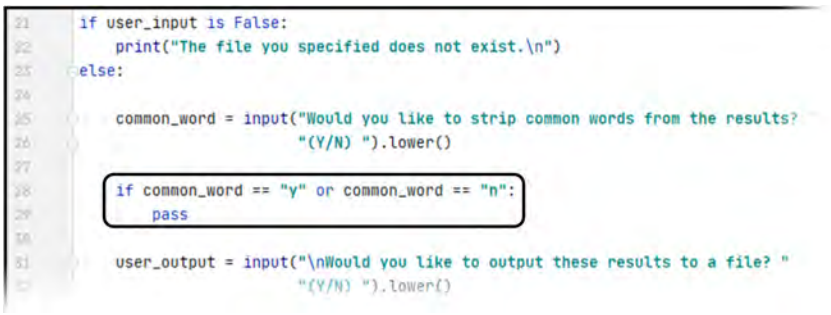
g) Select lines 25 through 29 and press **Tab** to indent them.

```
21  if user_input is False:
22      print("The file you specified does not exist.\n")
23  else:
24
25      common_word = input("Would you like to strip common words from the results? "
26                          "(Y/N) ").lower()
27
28      user_output = input("\nWould you like to output these results to a file? "
29                          "(Y/N) ").lower()
```

4. Create a conditional statement that handles a user's decision to strip common words from the results.

a) Position the insertion point at the end of line 26 and press **Enter** twice.

b) Type two new lines of code as shown.

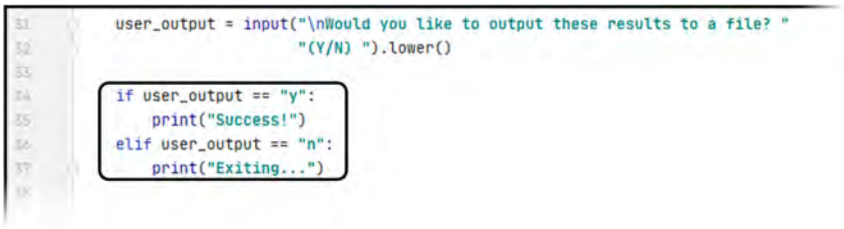


- If the user answers y or n, the loop will execute.
- For now, you've included the `pass` statement as a placeholder.
- Later, you'll replace this placeholder with code that strips the words in `COMMON_WORDS` from the results.

5. Create a conditional statement that handles a user's decision to output the results to a text file.

a) Position the insertion point at the end of line 32 and press **Enter** twice.

b) Type the following code:



- If the user agrees to outputting the results to a file, the code in the `if` branch (line 34) will execute. For now, this will just print `"Success!"` to the command line, but later, you'll write code to actually output the results to a file.
- If the user says no to the question, the program will print `"Exiting..."` to the console (line 37), with the intention that the program will terminate at this point.

6. Test your conditions.

a) Run the program and answer the first prompt by entering ***This is a test***

b) When asked to strip common words, enter ***y***

Note: Because the code for `user_input` and `common_words` is incomplete, their behavior will not change based on your input.

c) Verify that, depending on how you answer the third question, Python will output something different. Try lowercase and uppercase "y" and "n", and also try input that isn't either of these characters, like "no" or "yes". Restart the program as necessary to test additional input.

When more of your code is fleshed out, these conditional statements will allow you to guide users down certain paths of your program depending on their choices.

7. **When you input "no" to the third question, why does it output nothing to the console?**

TOPIC B

Write a Loop

Now that you've written conditional statements, you can move on to loops, another way to control the flow of logic in a program.

Loops

Another useful way to control flow in a program's code is by implementing loops. A ***loop*** is any statement that executes code repeatedly. The loop repeats until a certain exit condition is met. Code within the loop runs each time the loop repeats. When the necessary exit condition is met, the loop is finished, and the flow of execution resumes with the code that follows outside the loop.

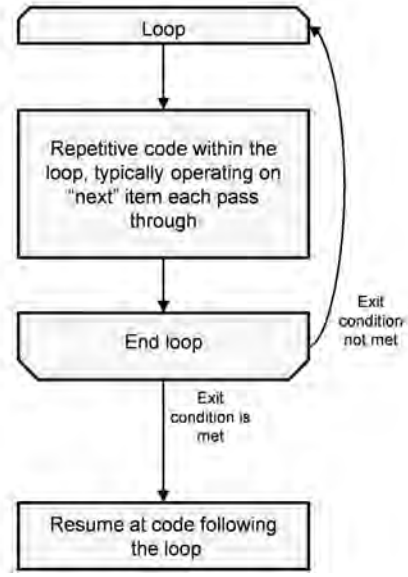


Figure 4-2: The basic flow of a loop.

In Python, there are two main kinds of loops: `while` and `for` loops.

While Loops

A `while` loop executes code repeatedly until some condition is met. For example, consider a typical desktop program. You want the window of the app to stay active until the user voluntarily selects the close button. So, you'd wrap the relevant code in a loop, with the condition being that the user selects the close button. Once the program meets this condition (i.e., the user selects close), the program can terminate.

The syntax for a `while` loop is:

```
while condition:
    statement
```

Notice that the syntax is very similar to an `if` statement. Also, like in an `if` statement, the condition in a `while` loop can contain an expression using various operators. For example:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Count starts at one, and while it is less than or equal to 5, two things happen:

- The count is printed to the console.
- The count is incremented by one. Note that using `+=` achieves the same effect as `count = count + 1`; i.e., Python is reassigning the variable to whatever it currently is plus one.

So, the first time the loop executes, `count` starts as `1` and becomes `2`. Then, the very same print and variable assigning code loops again, with `count` starting at `2` and becoming `3`. This process repeats until `count` is equal to `6`, and then Python skips the code in the `while` loop. The output of this code is as follows:

```
1
2
3
4
5
```

If the condition for the `while` loop to run is never met, Python will never execute the code inside of the loop.

For Loops

A `for` loop iterates through an object a certain number of times, depending on what you specify. Unlike a `while` loop, a `for` loop does not depend upon a condition being evaluated to false for it to stop or never begin in the first place. So, `for` loops are meant to always execute code a given number of times. This makes them ideal for processing iterable objects like lists, ranges, and other data structures.

The syntax for a `for` loop is:

```
for iterator in sequence:
    statement
```

As you can see, the basic structure (indentation, colon, etc.) is familiar. For a practical example, observe the following:

```
my_list = [1, 2, 3, "Four", "Five"]
for i in my_list:
    print(i)
```

First, you define a simple list. Next, in the `for` loop, you assign `i` as the iterator. This is often used arbitrarily as the iterator value, but you can choose a more descriptive name if you prefer. The `in my_list` tells the Python to loop through the list you created. Within the loop is the print statement that outputs the iterator value. Essentially, the `for` loop assigns the first value in the list (`1`) to `i`, prints `i`, then loops and assigns `2` to `i`, prints `i`, and so on until it gets to the end of the list. Here is the output:

```
1
2
3
Four
Five
```

When it comes to iterating data structures, `for` loops are the preferred method in Python; `while` loops are less common and are typically reserved for user input.

Note: Like `if` statements, both loop types can be nested for greater complexity.

The else Branch in Loops

The `else` branch can also be used in loops. In a `while` loop, an `else` branch will execute once the condition becomes false:

```
count = 1
while count <= 5:
    print(count)
    count += 1
else:
    print("Done counting.")
```

In a `for` loop, the `else` branch will execute once the loop is finished iterating:

```
my_list = [1, 2, 3, "Four", "Five"]
for i in my_list:
    print(i)
else:
    print("We've reached the end of the list.")
```

Loop Control

Within loops, there are statements that give you greater control over how the code is executed. The control statements are `break`, `continue`, and `pass`.

The `break` statement terminates the current loop and executes the very next statement. This is useful if you need to end a loop's execution due to some change in condition. For example:

```
my_list = [1, 2, 3, 4, 5.2, 6]
for i in my_list:
    if i % 1 != 0:
        break
    print(i)
```

In this example, Python iterates through each entry in `my_list`. If it comes to a value that is not an integer, it breaks out of the loop. The resulting output is:

```
1
2
3
4
```

The `continue` statement will stop Python from executing the code in that particular iteration, while allowing the rest of the iterations to proceed. For example:

```
my_list = [1, 2, 3, 4, 5.2, 6]
for i in my_list:
    if i % 1 != 0:
        continue
    print(i)
```

Instead of terminating the entire loop, Python will simply ignore the iteration in which `5.2` appears. The output is therefore:

```
1
2
3
4
6
```

Lastly, the `pass` statement essentially tells Python to do nothing. It is typically used as a placeholder before writing actual statements.

```
my_list = [1, 2, 3, 4, 5.2, 6]
for i in my_list:
    pass
```

Guidelines for Writing Loops

Use the following to help you write loops in your Python programs.

Write Loops

When writing loops:

- Use `while` loops to execute code based on a condition that is either true or false.
- Use `while` loops primarily for processing user input.
- Use `for` loops to iterate through objects like lists, ranges, and other data structures.
- Use `for` loops to define exactly how many times to iterate through these objects.
- Always indent the statement to be executed in a loop, and end the first line of a loop with a colon (`:`).
- Nest loops within each other to achieve more complexity in processing data.
- Use an `else` branch in a `while` loop to tell Python what to do when the condition becomes false.
- Use an `else` branch in a `for` loop to tell Python what to do once the loop completes.
- Use the `break` statement if you need to terminate a loop based on some condition.
- Use a `continue` statement if you need to stop a particular iteration in a loop.
- Use the `pass` statement as a placeholder for code in a loop you plan to write later.

ACTIVITY 4-2
Writing While Loops

Scenario
As your program stands now, if a user enters an unexpected input to your yes or no questions, the program will simply carry on. In the case of the last question, the program will just stop entirely. This is not ideal behavior because users sometimes make mistakes. Your program should react gracefully to these mistakes. So, you'll need to construct `while` loops to keep your program running as long as the user hasn't given it a correct input.

---

1. Write a `while` loop to keep the whole program running.
   a) Position the insertion point in line 16 and press **Enter**.
   b) Type **`while True:`** and press **Enter**.
      Writing `while True` is essentially telling Python to execute the following code as long as `True` is `True`. Since this is always the case, the code will run indefinitely until it encounters a `break` statement.
   c) Highlight all of the code under the `while` loop—from lines 19 through 39—and press **Tab**.
      You want all of this code to run indefinitely, so indenting it places it under the control of the loop.
   d) Verify that this entire block of code is positioned within the `while` loop.

```
15  word_count = {}
16
17  while True:
18
19      user_input = input("Please enter the path and name of the text file you want"
20                         " to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):"
21                         "\n")
22
23      if user_input is False:
24          print("The file you specified does not exist.\n")
25      else:
26
27          common_word = input("Would you like to strip common words from the results? "
28                              "(Y/N) ").lower()
29
30          if common_word == "y" or common_word == "n":
31              pass
32
33          user_output = input("\nWould you like to output these results to a file? "
34                              "(Y/N) ").lower()
35
36          if user_output == "y":
37              print("Success!")
38          elif user_output == "n":
39              print("Exiting...")
```

2. Write a `while` loop to keep the second input question running until given an acceptable answer.
   a) Position the insertion point in line 26 and press **Enter**.

b) Press **Tab** twice to align the insertion point with `common_word`

```
    if user_input is False:
        print("The file you specified does not exist.\n")
    else:

        common_word = input("Would you like to strip common words from the results? "
                            "(Y/N) ").lower()
```

c) Type another `while` statement as shown.

```
    if user_input is False:
        print("The file you specified does not exist.\n")
    else:

        while common_word != "y" or common_word != "n":
        common_word = input("Would you like to strip common words from the results? "
                            "(Y/N) ").lower()

        if common_word == "y" or common_word == "n":
            pass

        user_output = input("\nWould you like to output these results to a file? "
                            "(Y/N) ").lower()
```

d) Select the `input` and `if` statements in lines 28-32.

```
    else:

        while common_word != "y" or common_word != "n":
        common_word = input("Would you like to strip common words from the results? "
                            "(Y/N) ").lower()

        if common_word == "y" or common_word == "n":
            pass

        user_output = input("\nWould you like to output these results to a file? "
                            "(Y/N) ").lower()
```

e) Press **Tab** to indent the selected block.

As long as the user doesn't answer the question correctly (with "y" or "n"), the question will keep prompting for new input.

3. Write a `while` loop to keep the third input question running until given an acceptable answer.

a) Place the insertion point on line 33 and press **Enter** to create a new line.

b) Press **Tab** twice to position the insertion point in line with `user_output`, and enter another `while` statement as shown.

```
            if common_word == "y" or common_word == "n":
                pass

        while user_output != "y" and user_output != "n":
        user_output = input("\nWould you like to output these results to a file? "
                            "(Y/N) ").lower()

        if user_output == "y":
            print("Success!")
```

c) Indent the rest of the code (lines 35-41) so that it is contained by this `while` loop as shown.

```
31              if common_word == "y" or common_word == "n":
32                  pass
33
34          while user_output != "y" and user_output != "n":
35              user_output = input("\nWould you like to output these results to a file? "
36                                  "(Y/N) ").lower()
37
38              if user_output == "y":
39                  print("Success!")
40              elif user_output == "n":
41                  print("Exiting...")
42
```

Like the previous `while` loop, as long as the user doesn't supply a "y" or "n" answer to the question, it will keep being asked, and the subsequent code will keep executing.

4. Add control statements where the loops need to do something different.

a) Position the insertion point at the end of line 24, and press **Enter**.

b) Type **`continue`**

```
17  while True:
18
19      user_input = input("Please enter the path and name of the text file you want"
20                         " to analyze. (E.g.: C:/Users/Monty/Desktop/file.txt):"
21                         "\n")
22
23      if user_input is False:
24          print("The file you specified does not exist.\n")
25          continue
26      else:
27
28          while common_word != "y" or common_word != "n":
```

If the program doesn't find the file requested by the user, the top-level `while` loop will restart the code under it. In this case, the loop will keep asking the file's path until it gets what it's looking for.

c) On line 33, change the `pass` placeholder to **`break`**.

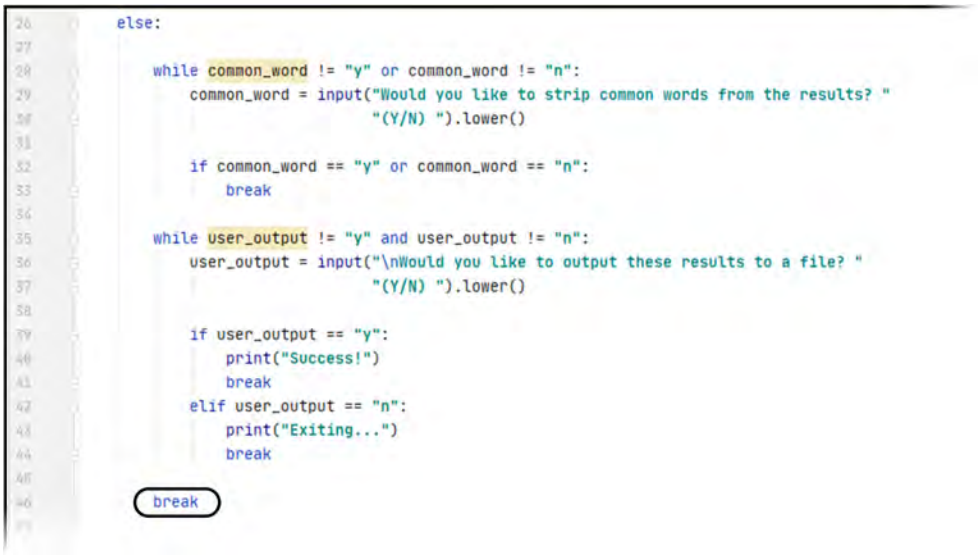
You'll write the code to handle this option later, but for now, you'll break out of this `if` statement if the user provides an acceptable answer ("y" or "n").

d) Add break statements after the "Success!" and "Exiting..." print lines.

```
28          while common_word != "y" or common_word != "n":
29              common_word = input("Would you like to strip common words from the results? "
30                                  "(Y/N) ").lower()
31
32              if common_word == "y" or common_word == "n":
33                  break
34
35          while user_output != "y" and user_output != "n":
36              user_output = input("\nWould you like to output these results to a file? "
37                                  "(Y/N) ").lower()
38
39              if user_output == "y":
40                  print("Success!")
41                  break
42              elif user_output == "n":
43                  print("Exiting...")
44                  break
45
```

When your program has finished outputting the results to a file, or if the user doesn't want the results output to a file, this will terminate the `while` loop and the program will stop.

e) Add a `break` statement at the end of the source code, nested within the `else` branch above.



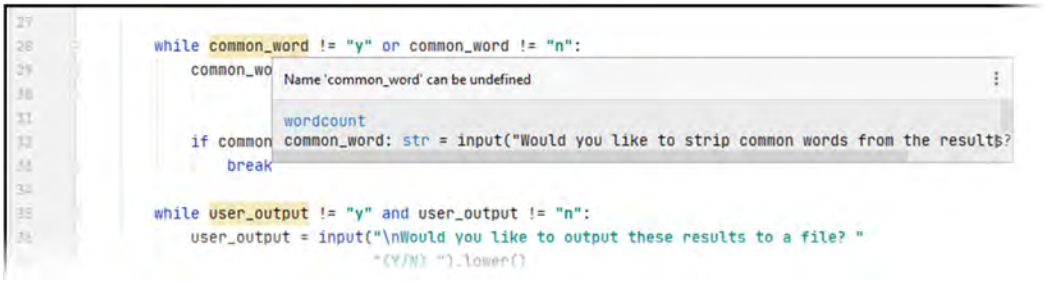
If the program enters the `else` branch (i.e., the user provides a valid file and path), the program can exit when all the code in the `else` branch is finished executing.

5. Define variables so the loops can execute properly.

a) Observe the highlighted variables on lines 28 and 35.

The PyCharm editor has identified a problem in the code.

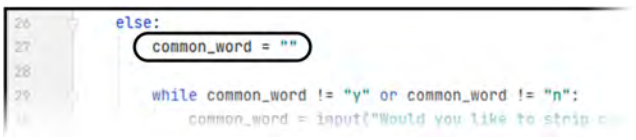
b) Point at the highlighted variables on lines 28 and 35, and pause to view the warning message.



The way the code is currently written, during execution, it is possible that these variables may not be assigned a value because both are referenced in the `while` condition, but are only assigned a value after that point in the code. You can resolve this by assigning them an initial value before the `while` statement.

c) Position the insertion point at the end of line 26 and press **Enter**.

d) Type **`common_word = ""`**, making sure that the statement is within the `else` branch.

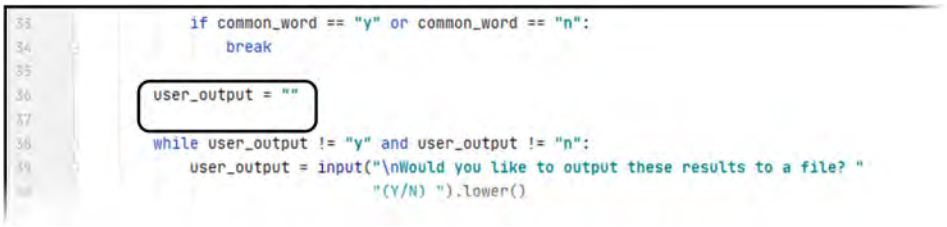


This defines common_word and initializes it with an empty string, which will be filled during the program's execution. The variable can now be evaluated when the `while` condition is evaluated the first time.

e) Position the insertion point in line 35 and press **Enter**.

f) Press **Tab** twice to indent.

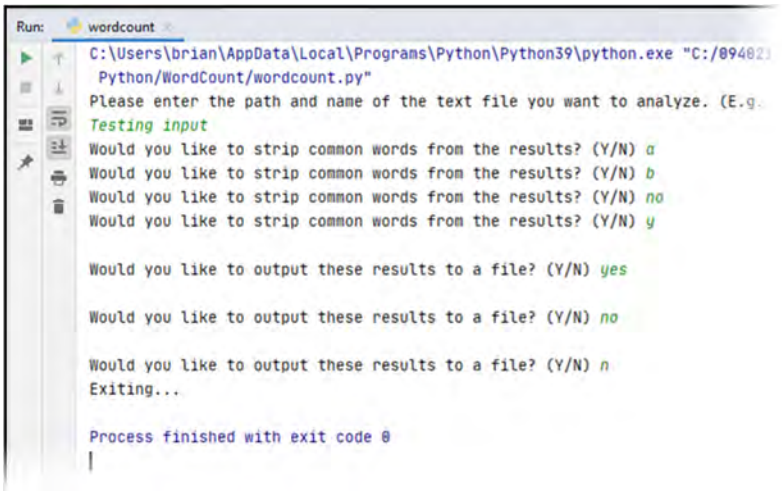
g) Type a statement to initialize `user_output`, followed by an empty line, as shown.



h) Observe that PyCharm has removed the highlighting from `common_word` and `user_output`.

6. Test out your input loops.

a) Run the program.
b) Enter any test input for the first prompt.
c) For the second question, enter any input other than "y" or "n".
d) Verify that, until you answer "y" or "n", the question repeats.
e) Do the same for question three, and verify that it acts the same way.



ACTIVITY 4-3
Writing For Loops

Scenario
Now you're ready to write the core processes in your program. The `for` loop is an easy and efficient way to iterate through structures like lists and dictionaries, so you'll be using several for a variety of purposes. One `for` loop will strip punctuation from all the words it finds in a file. Otherwise, for example, any words that end a sentence with a period will be counted as if that period is a part of the word. Your second major loop will actually do the comparison between the user's input file and the list of English words that you'll be using. When Python finds that a word in the user's file is also in the wordlist, Python will consider it valid and start counting. Every time this word appears, the count will increase by one. The last major `for` loop you'll write will format and output the end results to the console, while deciding whether or not to suppress the common words you defined earlier.

---

1. Create a loop to strip punctuation from the input file.
   a) Position the insertion point in line 14 and press **Enter**.
   b) Starting on line 15, type the code as shown.

```
5   COMMON_WORDS = {"the", "be", "are", "is", "were", "was", "am",
6                   "been", "being", "to", "of", "and", "a", "in",
7                   "that", "have", "had", "has", "having", "for",
8                   "not", "on", "with", "as", "do", "does", "did",
9                   "doing", "done", "at", "but", "by", "from"}
10
11  read_input = ["This", "is", "a", "test!", "this", "is,", "another", "test.",
12                "TesT", "value", "this", "!test", "is?"]
13  read_wordlist = ["this", "is", "a", "test"]
14
15  count = 0
16
17  for word in read_input:
18      word = word.lower()
19      # Removes common punctuation so it's not part of the word.
20      read_input[count] = word.strip(".,?!\"\';:()")
21      count += 1
22
23  word_count = {}
24
25  while True:
```

This routine finds the first word in the input file, adds it to a list after stripping any punctuation, then does the same thing for every other word in the file. The loop ends when the last word in the file is read.

- Line 15 initializes a counter you'll be incrementing in the loop to read through each index of the `read_input` list.
- Line 17 is the start of the loop. Python will iterate through each word in the `read_input` list.
- On line 18, all words are forced to lowercase so that they're uniform.
- On line 20, the value at a certain index of `read_input` is set to whatever the word is, minus any extraneous punctuation. Basically, `strip()` removes the provided characters from the word, if they appear.
- Line 21 increments the count by one, so that the list index for the next iteration of the loop also increments by one.

2. Write a loop to compare the input file to an existing English wordlist.

a) Position the insertion point in line 24 and press **Enter**.
b) Starting in line 25, enter the following code block:

```
23  word_count = {}
24
25  for word in read_input:
26      word = word.lower()
27      if word in read_wordlist:
28          if word not in word_count:
29              word_count[word] = 1
30          else:
31              word_count[word] += 1
32      else:
33          continue
34
35  while True:
36
37      user_input = input("Please enter the path and name of the text file you want:
```

This routine checks every word in the input text file for its presence in the wordlist. Once it finds a match, it checks to see if that word was already processed. If it was, the count goes up by one. If not, the word is added to the list with an initial count of one. When this routine is done, the word_count dictionary will be populated with key-value pairs, where each key is a unique word, and its associated value will contain the number of times the word appears.

- On line 25, Python reads through each word it finds in the `read_input` list.
- On line 26, each word is forced to lowercase.
- On line 27, while still in the `for` loop, Python checks to see if the word in `read_input` also exists in `read_wordlist`.
- If the word is in the wordlist, then line 28 checks to see if the word has not yet been added to the `word_count` dictionary.
  - If the word doesn't exist in `word_count` yet, line 29 sets the word's value to `1`. (Remember, the word itself is used as the key.)
  - If the word has already been added to `word_count`, then line 31 increases the word count for that word by one.
- Line 32 starts the `else` branch, which deals with words *not* in the word list. If a word is not in the word list, line 33 tells the `for` loop to continue on with the next iteration (the next word).

3. Write a loop to print the results to the console.

a) Position the insertion point in line 34 and press **Enter**.

b) Type the code as shown.

```
32      else:
33          continue
34
35  choice = ""
36  items = []
37  results_list = []
38
39  # Truncates output if user wants to suppress common words.
40  for word in items[:50]:
41      if choice == "y" and word[0] not in COMMON_WORDS:
42          result = word[0] + ": " + str(word[1]) + " times"
43          results_list.append(result)
44          print(result)
45      elif choice == "n":
46          result = word[0] + ": " + str(word[1]) + " times"
47          results_list.append(result)
48          print(result)
49
50  while True:
```

- Lines 35 through 37 initialize objects used in the routine.
  - The `choice` string will be a yes or no input.
  - The `items` list will contain a sorted version of the dictionary you processed above. Each item in the list holds both the key and its value.
  - `results_list` will format the words and their counts in a more output-friendly context.
- Line 40 checks every item in the list of sorted `items`. The slice tells the loop to only go through the first 50 values.
- Line 41 evaluates whether the user says yes to suppressing common words from the results *and* the word at the first index (the dictionary key) is found in the `COMMON_WORDS` set. If both conditions are true, then the program will execute lines 42 through 44.
- Line 42 assigns `result` to the word and concatenates it with some additional text, along with its count. The `str()` statement turns the integer count into a string for formatting purposes.
- Line 43 appends this result to the `results_list`.
- Line 44 prints the individual result to the console.
- Lines 46 through 48 do essentially the same thing as 42 through 44, but every word is added.

So, the loop works through the first 50 items in the list. Once the list is sorted, these will be the 50 words with the highest frequency. If the user wants to suppress common words from the results, the `results_list` will not include the values in `COMMON_WORDS`. If the user doesn't want to suppress these words, they'll appear in the list. In both instances, the list is formatted so that every index holds a string with a word, its frequency, and "times" at the end. For example: `the: 8 times` is one possible result.

4. Stage some values so that you can test your `for` loops.

a) In lines 11 through 13, observe that `read_input` and `read_wordlist` are being set to contain arbitrary test strings.

```
11  read_input = ["This", "is", "a", "test!", "this", "is,", "another", "test.",
12                "TesT", "value", "this", "!test", "is?"]
13  read_wordlist = ["this", "is", "a", "test"]
```

These will eventually hold real values that you'll get from two text files, but for now you can test your loops with these staged lists. Remember that `read_input` will be the written work that your program analyzes, and `read_wordlist` will be the list of English words it gets compared to.

b) In the source code, scroll down to line 35 and change the value assigned to the `choice` variable to **`"n"`**

c) On the next line, change the `items` variable to **`sorted(word_count.items())`**

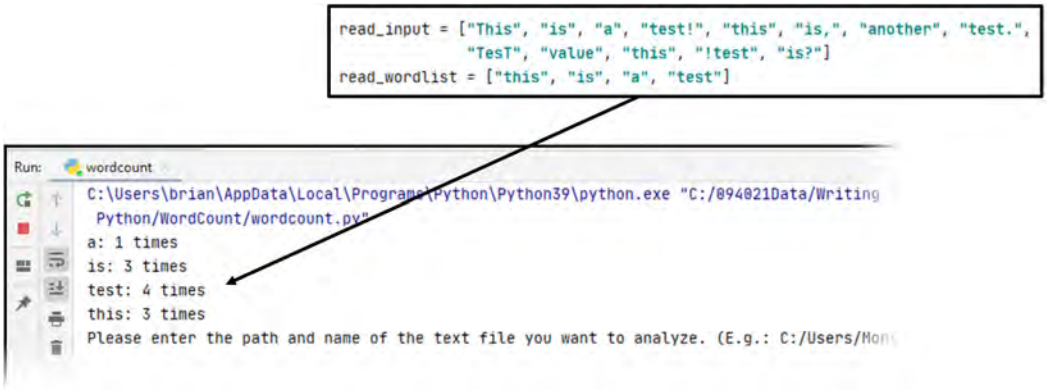
```
choice = "n"
items = sorted(word_count.items())
results_list = []
```

Python's built-in `sorted()` function creates a new list based on an existing structure, like a dictionary. Here, its first argument is the `word_count` variable you defined above. Because of `.items()` after the variable, each index of the list contains a tuple of two items: the dictionary's key (the word) and its value (the number of times the word appears). You'll later use `sorted()` to actually sort the results by descending frequency.

5. Test the program.

a) Run the program.

b) Verify that your program prints out the `results_list`: each word followed by the number of times it appears. Notice that, compared to the arbitrary values in `read_input`, these results have stripped punctuation and every character is in lowercase format.



c) Stop the running program.

6. Clean up the workspace.

a) Select **File→Close Project**.

b) Close the Welcome to PyCharm window.

Summary

In this lesson, you implemented conditional statements and loops into your code. The former will allow your program to respond to changes in user input or program state, and the latter will help you process repetitive logic more efficiently and easily.

What are some common conditions in your apps that you'd want to test for in a conditional statement?

When are `for` loops more useful than `while` loops in your program?

Note: To learn more about streamlining code with the help of loops, check out the LearnTO **Create Data Structures Using Comprehension** presentation from the **LearnTO** tile on the CHOICE Course screen.

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

5 Structuring Code for Reuse

Lesson Time: 2 hours, 20 minutes

Lesson Introduction

Object-oriented languages excel at giving programmers the tools they need to make coding easier. Reusing code not only allows you to develop at a quicker pace, but it also makes your source code more readable. Inefficient and ugly code is incredibly difficult to maintain, so you'll leverage Python's built-in reusable objects to avoid this pitfall.

Lesson Objectives

In this lesson, you will:

- Define and call functions.
- Define and instantiate classes.
- Import and use modules.

TOPIC A

Define and Call a Function

Python® allows you to organize subroutines for easy reuse all throughout your source code. In this topic, you'll write Python code to both define and call functions.

Function Definition and Calling

As you've seen before, a function is a block of code that is meant to be reused. The syntax for defining a function is similar to other objects you've seen, such as conditional statements and loops. The first line of a function includes its name and ends in a colon, and the code within a function is indented. However, functions always begin with the statement `def` and include open and closed parentheses. The following is an example of a simple function definition:

```
def my_function():
    print("This is a function.")
```

Notice that the first parentheses are empty. This is acceptable in Python, but most functions are designed to hold variables within these parentheses. These variables are called ***arguments*** or ***parameters***. An argument is any value that is "passed" into the function when it is used elsewhere. When you pass an argument into the function, that function uses whatever value you provided in its code. For example:

```
def questions(name, quest, favorite_color):
    print("Your name is {}.".format(name))
    print("Your quest is {}.".format(quest))
    print("Your favorite color is {}.".format(favorite_color))
```

The function's name is `questions` and it takes three variables as arguments: `name`, `quest`, and `favorite_color`. The code within the function uses these arguments to process the print statements.

But now that you have a function defined, how do you use it? Using functions, or more accurately, ***calling*** them, is the process of actually executing the code within a function. Calling a function is where you actually provide the values of the arguments that the function uses (if it has any). Up until this point, you've actually been calling several functions. For example, when you supply a value in `print()`, you're passing that value as an argument to Python's built-in `print()` function. Using the preceding `questions()` function, the following code calls the function and supplies three arguments:

```
questions("Lancelot", "to seek the Holy Grail", "blue")
```

This will print to the console:

```
Your name is Lancelot.
Your quest is to seek the Holy Grail.
Your favorite color is blue.
```

The preceding example uses all strings, but you can pass any data type as an argument. Notice that the example never explicitly defined either of the three variables like a typical `name = "Lancelot"` assignment. This means that each argument is local to its own function; that is, it won't be accessible outside the function without calling that function.

Returning Values in Functions

When you define a function, you can also return a value used within that function. Returning values in functions is useful when you want to assign variables to function calls. You can use the `return` statement to return a value. Consider the following function:

```
def age(year_of_birth, current_year):
    return current_year - year_of_birth
```

This function takes a person's year of birth and the current year as arguments. The `return` function returns that person's age by subtracting their year of birth from the current year. Now, consider the function call:

```
>>> john_age = age(1939, 2015)
>>> print("John is approximately {} years old.".format(john_age))
John is approximately 76 years old.
```

Because it returned a specific value, you are able to assign that value to a variable when calling the `age()` function. This gives you a lot more power to process the results of your function's code.

The Function Documentation String

Just like using docstrings for an entire program, you should use docstrings for each function.

```
def questions(name, quest, favorite_color):
    """Parrot user's answers back to them."""
    print("Your name is {}.".format(name))
    print("Your quest is {}.".format(quest))
    print("Your favorite color is {}.".format(favorite_color))
```

Notice that the docstring is placed as the first line within the function itself, as it's part of the function. This is useful if you use a tool to retrieve docstring information for specific functions.

For more complex functions that aren't as self-evident, your docstring should be more detailed. The following are important details to include in a complex function's docstring:

- Summarize its behavior.
- Document its arguments, if applicable.
- Document its return values, if applicable.
- Document any exceptions it may raise, if applicable.
- Document any additional information that may be significant, such as any self-imposed limitations or side effects of calling the function.

According to Python's official guidelines, a docstring should not just *describe* a function, but *prescribe* what that function does. In other words, think of the docstring as commanding the function to do something, rather than saying what it does. The example above correctly prescribes a function's behavior. The following block of code, on the other hand, uses a docstring incorrectly, as it's just a description:

```
def questions(name, quest, favorite_color):
    """Formats answers that the user provided."""
    print("Your name is {}.".format(name))
    print("Your quest is {}.".format(quest))
    print("Your favorite color is {}.".format(favorite_color))
```

Note: For more information on how to format docstrings, consult Python's official documentation at **https://www.python.org/dev/peps/pep-0257/**.

Scope

When you create an object such as a variable, it is not necessarily available to be used throughout the entire program. The programming concept of ***scope*** refers to where objects like variables are "visible" within a program, and where they are not. For example, a variable used inside of one function may not be visible to other functions.

Example A

```
1 # No variable created here.
2
3
4 def get_title(name, title):
5     # Variable created inside the function.
6     full_title = title + " " + name
7     print(full_title)
8
9 get_title("Eddie", "Sir")
10
11 print(full_title)
```

```
Sir Eddie
NameError: name 'full_title' is not defined

Process finished with exit code 1
```

Figure 5-1: Scope of variable created within a function.

In Python, the scope of an object depends on where it was created. Suppose a variable is created within a function as shown in line 6 of Example A. When Python encounters that line of code, it creates the variable in the context where it was encountered. So the variable `full_title` is created inside of the `get_title` function. The variable is said to be ***local*** in scope to that function, and it will only be visible within that function. The function is called in line 9, passing in values for the three arguments. Line 6 creates the variable, assigning it a string composed of the title and name that were passed into the function, with a space in-between. The `print` statement in line 7 successfully prints out the resulting string "Sir Eddie". However, after the function is called, line 11 of Example A attempts to print out the value of `full_title`. This causes Python to throw an exception because `full_title` doesn't exist within the scope of line 11.

Example B

```
1 # Global variable created here.
2 full_title = "global created"
3
4 def get_title(name, title):
5     # Variable created inside the function.
6     full_title = title + " " + name
7     print(full_title)
8
9 get_title("Eddie", "Sir")
10
11 print(full_title)
```

```
Sir Eddie
global created

Process finished with exit code 0
```

Figure 5-2: Local variable not visible in global scope.

Example B is nearly identical to Example A, except that a ***global*** variable named `full_title` is created in line 2. In this case, the `print` statement inside the function still sees the local variable named `full_title`. The function "sees" the local variable by that name, not the global variable with the same name. Outside the function, line 11 prints `full_title`, but due to *scope*, it accesses the variable that was created in line 2, which still contains the string "global created".

Example C

```
1 # Global variable created here.
2 full_title = "global created"
3
4 def get_title(name, title):
5     # Variable created inside the function.
6     full_title = title + " " + name
7     print(full_title)
8     return full_title
9
10 full_title = get_title("Eddie", "Sir")
11
12 print(full_title)
```

```
Sir Eddie
Sir Eddie

Process finished with exit code 0
```

(from local variable)

(from global variable)

Figure 5-3: Using a function return value.

Because of scope, some programmers moving into Python from another programming language may experience some frustration trying to use global variables within functions. However, Python operates sensibly, since using global variables as a way to pass data into and out of a function is generally considered a poor programming practice. This is because global variables run the risk of being modified anywhere within the code, potentially introducing errors or unwanted behavior in your program. For larger, more complex programs, it is much more difficult to spot when and where this can happen. So, it is generally advised to use parameters to provide a function with input, and to return output by providing a function return value, as shown in Example C. Using well-defined points of input and output in a function will help you avoid unwanted side effects in your programs.

Guidelines for Defining and Calling Functions

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following guidelines to help you work with functions in your Python programs.

Defining and Calling Functions

To work with functions in Python programs:

- When defining functions:
  - Place any necessary arguments that need to be passed into the function in parentheses.
  - Insert a docstring at the very first line of the function's code, detailing the behavior of the function.
  - Use the docstring to prescribe the function. Word it like a command, not a description.
  - Use `return` to pass values out of the function when it is called, especially when you need to assign the call to a variable.
  - Make sure the variables used in a function are within the proper scope. Don't use variables local to a function outside of that function.
  - Avoid using global variables in functions when possible.
- When calling functions:
  - Write the name of the function followed by open and closed parentheses.
  - Provide the necessary arguments that the function will work with in the parentheses.
  - Be mindful of which data types your arguments are, so that they do not cause errors when processed in the function.
  - Be aware of which values, if any, are returned from the function.

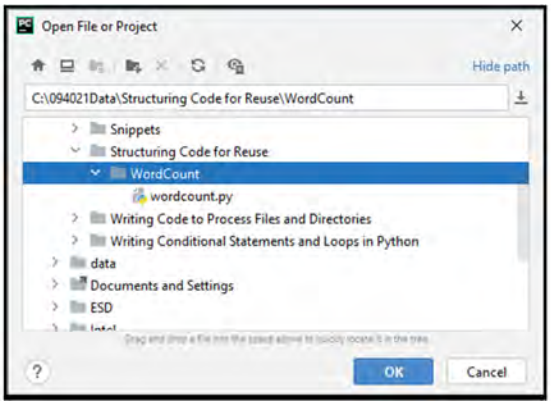
ACTIVITY 5-1
Defining and Calling Functions

Data File
C:\094021Data\Structuring Code for Reuse\WordCount\wordcount.py

Scenario
The WordCount program currently lacks structure. Every line is essentially in the same universal scope, which will make it difficult to reuse important sections of code. The code could also be much easier to read and manage should you or another team member need to expand upon it in the future. So, to add some reusability to your code, you'll wrap some of your most important processes in functions. These functions will help you logically group code blocks together so that you can call upon them when necessary.

---

1. Launch PyCharm and open the WordCount project.
   a) Launch **PyCharm** if you have not left it running from a previous session.
   b) In the Welcome to PyCharm window, select **Open**.
      The **Open File or Project** dialog box is shown.
   c) Beneath the **094021Data** folder, expand **Structuring Code for Reuse**, and select the **WordCount** project folder.



   d) Select **OK**.
      The project is loaded and configured to use the Python 3.9 interpreter.
   e) In the **Project** pane, expand the **WordCount** folder if necessary, and double-click **wordcount.py** to show the source code in the editor.

2. Define the `word_dict()` function that analyzes the input text file.
   a) Place the insertion point on line 12.
   b) Type `def words_dict():`
      This creates a function that takes no arguments. Later, the function will take an argument that passes in the path of the user's input file.

c) Highlight the code in lines 13 through 35.

d) Press **Tab**.

This places all of the code that compares the input file to the wordlist under the function. Now this code is no longer in scope of the whole program, but in scope of the `words_dict` function.

3. Provide a return value.

a) Place the insertion point on line 37, and press **Tab** to indent beneath the `words_dict` function.

b) Type **`return word_count`**

When the function completes, it will now pass the result (contents of the `word_count` dictionary) back to the statement that called the `words_dict()` function.

4. Document the function.

a) Position the insertion point at the end of line 12, and press **Enter**.
b) In line 13, type a docstring that describes what the function does.

```
12  def words_dict():
13      """Compare user input text file to English wordlist and return matches."""
14      read_input = ["This", "is", "a", "test!", "this", "is,", "another", "test.",
15                    "TesT", "value", "this", "!test", "is?"]
16      read_wordlist = ["this", "is", "a", "test"]
```

5. Define the function `print_top_words` that prints the results to the console.
   a) Place the insertion point on line 39 and press **Enter**.
   b) Type **`def print_top_words(choice):`**
      This creates a new function. The function accepts one argument, `choice`, which will identify whether common words should be suppressed.
   c) Indent lines 41 through 54 beneath the function.

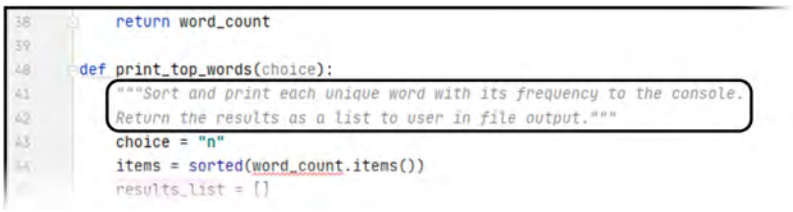
```
36              continue
37
38      return word_count
39
40  def print_top_words(choice):
41      choice = "n"
42      items = sorted(word_count.items())
43      results_list = []
44
45      # Truncates output if user wants to suppress common words.
46      for word in items[:50]:
47          if choice == "y" and word[0] not in COMMON_WORDS:
48              result = word[0] + ": " + str(word[1]) + " times"
49              results_list.append(result)
50              print(result)
51          elif choice == "n":
52              result = word[0] + ": " + str(word[1]) + " times"
53              results_list.append(result)
54              print(result)
```

   As before, this places the block of code under the scope of the newly defined function.
   d) Place the insertion point on line 56, and press **Tab**.
   e) Type `return results_list`

```
53              results_list.append(result)
54              print(result)
55
56      return results_list
57
58
59  while True:
        user_input = input("Please enter the path and name of the text file you want
```

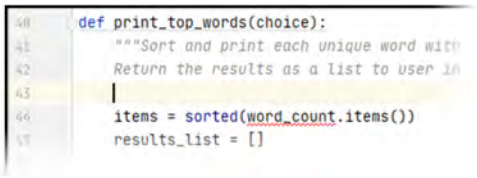
   Now, when you call the `print_top_words()` function elsewhere in your program, it will return `results_list` back to the calling statement.

f) Write a docstring below the function definition as shown:



6. Do some cleanup and call both the `words_dict()` and `print_top_words()` functions.

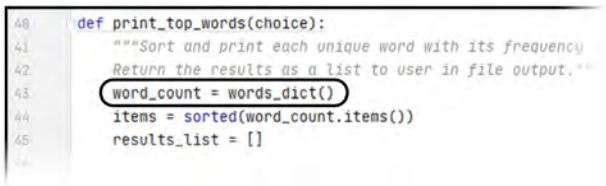
a) In line 43, delete the `choice = "n"` statement, leaving an empty line as shown.



Since you're going to pass in the user's choice as a parameter, you no longer need to define it here.

b) In line 43, type **`word_count = words_dict()`**

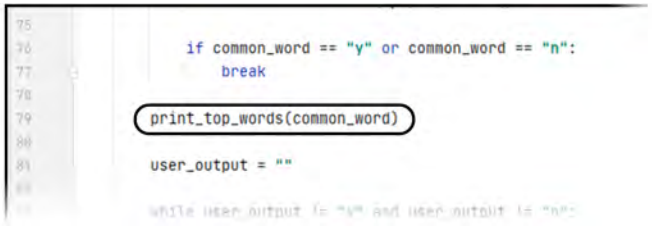
This calls the function and assigns the returned dictionary to `words_count`.



c) Position the insertion point in line 78, and press **Enter**.

d) Press **Tab** twice to indent even with the `user_output` line below.

e) Type `print_top_words(common_word)` and press **Enter**.



This calls the function `print_top_words()` while passing in the user's answer to suppressing words as the parameter.

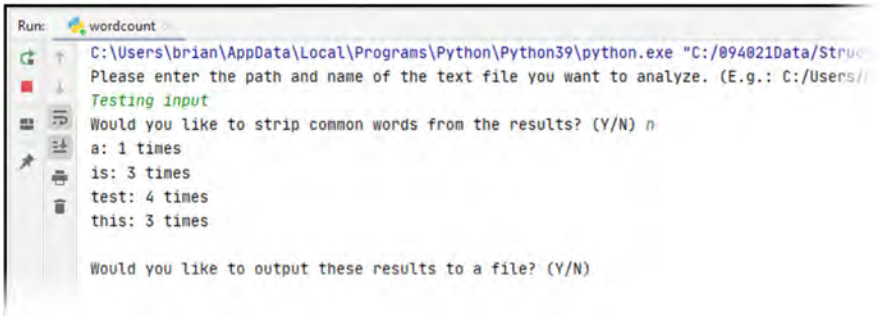
7. Test the program.

a) Run the program.

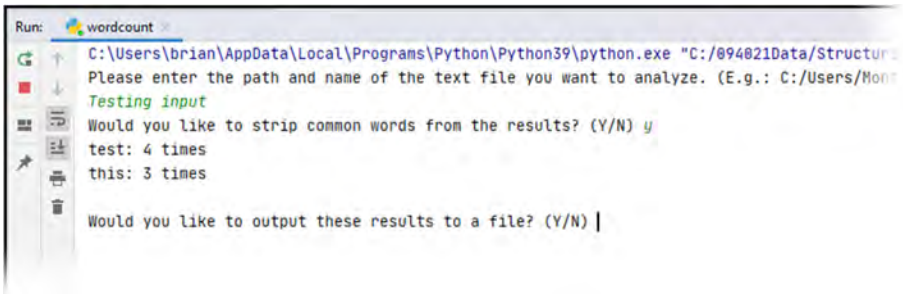
b) Provide any value for the first prompt.

c) At the second prompt, enter **`n`**

d) Verify that the results printed to the console.



e) Rerun the program, this time entering **`y`** at the suppression question. Verify that the results are accurate:



f) In the Run window, close the **wordcount** tab, and select **Terminate** to terminate the process.

TOPIC B

Define and Instantiate a Class

Another object you can leverage in Python is the class, which is yet another tool for organizing and reusing your code. In this topic, you'll define and instantiate classes.

Class Definition

In object-oriented programming, a ***class*** is a way to combine similar code, especially function definitions, into a single container object. A class includes ***attributes***, which are variables within that class's scope. Using a class, you can create objects that share similar attributes, which can improve the efficiency and readability of your code.

The basic form of a class definition is as follows:

```
class ClassName:
```

The `class` statement explicitly defines `ClassName` as the class object, and like other objects, the code underneath should be indented.

Examine the following code block for a more concrete example of a class definition. The specific elements of this class will be discussed later. For now, just know that this block of code creates a class called `Knight` that can process attributes that all knights share, including name, quest, and favorite color.

```
class Knight:
    def __init__(self, name, quest, favorite_color):
        self.name = name
        self.quest = quest
        self.favorite_color = favorite_color

    def display_name(self):
        print("Welcome, Sir {}.".format(self.name))
```

Formatting Class Names

Based on Python's style guide, you should always define variables in CapWords format.

- **Correct**: `class MyClass`
- **Incorrect**: `class my_class`

It's good practice to give your classes meaningful names in order to avoid ambiguity. You also cannot begin class names with a number. Python will produce a syntax error.

Instance Construction

Like functions, classes are callable. How Python calls classes is a little different, though. When you call a class, Python creates an instance of that class. An instance is a certain realization of the class based on a number of arguments that you provide. In the `Knight` class, you can consider each individual knight as an instance. The process of creating instances from classes is called ***instance construction***.

Constructing an instance from a class is similar to calling a function:

```
robin = Knight("Robin", "to seek the Holy Grail", "yellow")
```

This creates the instance `robin` based on the provided arguments.

Notice that the arguments mirror the very first function defined in the class:

```
class Knight:
    def __init__(self, name, quest, favorite_color):
        self.name = name
        self.quest = quest
        self.favorite_color = favorite_color
```

The `__init__()` function is required for initializing (constructing) instances of a class. It can take a number of arguments, the first of which is always `self`. This `self` argument refers to the instance that is being created by the class.

Within this function's code is some variable assignment, using `self.<variable>`. These are called ***instance variables*** because they are defined inside of the function and only apply to the current instance of a class. In this case, `__init__()` is assigning instance variables to each argument. Python can later use these instance variables in other areas of the class.

Methods

Functions inside of classes are actually called ***methods***. More specifically, ***instance methods*** are methods associated with a particular instance of a class. Instance methods are useful when you want to perform specific operations on that instance. The rules for creating an instance method are nearly identical to that of creating functions. The main difference is that instance methods must always take at least one argument, and that required argument must be `self`. Like with the `__init__()` method, `self` refers to the instance being created by the class.

Refer back to the `Knight` class, which has two methods, `__init__()` and `display_name()`:

```
class Knight:
    def __init__(self, name, quest, favorite_color):
        self.name = name
        self.quest = quest
        self.favorite_color = favorite_color

    def display_name(self):
        print("Welcome, Sir {}.".format(self.name))
```

Once the instance variables are initialized in `__init__()`, Python can use these in other methods within the class. For the `display_name` method, Python uses the `self.name` instance variable to, as the name suggests, display the knight's name.

Now that you have an instance method defined in your class, how do you use it? Invoking methods is easy, and in fact, you've already done it. For example, string interpolation using `.format()` at the end of the string is actually invoking the `format()` method that exists in Python's built-in `Formatter` class. For the `Knight` class, you can invoke the `display_name()` method as follows:

```
robin = Knight("Robin", "to seek the Holy Grail", "yellow")

robin.display_name()
```

Notice that you must first construct an instance of the class (`robin` in this case). Then, you can invoke a method on that instance. This results in:

```
Welcome, Sir Robin.
```

Instance Versus Class Methods

Instance methods are not the only type of method. There's also ***class methods***, which are not associated with any instance. Therefore, when you call a class method, you don't need to call it on an instance. To create a method as a class method, add the `@classmethod` decorator on the line before the method definition. A ***decorator*** is simply an object that modifies the definition of a function, method, or class. Also, when you construct a class method, you'd typically use `cls` as its first argument, rather than `self`. For example, the following class `Knight` has a class method `population()`:

```
class Knight:
    count = 0

    def __init__(self):
        Knight.count += 1

    @classmethod
    def population(cls, knights):
        return "There are {} knights.".format(knights)
```

After you initialize an instance of this class, the counter increments by one. The `population()` method returns how many instances you've constructed. Methods like these are useful if you need to execute some process outside of any instances of a class. In this case, it would be pointless to call this method on an individual instance, so `population()` must be a class method.

Assume you constructed four different instances of Knight. To take advantage of the `population()` class method and see how many instances you've constructed, you'd type:

```
>>> Knight.population(Knight.count)
"There are 4 knights."
```

Notice that the call is not attached to any instance—it is merely referencing the class (`Knight`) and the method (`population()`).

Dynamic Class Structure

Instances of objects in Python are dynamic. That is, you can add, change, or delete attributes of an instance at any time. But unlike some other object-oriented programming languages, classes in Python are also dynamic. And it is important to understand that the presence of an instance attribute will essentially hide a class attribute by the same name. The following example illustrates how this works.

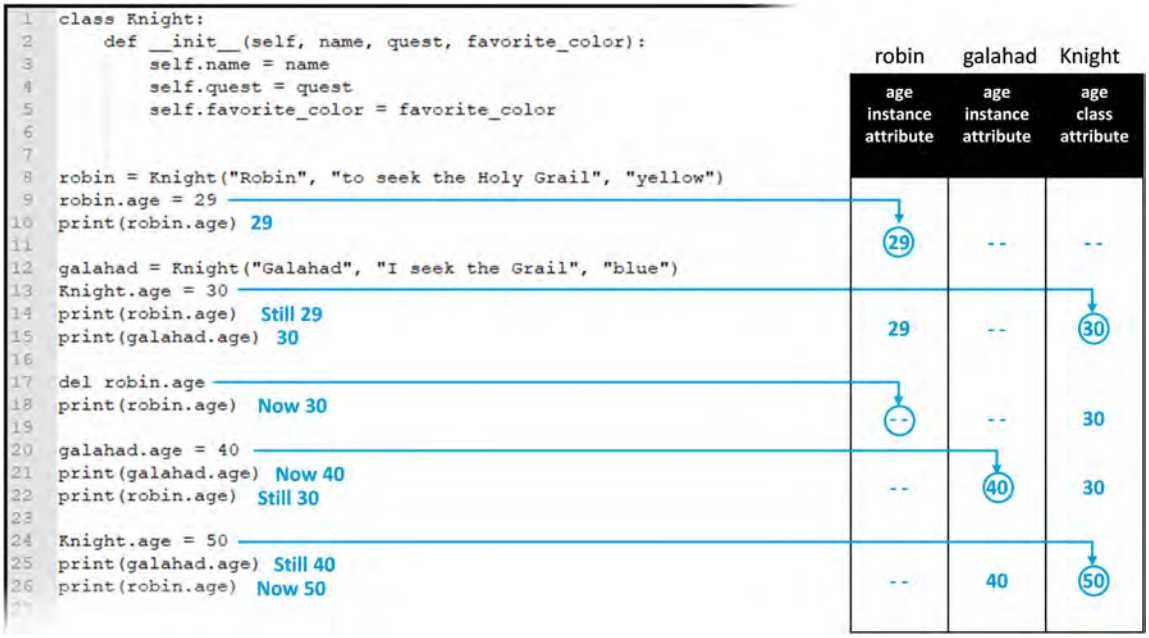


Figure 5-4: Dynamic class structure example.

The `Knight` class doesn't define an `age` attribute, but line 8 assigns an `age` value to the `robin` instance of the `Knight` class. The `age` attribute can now be used with the `robin` instance, and it is successfully printed in line 10. But at this point, only the `robin` instance of `Knight` has an `age` attribute.

In line 12, a new instance of `Knight` is created in a variable named `galahad`. The `galahad` instance doesn't have an `age` attribute, and there is no `age` class attribute in `Knight`. So if you were to print `galahad.age` immediately after line 12, Python would throw an exception. But line 13 assigns a value to `Knight.age`, so Python creates a new class attribute in the `Knight` class. This makes `age` a valid attribute of the `Knight` class, available to any instance of the class. A newly created `Knight` will have a default `age` of 30. But you've already assigned an `age` instance attribute to `robin`, so robin's `age` instance attribute will hide the new `age` class attribute. You can see this in lines 14 and 15. When printed, the `age` of `robin` shows the instance attribute of 29, while the `age` of `galahad` is shown as 30, because galahad has no instance attribute named `age`, and the class attribute is used.

If you delete the instance attribute, as is done in line 17, then the class attribute is no longer hidden, and will be shown when printed. You can use `del` to delete class attributes as well. Using these techniques, you can continue changing both class and instance attributes with predictable results, as demonstrated in lines 20 through 26.

Python's dynamic class structure makes it possible to modify existing classes to fulfill a particular purpose, without having to actually edit that class directly. For example, say you import a module that provides some functionality to your program. You want to expand on this functionality to cater to your own program, but you don't want to change the module itself. Altering attributes dynamically can help you achieve this.

Verifying Attributes

Because classes are dynamic, you may need to verify at some point if a class or instance still has a specific attribute. To do this, use the following syntax:

```
hasattr(instance_name, attribute_name)
```

Using the `Knight` class, this piece of code checks to see whether the `age` attribute still exists:

```
if hasattr(robin, "age") is True:
    print("Attribute exists.")
else:
    print("Attribute does not exist.")
```

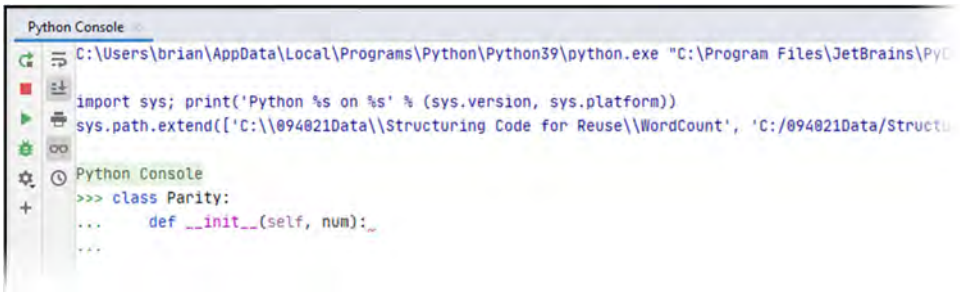
ACTIVITY 5-2
Working with Classes

Before You Begin
Instead of changing your source code, you'll be working with the interactive console in this activity.

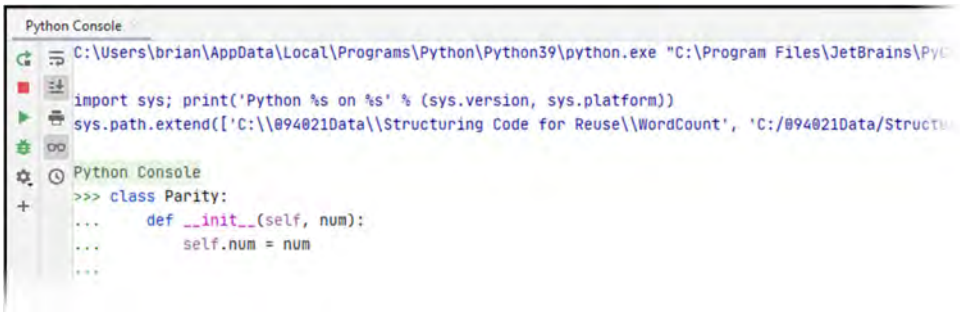
Scenario
Before you implement a class in your WordCount program, you want to make sure you've grasped the concept first. So, you'll use PyCharm's interactive console to create a test class and construct an instance of that class. You'll also dynamically change the attributes of that class.

---

1. Create the `Parity` class and its initialization method.
   a) From the **PyCharm** menu, select **Tools→Python or Debug Console**.
   b) In the console, type **`class Parity:`** and press **Enter**.
      Parity is the concept of numbers being odd or even. Your test class will determine if the number you provide is either odd or even.
   c) Verify that the next prompt begins indented. Type **`def __init__(self, num):`** and press **Enter**.



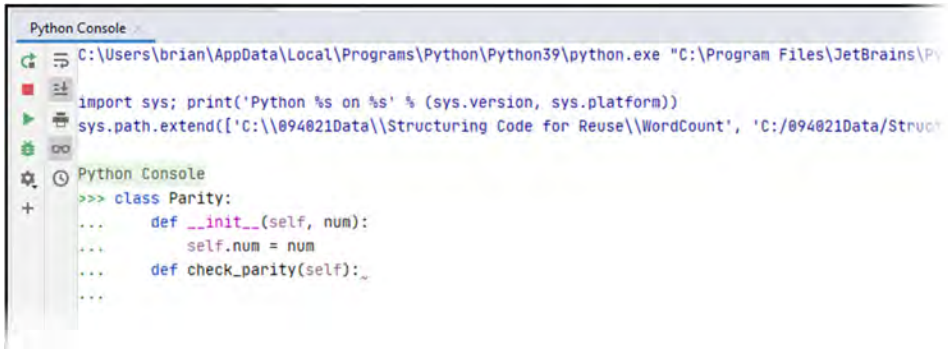
   d) Your initialization method takes `self` because it is an instance variable. This variable will be applied to all instances of the class. The other argument, `num`, is the number that you'll provide when you construct an instance of the class in order to check its parity.
   e) Type **`self.num = num`** and press **Enter**.



   This assigns the instance variable `self.num` to the number you're passing in. Now, in other methods that could go in this same class, you can use `self.num` to refer to each instance's `num` argument.

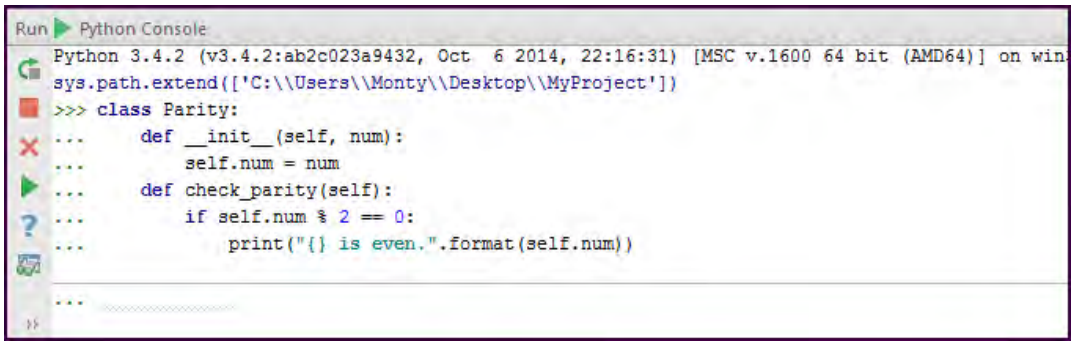
2. Create the `check_parity()` method to check if the number is odd or even.

a) Press **Shift+Tab** to go back one level of indentation.
   The insertion point should now be on the same level as the `def` above.
b) Type **`def check_parity(self):`** and press **Enter**.



This method takes the instance variable `self` as its only argument. You'll only need to use an instance variable in this method, so this is the only argument it needs.

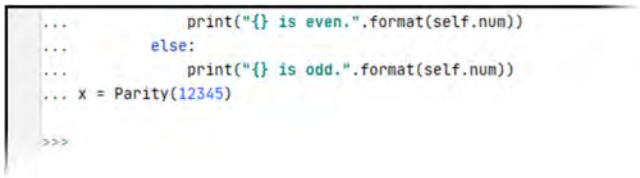
c) Type **`if self.num % 2 == 0:`** and press **Enter**.
   This checks if the instance's number is evenly divisible by two.
d) Type **`print("{} is even.".format(self.num))`** and press **Enter**.



e) Press **Shift+Tab** to go back one level of indentation.
f) Type **`else:`** and press **Enter**.
g) Type **`print("{} is odd.".format(self.num))`** and press **Enter**.

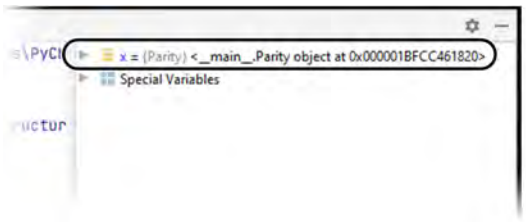
3. Construct an instance of the `Parity` class.

a) Press **Shift +Tab** three times so that you're back at the highest level of indentation and no longer inside the class.
b) Type **`x = Parity(12345)`** and press **Enter**.



You're constructing an instance of class `Parity` as `x` and passing in `12345` as the `num` argument.

c) Observe the variable list.



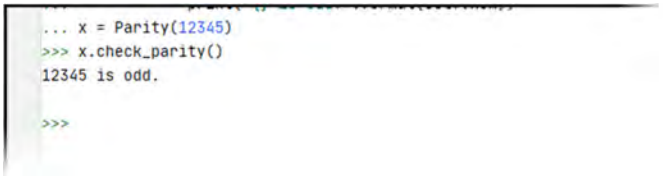
An object named x has been created.

d) Select the arrow to the left of the x variable to expand the list.

When you created an instance of the object named x, you initialized the object with a `num` value of 12345.

e) In the Python Console, type **`x.check_parity()`** and press **Enter**.

f) Verify that the console tells you the number contained within the object is odd.

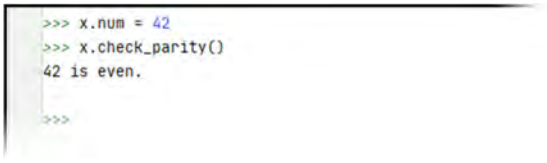


4. Adjust your instance's number value dynamically.

a) At the prompt, type **`x.num = 42`** and press **Enter**.

b) Type **`x.check_parity()`** and press **Enter**.

c) Verify that the `num` value changed and that the result is now even.



d) Close the Python Console.

Class Dictionaries

Elements inside of a class can be represented as dictionaries. These dictionaries list the attributes of a class or its instance. Assume that you still have the `robin` instance of the `Knight` class. You'll then add a few new attributes to the instance, as follows:

```
robin.age = 29
robin.crest = "bird"
robin.honorific = "the Brave"
```

To retrieve a dictionary, you use Python's built-in `__dict__` attribute that all classes have:

```
robin.__dict__
```

Printing this dictionary will display the following:

```
{'crest': 'bird', 'quest': 'to seek the Holy Grail', 'name': 'Robin',
'favorite_color': 'yellow', 'age': 29, 'honorific': 'the Brave'}
```

As you can see, every key in this dictionary is an attribute (including the ones you added specifically for the instance) and has a corresponding value. Creating dictionaries from classes and instances is useful for processing specific attribute information in a contained format.

You can also use dictionaries to add, modify, and delete attribute values in an instance or class. In the following example, the `honorific` attribute is being changed in the instance's dictionary:

```
robin.__dict__["honorific"] = "the Not-So-Brave"
```

This modifies the key called "honorific" to a new value. When it's printed:

```
print(robin.name + " " + robin.honorific)
Robin the Not-So-Brave
```

Note: If you add an attribute to an entire class dynamically, this attribute will not be in an instance's dictionary. You must assign the attribute to that particular instance.

Properties

A ***property*** is an attribute with getter, setter, and delete methods. These types of methods retrieve an attribute, modify an attribute, and delete an attribute, respectively. You'd typically use a property to streamline this behavior without needing to explicitly invoke each getter, setter, and delete method for an instance. For example, this is an example of the `Knight` class without a property:

```
class Knight:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_name(self, value):
        self.name = value

    def del_name(self):
        del self.name
```

Assume you constructed an instance of `Knight` called `arthur`:

```
arthur = Knight("Arthur")
```

Now, here's how you would normally invoke each method:

```
arthur.get_name
arthur.set_name = "King Arthur"
arthur.del_name
```

By using a property, you can actually do this more efficiently. Take a look at the following code, which uses a property:

```
class Knight:
    def __init__(self, name):
        self._name = name

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value

    def del_name(self):
        del self._name
```

```
    name = property(get_name, set_name, del_name)
```

Note the two major differences: the `property()` function at the bottom and the insertion of _ before the `name` variable. Using this underscore before the variable provides a hint to the interpreter and to the programmer that the variable is meant to be private. A ***private variable*** cannot be used outside of the class and is a requirement for using a property. Second, assigning the `name` variable to the `property` of each method allows you to bypass invoking these functions when you want to use them. Compare the following code to the three invocations above:

```
arthur.name
arthur.name = "King Arthur"
del arthur.name
```

As you can see, you no longer need to invoke a method to either get, set, or delete an attribute. You just need to reference the attribute and perform the operation like you would with any other variable. Python's `property()` function automatically invokes the relevant method in the class. This is particularly useful if you change method names; with a property, you won't have to change every invocation of these methods in your program. Using properties also makes for cleaner, easier-to-read code.

Property Decorator

Python has an object called a ***decorator*** that you can use to modify the definition of a function, method, or class. Rather than using the `property()` function as above, you can alternatively implement a property with the `@property` decorator. The following code is equivalent to the class definition above:

```
class Knight:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @name.deleter
    def name(self):
        del self._name
```

Notice that the class reuses `name` for the method definitions. The `@property` decorator initiates the property, and each decorator below that is adding the proceeding method to the property. You can get, set, and delete an attribute just like before, without explicitly invoking the method:

```
arthur.name
arthur.name = "King Arthur"
del arthur.name
```

Inheritance

Inheritance is the process by which a class extends the capabilities of other classes. So, a class that inherits another class is using that other class's code to its advantage. Extending class functionality through inheritance is yet another way to optimize your code for reuse and to maximize your efficiency as a programmer. For example, if a class already exists that provides half of the functionality you need for your new class, you can just inherit it rather than writing a bunch of superfluous code.

The basic syntax for defining a class that inherits from other classes is:

```
class SubClass(SuperClass1, SuperClass2, SuperClass3):
```

In inheritance, the term ***superclass*** refers to any class that is *being inherited*. Likewise, a ***subclass*** is a class that *does the inheriting*. As you can see, a subclass can inherit multiple superclasses in Python.

Note: The subclass/superclass relationship is also referred to as a child/parent relationship.

When your class inherits a superclass, it is able to use all of the members of that superclass. Consider the following two class definitions. The first is a superclass (`Citizen`), and the second is a subclass (`Knight`) that inherits the first:

```
class Citizen:
    def __init__(self, name, occupation, birthplace):
        self.name = name
        self.occupation = occupation
        self.birthplace = birthplace

    def display_info(self):
        print("{}, "
              "the {}, "
              "is from {}.".format(self.name,
                                   self.occupation,
                                   self.birthplace))

class Knight(Citizen):
    def knight_quest(self):
        print("{}, you must seek Camelot.".format(self.name))
```

The `Knight` subclass is inheriting `Citizen` because a knight is a type of citizen. This "is-a" relationship is a common use case for inheritance.

Now, construct an instance of `Knight`:

```
arthur = Knight("Arthur", "King", "Great Britain")
```

This instance is constructed with arguments that are defined in the superclass, despite using the subclass. Try and guess what the following code will do with the `arthur` instance:

```
arthur.display_info()
arthur.knight_quest()
```

Each method invoked is from a different class, yet both work as intended:

```
Arthur, the King, is from Great Britain.
Arthur, you must seek Camelot.
```

This is the power of inheritance.

Note: A method in a subclass will override a method in the superclass if it has the same name. However, you can still directly reference the superclass in order to use its method: `SuperClass.method(instance)`

Checking Class Relationships

When inheritance gets complex, it can be helpful to know the direction of the subclass-superclass relationship. You can check this with the `issubclass(SubClass, SuperClass)` function. If the first argument is indeed a subclass of the second argument, the result returns true. Using the classes above:

```
issubclass(Knight, Citizen)
```

This will return true because `Knight` is a subclass of `Citizen`.

Likewise, you can check if a certain object is an instance of a class or subclass using `isinstance(instance, Class)`. If the first argument is indeed an instance of the class or subclass in the second argument, it will return true:

```
isinstance(arthur, Citizen)
```

Special Methods

Special methods are methods that are reserved by Python and perform certain tasks within a class. You've already seen the special method `__init__()`, but there are more. All special methods have two leading and trailing underscores. The following table describes some special methods used by Python.

Special Method	*Description*
`__init__()`	Use to initialize an instance.
`__del__()`	Use to destroy an instance. Opposite effect of `__init__()`
`__setattr__()`	Use to override the attributes of a class.
`__getattr__()`	Use to retrieve the attributes of a class.
`__delattr__()`	Use to delete the attributes of a class.
`__str__()`	Use to print returned values as neatly formatted strings.
`__int__()`	Use to print returned values as integers.
`__float__()`	Use to print returned values as floats.

Note: This is not an exhaustive list. For a complete list of special methods, navigate to **https://docs.python.org/3/reference/datamodel.html**.

Operator Overloading

You can also use certain special methods to perform a process called ***operator overloading***. This allows your class to define how it handles operators. Failing to overload operators may result in errors. For example, consider the following code:

```
class Addition:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "The answer is {}.".format(self.a)

operator1 = Addition(3)
operator2 = Addition(5)
print(operator1 + operator2)
```

This code attempts to add the values of two instances together, but will produce an error. To actually perform the addition operation, you can use the `__add__()` special method to overload the operator:

```
class Addition:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "The answer is {}.".format(self.a)
```

```
    def __add__(self, other):
        return Addition(self.a + other.a)

operator1 = Addition(3)
operator2 = Addition(5)
print(operator1 + operator2)
```

This outputs `8` because the `__add__()` method is defining how it handles the `+` operator when an instance uses it. Technically, you could change the `+` operator in the `__add__()` method to whatever you wanted. For example:

```
    def __add__(self, other):
        return Addition(self.a - other.a)

operator1 = Addition(3)
operator2 = Addition(5)
print(operator1 + operator2)
```

In the last line, even though you've used the `+` operator, the `__add__()` method in the class treats it like a subtraction operation, and the output will be `-2`.

Special Methods

The following table lists various operator overloading special methods:

Special Method	*Equivalent Expression*
`__add__()`	`a + b`
`__sub__()`	`a - b`
`__mul__()`	`a * b`
`__truediv__()`	`a / b`
`__floordiv__()`	`a // b`
`__mod__()`	`a % b`
`__pow__()`	`a ** b`
`__and__()`	`a & b`
`__or__()`	`a | b`
`__eq__()`	`a == b`
`__ne__()`	`a != b`
`__gt__()`	`a > b`
`__lt__()`	`a < b`
`__ge__()`	`a >= b`
`__le__()`	`a <= b`
`__contains__()`	`a in b`
	`a not in b`

Class Scope

Like with functions, classes in Python can be thought of in terms of scope. Besides instance variables, Python also has class variables. A ***class variable*** can be shared by all instances of a class.

They are defined outside of any methods within the class to differentiate them from instance variables. For example, say you want to use the `Citizen` class to keep a running count of all constructed instances of the class. You can use a class variable to ensure that all instances share a common count to increment:

```
class Citizen:
    citizen_count = 0

    def __init__(self, name, occupation, birthplace):
        self.name = name
        self.occupation = occupation
        self.birthplace = birthplace
        Citizen.citizen_count += 1
```

The `citizen_count` variable is the class variable. In the initialization method, each time an instance is initialized, `Citizen.citizen_count` increments by one. The `Citizen.` reference at the beginning of the assignment points directly to the class variable. Just using `citizen_count += 1` would force Python to treat this variable as local to the method; that is, it would be within the wrong scope.

Using the correct code above, the following will result in a count of `4`:

```
>>> arthur = Citizen("Arthur", "King", "Great Britain")
>>> lancelot = Citizen("Lancelot", "Knight", "Great Britain")
>>> bedevere = Citizen("Bedevere", "Knight", "Great Britain")
>>> galahad = Citizen("Galahad", "Knight", "Great Britain")
>>> print("There are {} citizens of the realm.".format(Citizen.citizen_count))
There are 4 citizens of the realm.
```

Be mindful of the scope of your classes' elements, just as you would when only working with functions.

Guidelines for Defining and Using Classes

Use the following guidelines to help you work with classes in your Python programs.

Defining and Using Classes

To work with classes in your Python programs:

- Use classes to streamline code that draws from similar characteristics.
- Think of classes as templates for creating new objects.
- Define a class with the `class` statement and append a colon (`:`) to the end of the line.
- Indent the code in the class, much the same as functions.
- Construct an instance of a class and provide arguments in the format `instance = Class(args)`.
- Use the `__init__()` method to initialize code for each instance, such as instance variables.
- Define functions inside of classes to create methods which an instance can use.
- Use a method with an instance in the following format: `instance.method()`.
- Consider that you can modify the attributes of a class at any time, including adding and deleting attributes.
- Create dictionaries from instances and classes to work with the data they contain in a more structured format.
- Use properties to avoid having to explicitly call methods from a class. Use either:
  - The `property()` function.
  - The `@property` decorator.
- Optimize your code and minimize the time you spend writing it by leveraging class inheritance.
- Use inheritance when working with an "is-a" relationship; for example, a square is a rectangle.

- Keep track of the relationships between superclasses and subclasses by using the `issubclass` function.
- Take advantage of Python's built-in special methods, like `__del__()` and `__str__()`.
- Use operator overloading to perform calculations on class attributes.
- Use class variables for sharing values across all instances of a class.
- Keep the scope of your variables in mind when using classes to prevent conflicts and other related errors.

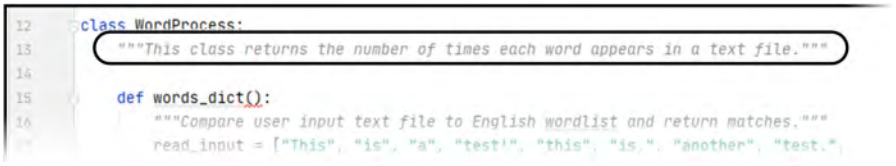
ACTIVITY 5-3
Defining and Instantiating a Class

Before You Begin
You'll be returning to your source code in this activity.

Scenario
Restructuring your processes into functions has given you a good deal of flexibility with your source code. However, there's value in structuring your code even further. Your program will eventually be integrated with other programs in the project, and your own source code may grow considerably in size down the road. To accommodate this, you'll implement a class in your code to group related functions together. You'll also construct an instance of that class that will take a user's text file input and execute the program from there. Lastly, you'll clean up your code so that its references are all in scope, and you'll also ensure that your results are sorting properly.

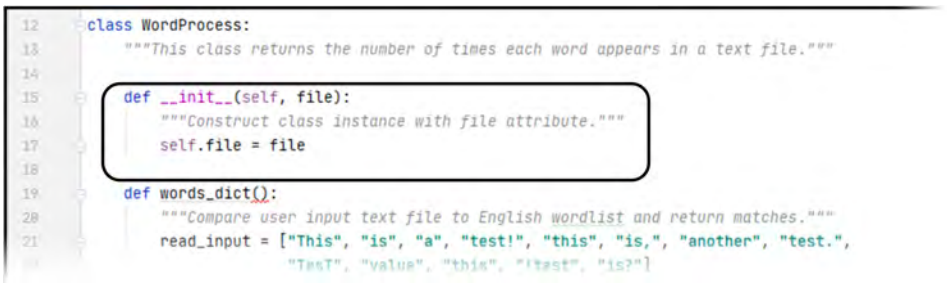
---

1. Wrap the `words_dict()` and `print_top_words()` functions in a class called **`WordProcess`**
   a) Place the insertion point on line 10 and press **Enter** twice.
      There should be two blank lines between your insertion point and the end of the `COMMON_WORDS` definition.
   b) Type **`class WordProcess:`**
   c) Highlight the code from lines 14 through 60 and press **Tab**.
      Your two functions are now within the scope of the overall `WordProcess` class. In other words, they are now methods of the `WordProcess` class.
   d) Position the insertion point at the end of line 12, and press **Enter**.
   e) Type the following docstring with a blank line after it:



2. Add a constructor to your new class.
   a) Position the insertion point in line 14. Press **Enter** and **Tab**.

b) Type the following code, including the empty line at the end.

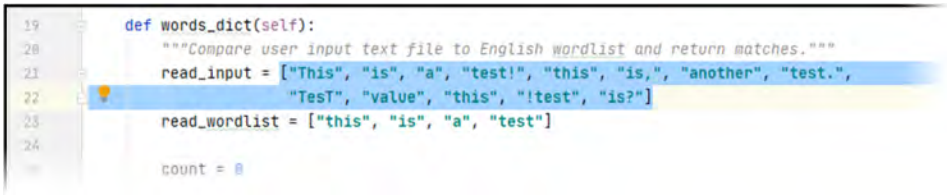


This constructor takes `self`, as all constructors do, and it also takes `file` as a parameter. This will be the file path that the user provides, and in this constructor, you're assigning an instance attribute to this parameter.

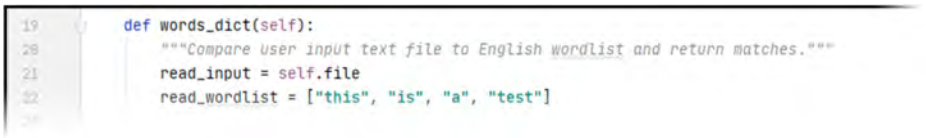
c) On line 19, in the `words_dict()` method, add **`self`** as a parameter.

You'll be using the reference to the class instance in order to invoke the file path that the user provides.

d) Select the list expression in lines 21 and 22 as shown.



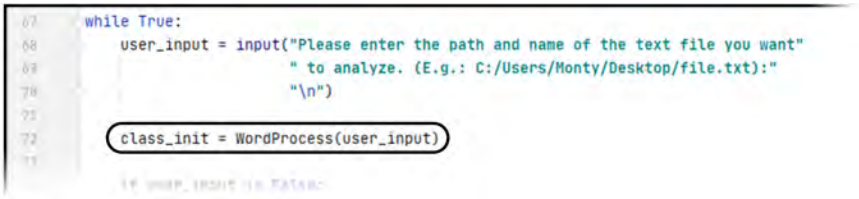
e) Replace the selected code by typing **`self.file`**



3. Construct an instance of the class with your user's file path input.

a) Position the insertion point at the end of line 70 and press **Enter** twice.

b) Type the following:



This constructs an instance of `WordProcess` with the user's input as the `file` parameter. The instance is assigned to the variable `class_init`.

4. Change function calls to class method references and perform additional cleanup.

a) In line 87, change the `print_top_words()` function call to the following:

```
            if common_word == "y" or common_word == "n":
                break

        new_result = WordProcess.print_top_words(class_init, common_word)

        user_output = ""
```

In this line, you're calling the `print_top_words()` method by first referencing the `WordProcess` class. This prevents issues arising with scope. When you call this method, you're passing in two parameters: the class instance you constructed previously (which contains the user's file path) and the `common_word` answer to the question of suppressing common words. This all ends in the `print_top_words()` method executing its code and printing the results to the console.

b) Scroll up to line 46, where the `print_top_words()` method is defined.

c) Add **`self`** as the *first* parameter, keeping `choice` as the second.

```
    def print_top_words(self, choice):
        """Sort and print each unique word with its frequency to the console.
        Return the results as a list to user in file output."""
        word_count = words_dict()
```

Now, `choice` will take on the `common_word` answer, and `self` will, as usual, refer to the class instance.

d) In line 49, change the function call to a method call.

```
    def print_top_words(self, choice):
        """Sort and print each unique word with its frequency to the console.
        Return the results as a list to user in file output."""
        word_count = self.words_dict()
        items = sorted(word_count.items())
        results_list = []
```

The `word_count` variable will now hold the dictionary returned by the `words_dict()` method, while using the class instance.

5. Add a class method that creates a key to be used in sorting results.

a) Position the insertion point in line 18, and press **Enter** and **Tab**.

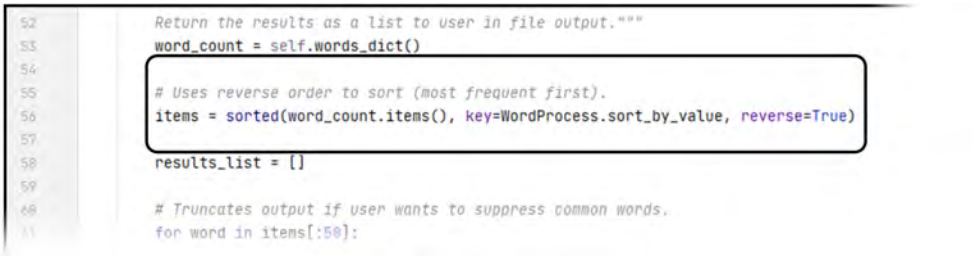
b) Type the following code, including the empty line at the end.

```
    def __init__(self, file):
        """Construct class instance with file attribute."""
        self.file = file

    def sort_by_value(item):
        """Create a key to be used to sort wordlist later."""
        return item[-1]

    def words_dict(self):
        """Compare user input text file to English wordlist and return matches."""
        read_input = self.file
        read_wordlist = ["this", "is", "a", "test"]
```

This is a class method and not associated with any instance. It takes `item` as a parameter and returns index `-1` of `item`. The `-1` index tells Python to start from the end of the list rather than the beginning.

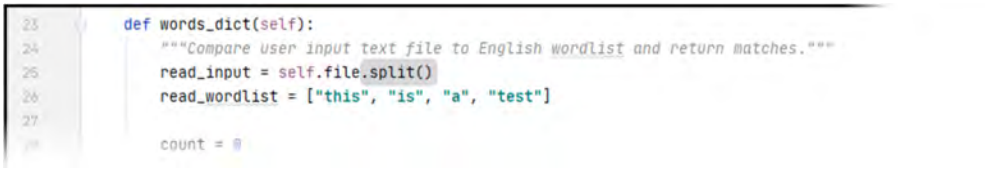
c) Scroll down to the `print_top_words()` method. On line 54, locate the `items` variable definition.

d) Add a comment, revise the statement, and add blank lines before and after it as shown.



The `sorted()` function also takes two additional arguments, a `key` and a `reverse` value. The key references the `sort_by_value()` class method you just created and tells Python to sort each tuple in the list by its last item. Recall that each tuple holds two values: the dictionary key and its associated value. The value comes second, and is technically last (`[-1]`), so Python is sorting by value. Since the value is the frequency that a word appears, Python is sorting by frequency. Lastly, the `reverse` argument simply tells Python to output the results in descending order (most frequent word first).

e) On line 25, add `.split()` to the end of the `read_input` definition so that your string input can be split into a list.



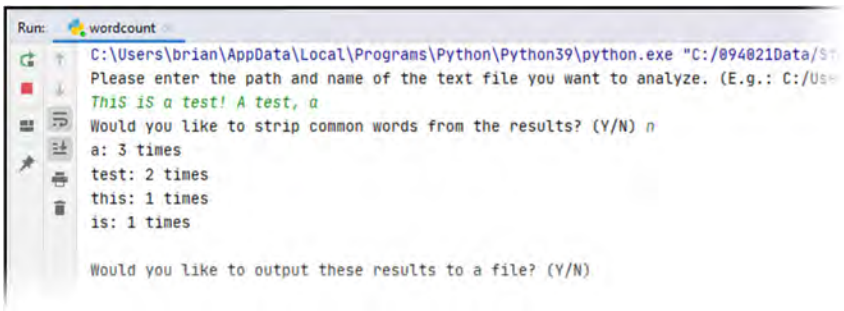
6. Test out your program to confirm that your class code works.

a) Run the program.

b) At the first prompt, type ***ThiS iS a test! A test, a***

c) Answer "**y**" or "**n**" to suppressing common words, whichever you prefer.

d) Verify that the results are successfully printed to the console.



e) Run the program again, this time verifying the opposite choice for suppressing words.

TOPIC C

Import and Use a Module

The last object that will help you put existing code to use in your applications is a module. Relying on other Python modules, you'll be able to add a great amount of functionality to your application.

Modules

In Python, a ***module*** is essentially any file that contains Python code. This file can include any of the objects you've used so far, including variables, functions, classes, loops, branches, and more. The module may also be executable on its own. Technically, the files you've been creating thus far are modules, and each module has a name. Other than being programs themselves, modules are useful because they are an even more high-level way to structure code than a class. A module is a resource that you can take advantage of in your own code to easily perform tasks that would otherwise require you to write a ton of code yourself.

Consider what you would do if you wanted to introduce a random number into your program. Creating a complex random number generator yourself is likely not feasible. This is where modules come in handy. Python has a `random` module that you can use to quickly and easily generate a random number in your program. For example, you can assign a random integer to a variable:

```
my_random_int = random.randint(1, 10)
```

In this case, the `random` module's `randint()` method is being called, with the arguments `1` and `10` indicating the bounds (inclusive). In other words, at runtime, `my_random_int` will be assigned a random number between 1 and 10.

Modules like this one help make you a more efficient programmer. Rather than starting from the ground-up, you can, as the name implies, work in a modular fashion and build upon the many tools that have been freely provided to you.

Note: Python modules are not in any special format—most of them are just Python code.

Import

In order to take advantage of a module, you must import it into your program's source code. There are three general ways to do this, the first of which is called a ***general import***. In a general import, you're importing every single object that's available in the module for you to use. The syntax for a general import is as follows:

```
import module
```

You should place this `import` statement at the beginning of your source code so that it executes first.

Say you want to take input from a user in which they guess a number between 1 and 10. You want the actual number to be random each time the program runs to add an element of unpredictability. If the user guesses correctly, you tell them they've guessed correctly. If the guess was incorrect, you tell them they guessed incorrectly, and you provide the correct answer. The following code does just that, while importing the `random` module:

```
import random

random_num = random.randint(1, 10)
user_guess = int(input("Guess a number between 1 and 10: "))

def result(random_num, user_guess):
```

```
    if user_guess == random_num:
        print("Good guess! {} was the correct number.".format(random_num))
    else:
        print("Sorry, the correct number was {}.".format(random_num))

result(random_num, user_guess)
```

The very first line of this code includes the `import` statement. The next line of code references the name of the module (`random`) and calls a function (`randint()`) from this module, providing the arguments. This is all assigned to a variable (`random_num`) in the current program. With a simple `import` statement, you can use any of this module's objects.

From … Import

The second type of import is a ***selective import***. In a selective import, you specify the exact objects that you need from a module, and nothing else. When you do this, only the objects you specify are available to you. The advantage of a selective import is that you don't need to continually reference the module itself when calling objects. So, instead of calling `random.randint()` every time, you'd be able to just write `randint()`. The syntax for a selective import is as follows:

```
from module import object
```

As with a general import, you would typically place this statement at the beginning of your program.

So, in the guessing game example, you'd replace the general import with a selective import as follows:

```
from random import randint

random_num = randint(1, 10)
```

Universal Import

The last type of import is a ***universal import***. This combines the inclusiveness of a general import, with the expediency of a selective import. In other words, you can import every object in a module, while not being required to type out that module every time you call one of its objects. The syntax for a universal import is as follows:

```
from module import *
```

In a universal import, the asterisk (`*`) acts as a wildcard character telling Python to import every object from the module. So, using the number guessing example:

```
from random import *

random_num = randint(1, 10)
```

While they may seem to be the best of both worlds, there is a pitfall to using universal imports. If you define any objects of your own that have the same name as the objects from the imported module, it will cause ambiguity. Your code may end up failing to work as intended. This is a lot easier to control when doing a selective import because you know exactly which objects you're importing and can easily keep track of their names. With a general import, referencing the module in every object call avoids the conflict. However, unless you know the names of every object in an imported module, a universal import could cause issues. This is especially true of large, complex modules.

Modules Bundled with Python

One of the major strengths of Python is that it has a large number of modules already built into the language. All of these modules together make up Python's ***standard library***. Many of the modules in this library are actually readily available to you as **.py** files in the Python directory. For example, you can find the `random` module in Python 3.9 by navigating your file manager to **/Python39/**

Lib/. The **random.py** file is in this folder, and you can open it up in a code editor just like any other Python file.

Note: Some modules, like `math`, are built into the interpreter and do not exist as discrete **.py** files.

The following table lists only some of the major modules available in Python's standard library.

Standard Module/Library	*Description*
`datetime`	Provides classes for working with dates and times.
`time`	Allows you to work with ***Unix time*** values.
`calendar`	Allows you to output calendars.
`math`	Provides advanced mathematical functions, like square root and logarithms.
`random`	Allows you to generate pseudorandom numbers.
`re`	Allows you to perform regular expression operations.
`csv`	Streamlines the process of reading from and writing to comma-separated values (CSV) files.
`os`	Allows you to access various operating system-level functionality, including the computer's file system.
`tkinter`	Provides tools for GUI programming.
`sys`	Provides access to resources used by the Python interpreter.
`socket`	Provides a low-level networking interface.
`collections`	Provides additional data structures beyond the standard dictionary, set, list, tuple, and range types.
`json`	Provides encoding and decoding functionality for JavaScript Object Notation (JSON).
`shutil`	Allows you to perform high-level file operations like copying and moving.
`urllib`	Provides modules for processing web URLs.
`logging`	Provides event logging functionality for programs.
`itertools`	Provides additional tools for iterative operations.
`functools`	Provides tools for working with higher-order functions.
`unittest`	Provides tools for creating and running tests on your code.
`argparse`	Provides tools for parsing command line arguments.

Note: For a complete list of all resources that make up the Python standard library, navigate to **https://docs.python.org/3/library/**.

External Libraries and Modules

Even though Python comes with a sizeable standard library, there are numerous custom modules available that can extend the language even further. These custom modules are written by many different individuals and many different organizations, each one tailored to fill a certain need. There are various websites that compile lists of custom modules that may be useful to you, including **https://wiki.python.org/moin/UsefulModules**.

Note: One of the key differences between Python 3 and Python 2 is that some of the custom libraries and modules available for Python 2 are not yet compatible with Python 3.

Guidelines for Importing and Using Modules

Use the following guidelines to help you work with modules in your Python programs.

Importing and Using Modules

To work with modules in your Python programs:

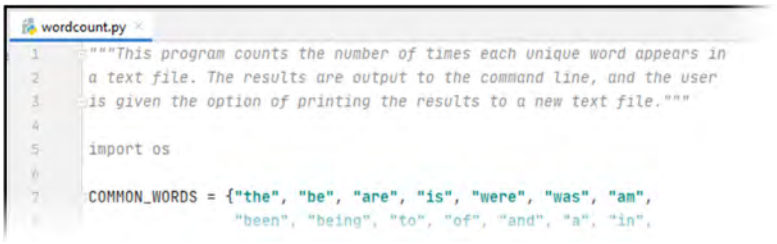
- Use modules to streamline your code and make the task of programming more efficient.
- Consult Python's standard library for all of the extended functionality it offers.
- Use a general import to leverage the full power of a module (`import module`).
- Use `from module import object` to perform a selective import.
- Use selective imports to only import the objects you need, and to avoid having to reference the module for each object call.
- Avoid using universal imports to prevent conflicts and ambiguity that may arise.
- Place `import` statements at the beginning of your source code.
- Research custom modules and libraries that might help you write your Python programs.

ACTIVITY 5-4
Importing and Using a Module

Scenario
Most programs you'll write will be supported by existing software, and the word count app is no different. One common module that's included with Python is the `os` module, which you'll need to do many different file and folder management tasks. So, you'll import this module and test one of its methods to see how it can integrate with and extend the capabilities of your program.

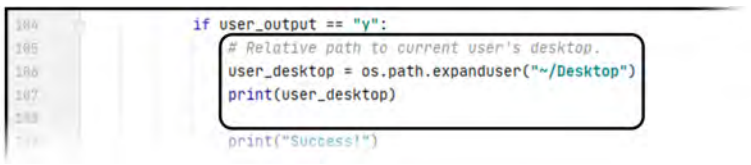
---

1. Import the `os` module.
   a) Place your insertion point on line 4, and press **Enter**.
   b) Type **`import os`** and press **Enter**.



   c) Verify that PyCharm has turned the text grey.
   This is because, at the moment, you haven't used this module in your code.

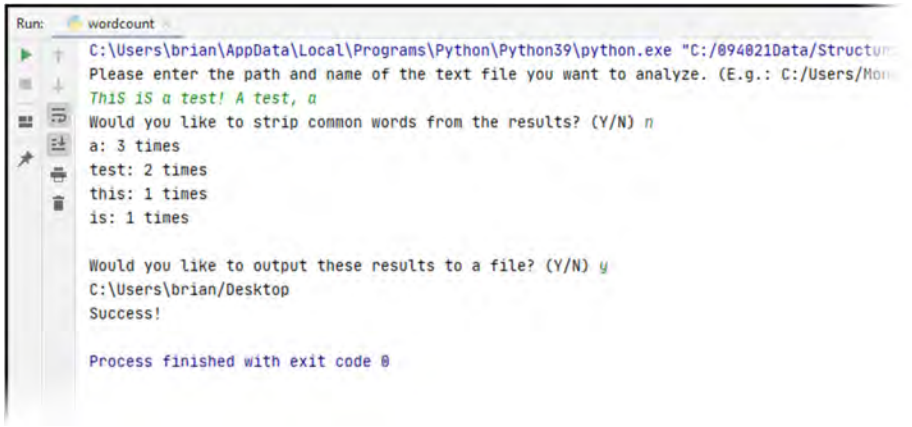
2. Test out the module by calling one of its methods.
   a) Place the insertion point at the end of line 104, and press **Enter**.
   b) Type the following code, including the empty line at the end.



   Line 106 references the `expanduser()` method from the `os.path` module, which extends the `os` parent module. The `expanduser()` method uses ~ to return a string value of the current user's home directory. This prevents you from needing to reference the path directly, and it can also work on Linux and Mac OS X environments. The rest of the argument adds the user's desktop to the directory path. The print statement in line 107 is temporary for testing purposes.

3. Verify that Python has referenced your desktop correctly.
   a) Run the program.
   b) Type in any test values as input, but say yes to outputting the results to a file.

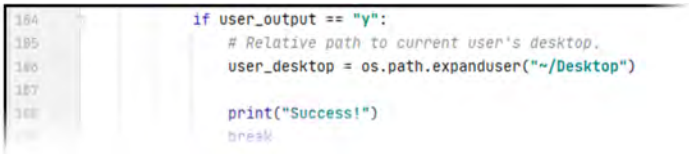
c) Verify that Python printed the correct path to your desktop directory.



Note: By default, Windows uses backslashes in directory names, and Python follows this convention. However, Windows can also handle forward slashes, which is why this mix of both slash types is acceptable.

d) In the Run window, close the **wordcount** tab.

e) Delete line 107 (the print statement) from your code.



4. **In line 106 where you call `expanduser()`, why do you need to include `os.path` before it?**

5. **Which of the following statements would allow you to selectively import `expanduser()` so you don't need to reference the `os.path` module?**

   - ○ import expanduser from os.path
   - ○ import os.path with expanduser
   - ○ from os.path import expanduser
   - ○ for os.path import *

6. Clean up the workspace.

   a) Select **File→Close Project**.

   b) Close the Welcome to PyCharm window.

Summary

In this lesson, you made your code more amenable to reuse through functions, classes, and modules. Structuring your code around these objects will make your code easier to manage and easier to incorporate elsewhere. This saves you and your colleagues time and keeps your code lean and optimized.

What examples can you think of that demonstrate when to use a class versus just defining stray functions?

What kind of external modules will you search for to extend your program's capabilities?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

6 Writing Code to Process Files and Directories

Lesson Time: 2 hours, 20 minutes

Lesson Introduction

One of the most common functions in any program is the ability to access an operating system's file structure. This can give your application the power to use, process, and create external data as necessary. In Python®, these tools are universal and easy to implement across a wide variety of operating environments.

Lesson Objectives

In this lesson, you will:

- Write text to files.
- Read from text files.
- Get the contents of a directory.
- Manage files and directories.

TOPIC A

Write to a Text File

You'll begin this lesson by learning how Python accesses files, and then you'll write data to a file using simple Python statements and functions.

File Objects

Reading and writing data to a file requires that you first open that file in Python. When you open a file, you assign it to a file object that can take several arguments. Python has a built-in function called `open()` that you'll use to create these file objects. The syntax for creating file objects is as follows:

```
file_object = open("file name", "access mode")
```

The two required arguments in the `open()` function are as follows:

- "File name" is a string value of the file that you're accessing.
- "Access mode" refers to *how* that file is opened, and what sort of operations can be performed on it.

So, assume you have a text file named **names.txt** that you want to write to. To open this file in Python, you'd write:

```
names_file = open("names.txt", "w")
```

The first argument provides the name of the file, and the second argument tells Python that you want to open the file in the "write" access mode. You can now process the file object to read, write, and manage the file in many other ways.

However, you're not done with your file until you explicitly close it in the source code. Closing a file in Python is necessary because Python keeps data in a temporary memory buffer before it actually writes the data to the file. So, as long as your file stays open, Python will not finish writing data to the file. The syntax for closing a file uses the `close()` method:

```
file_object.close()
```

If you fail to close your files when you're done with them, they may be overwritten later in the code, or the program may waste your system's resources.

Note: CPython currently includes a garbage collection mechanism that cleans up file objects no longer in use. However, this process is not guaranteed to always close your files in a timely fashion. Also, other Python interpreters may not have this same mechanism. So, it's still good practice to manually close your files.

The with ... as Statement

A more streamlined way to open *and* close your files in Python is to use the `with ... as` statement. Here is the syntax:

```
with open("file name", "access mode") as file_object:
```

This code wraps your I/O operations in the `with` statement, and the `as` statement defines the file object. When the `with` statement exits, it invokes the file object's built-in `__exit__()` method, closing the file. This way of opening and closing files is useful in longer, more complex programs that must open and close many files.

Access Modes

The following table lists some of the ***access modes*** available for the `open()` function.

Access Mode	Description
`"r"`	Opens the file for reading. (This is the default behavior.)
`"r+"`	Opens the file for reading and writing. Data is written to the beginning of the file, assuming it exists. If file doesn't exist, it is not created.
`"w"`	Opens the file for writing. Overwrites any data that already exists in the file. If the file doesn't exist, it creates the file.
`"w+"`	Opens the file for writing and reading. Overwrites any data that already exists in the file. If the file doesn't exist, it creates the file.
`"a"`	Opens the file for writing. Appends data to the end of a file, assuming the file exists. If the file doesn't exist, it creates the file.
`"a+"`	Opens the file for writing and reading. Appends data to the end of a file, assuming the file exists. If the file doesn't exist, it creates the file.

Best practice when using access modes is to only work with the required level of access and no more. If you only need to read the contents of a file, don't open it in `"r+"` mode. The more access you give a file object, the more ways it can introduce unwanted behavior in your program.

Note: This practice is often called the principle of least privilege, especially when in a security context.

Paths and Directories

When you open a file just by name, Python assumes the file is in the same directory as the **.py** module that you're programming in. So, assume that you've saved **readnames.py** to **C:\My Python Files**. The following code will look in **C:\My Python Files** for the **names.txt** file to write to:

```
names_file = open("names.txt", "w")
```

Of course, in the case of the `"w"` access mode, Python will create the file **names.txt** in this directory if it doesn't already exist there.

Instead of writing and reading files to and from the same directory, you can specify which directory to use with your file objects. To do this, you must supply the path to the directory you want in the file name argument of `open()`. The following example opens the **names.txt** file for writing on the user's Windows desktop:

```
names_file = open("C:/Users/You/Desktop/names.txt", "w")
```

Write Function

Once you've opened a file, writing text to it is relatively simple. You can use Python's built-in `write()` function to populate the file with a string you provide as an argument. The following code is a short example of opening the **names.txt** file, writing a line of text to it, then closing the file:

```
names_file = open("names.txt", "w")
names_file.write("This is a placeholder.")

names_file.close()
```

Because this code uses the `"w"` access mode, it will overwrite any data in the text file, if one exists, in the default directory. Otherwise, it will create a file named **names.txt** with the text "This is a placeholder." Remember, access modes are important. Giving this file object an `"r"` access mode instead of `"w"` will produce an error when you go to write to the file.

You can also format strings when writing text to a file, much like formatting strings that are output to a command line. For example, if you want to write multiple lines to the file:

```
names_file.write("This is a placeholder.\nThis is another placeholder.\nThis is a third placeholder.")
```

You aren't limited to writing to a file once. As long as it's open in a writable mode, you can use the `write()` function as many times as you want.

Guidelines for Writing to Text Files

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following guidelines to help you write to text files in your Python program.

Writing to Text Files

To help you write to text files in your Python program:

- Open files with `open()` by assigning them to file objects.
- Within the `open()` function, provide arguments for the path/name of a file and the access mode to open it in.
- Always close files with `close()` when you're done working with them.
- Streamline the file open and close operations by using the `with ... as` statement.
- Select the correct access mode for the operation(s) you're performing on the file.
- Only grant the type of access you need and nothing more.
- Open a file outside of the program's current directory by providing the directory path.
- Use the `write()` function to write a string to a text file.
- Format strings inside of this function for better readability.

ACTIVITY 6-1
Identifying Where File Operations Are Required

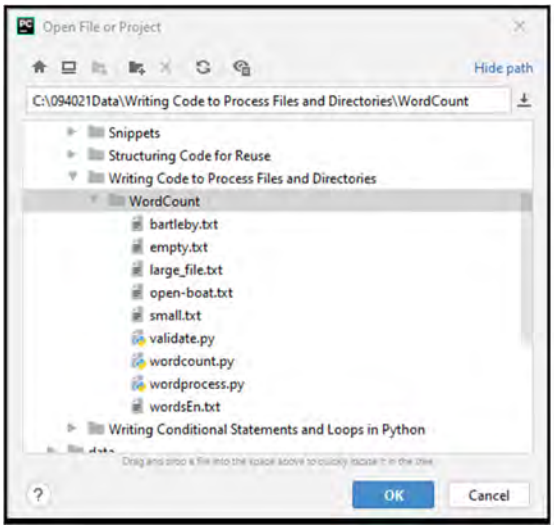
Data Files
C:\094021Data\Writing Code to Process Files and Directories\WordCount\wordcount.py
C:\094021Data\Writing Code to Process Files and Directories\WordCount\wordprocess.py
C:\094021Data\Writing Code to Process Files and Directories\WordCount\validate.py
C:\094021Data\Writing Code to Process Files and Directories\WordCount\open-boat.txt
C:\094021Data\Writing Code to Process Files and Directories\WordCount\bartleby.txt
C:\094021Data\Writing Code to Process Files and Directories\WordCount\large_file.txt
C:\094021Data\Writing Code to Process Files and Directories\WordCount\small.txt
C:\094021Data\Writing Code to Process Files and Directories\WordCount\empty.txt
C:\094021Data\Writing Code to Process Files and Directories\WordCount\wordsEn.txt

Scenario
Previously, you started making the WordCount project more modular, and started structuring the program for reuse. In this activity, you will work with a version of WordCount that is a bit more finished, and a bit more modular. The general framework for the entire program is in place. What remains to be done is primarily adding code to support various file operations the program must perform. You will start by running the program to review its current state of completion. Then you will examine the code and identify where you will need to add code to support various file operations.

---

1. Launch PyCharm and open the WordCount project.
   a) Launch **PyCharm** if you have not left it running from a previous session.
   b) In the Welcome to PyCharm window, select **Open**.
      The **Open File or Project** dialog box is shown.

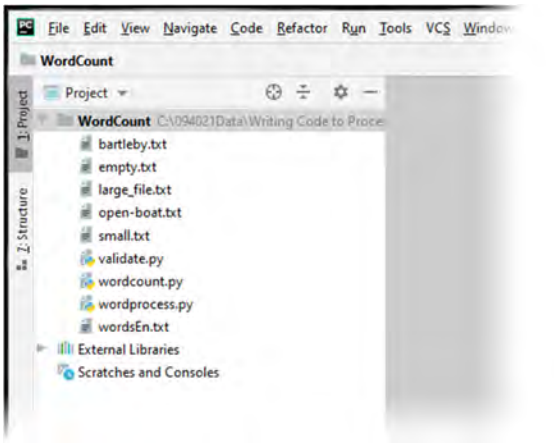
c) Beneath the **094021Data** folder, expand **Writing Code to Process Files and Directories**, and select the **WordCount** project folder.



d) Select **OK**.

The project is loaded and configured to use the Python 3.9 interpreter.

e) In the **Project** pane, expand the **WordCount** folder, if necessary.



Several files have been added to the project folder.

2. Open and run the code in the wordcount.py script.

a) Double-click **wordcount.py** to show the source code in the editor.

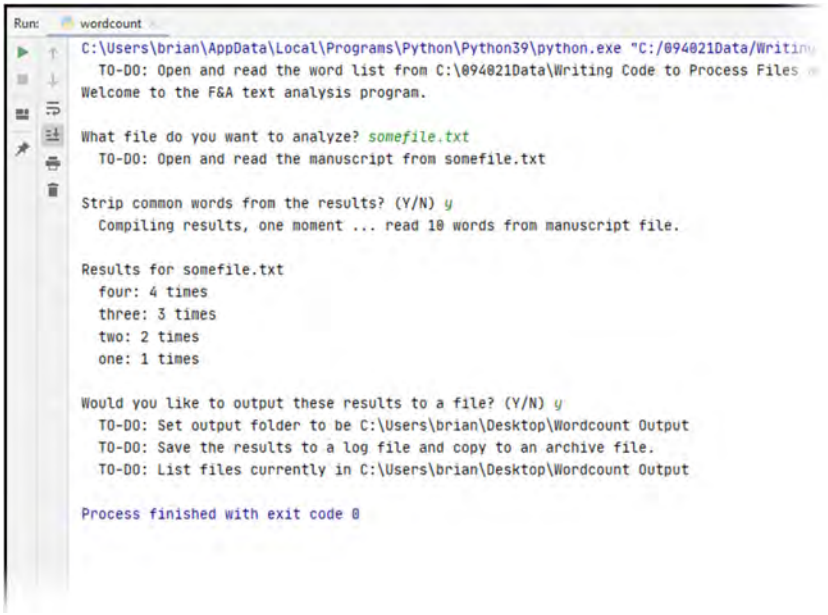
- This script is much shorter than before.
- Much of the complexity that used to be in `wordcount.py` has been moved into the `WordProcess` class stored in the `wordprocess.py` file.
- With the much more streamlined code and the comments in lines 9, 13, 20, 24, and 27, it would now be easier for someone new to this code to get an understanding of the general flow of the program.

b) Run the script in wordcount.py.

c) When prompted for a file, type ***somefile.txt*** and press **Enter**.

The actual file name that you enter doesn't matter yet, since you haven't added code to read the input file.

d) At each of the two remaining prompts, enter ***y***

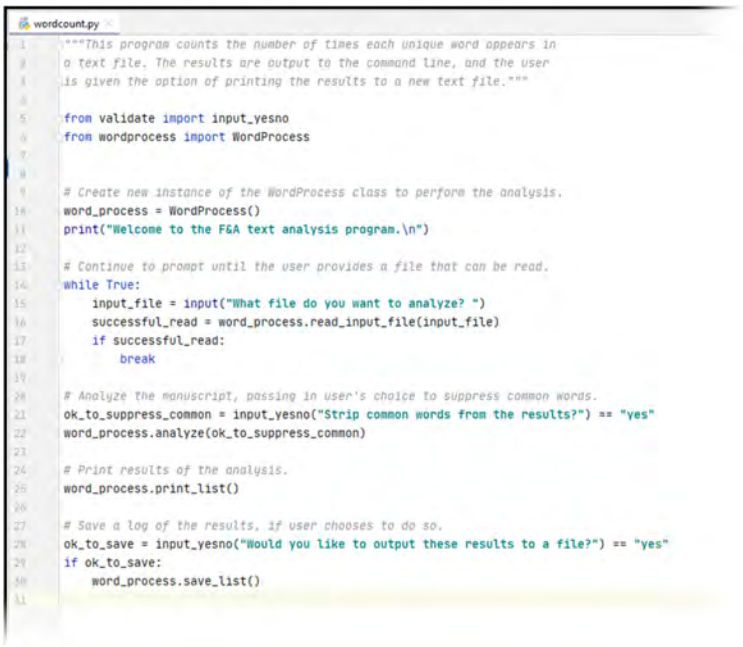


- Even though the program doesn't read a manuscript input file yet, a small set of manuscript words has been hardcoded into the program for testing purposes. The staged words include "four" (4 instances), "three" (3 instances), "two" (2 instances), and "one" (one instance).
- Print statements starting with "TO-DO" show the five instances where you must add code to perform various file operations in the project.

e) In the Run window, close the **wordcount** tab so you have more room to view the code editor.

3. Examine the main script.

a) In line 5 of the **wordcount.py** code editor, select the **+** button so the import statements are visible.



- Much of the complexity of the program has been moved out of wordcount.py. From the import statements in lines 5 and 6, you can see that wordcount.py now imports functionality from validate.py (another script containing a function named `input_yesno`) and wordprocess.py (the `WordProcess` class).
- Line 10 creates the `word_process` object based on the `WordProcess` class. Then the `word_process` object is used in other lines to perform the various general tasks the program must accomplish: reading the input file in line 16, analyzing the input file in line 22, printing the list of word counts in line 25, and saving the list of word counts to a file in line 30.
- In lines 21 and 28, much of the work involved in prompting and validating user input has been off-loaded to a new function named `input_yesno`. This function will repeatedly prompt the user until a proper yes or no response is provided.

4. Examine the new input validation function.

a) In line 5, double-click the function name **input_yesno**.

- This selects the function name. PyCharm then highlights other places in the code where the selected text appears.
- The function is imported from the validate module.
- The function is called in lines 21 and 28 to get a validated "yes" or "no" response from the user.

b) In the **Project** pane, double-click **validate.py**.

- The function accepts one argument, the message to be shown to the user.
- Line 11 prompts the user for input, converting it to lowercase.
- Line 12 checks the user's input against a list containing various acceptable responses. If the user's response is one of the acceptable responses, then line 13 is performed. If the user's response is not acceptable, then execution continues to the end of the loop, and the user will be prompted again for a response.
- Line 13 evaluates whether the response is a "y" or "yes" answer, in which case, the function always returns the full response "yes". For a "n" or "no" answer, the function always returns the full response "no".
- Putting a common task like this in a function makes it reusable, and also ensures that data entered by the user will be returned in a consistent form. Putting it in a separate module enables it to be easily reused in different projects.

c) Close the **validate.py** tab.

5. Open the file that contains the WordProcess class.

a) In line 10, double-click the object name **word_process**.

- This selects the object name, so PyCharm will highlight other places where it appears.
- The object is created in line 10 from the `WordProcess` class, and it is used in lines 16, 22, 25, and 30 to perform major tasks in the program.

b) In line 10, double-click the class name **WordProcess**.

This class contains much of the functionality that used to be in wordcount.py.

c) Press **F4**.

This PyCharm shortcut directly opens the file in which the selected class is implemented. The wordprocess.py file is shown, with the `__init__` function selected.

	**Note:** Of course, you could also have opened the wordprocess.py file from the **Project** pane.

d) In line 5, select the **+** button so the `import` statements are visible.

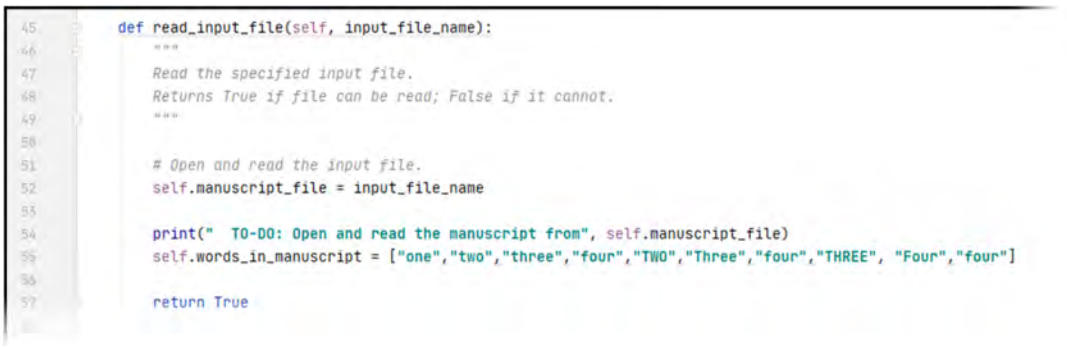
6. Examine the various methods implemented in the `WordProcess` class.

a) Scroll to line 15, and examine the initialization routine for the `WordProcess` class.

```
15  class WordProcess:
16      """
17      Class analyzes an input file and reports the number of times the most common words appear.
18      """
19
20      common_words = {"the", "be", "are", "is", "were", "was", "am",
21                      "been", "being", "to", "of", "and", "a", "in",
22                      "that", "have", "had", "has", "having", "for",
23                      "not", "on", "with", "as", "do", "does", "did",
24                      "doing", "done", "at", "but", "by", "from"}
25
26      def __init__(self):
27          """
28          Construct class instance and load the word list.
29          """
30
31          self.wordlist = []
32          self.words_in_manuscript = ""
33          self.manuscript_file = ""
34          self.word_count = {}
35          self.results_list = []
36
37          # Open and read the word list from the directory the script is in.
38          wordlist_filename = "wordsEn.txt"
39          wordlist_path = os.path.join(sys.path[0], wordlist_filename)
40
41          print("  TO-DO: Open and read the word list from", wordlist_path)
42          self.wordlist = ["one","two","three","four"]
43
```

- Lines 31 through 35 set the initial value of variables for a new instance (object) based on this class.
- Lines 38 and 39 obtain the path and file name for wordsEn.txt, the file that contains the wordlist. The value returned from `sys.path[0]` is the path for the directory in which the currently running Python script is located. The requirements for the WordCount program specify that the word list file will always be stored in the program directory.
- Lines 41 and 42 contain placeholder code that you must replace with code to actually read the word list file. For development and testing, for now the program manually creates a list of just four words that will be counted. Once the program can read wordsEn.txt, it will be able to look for more than 100,000 unique words.

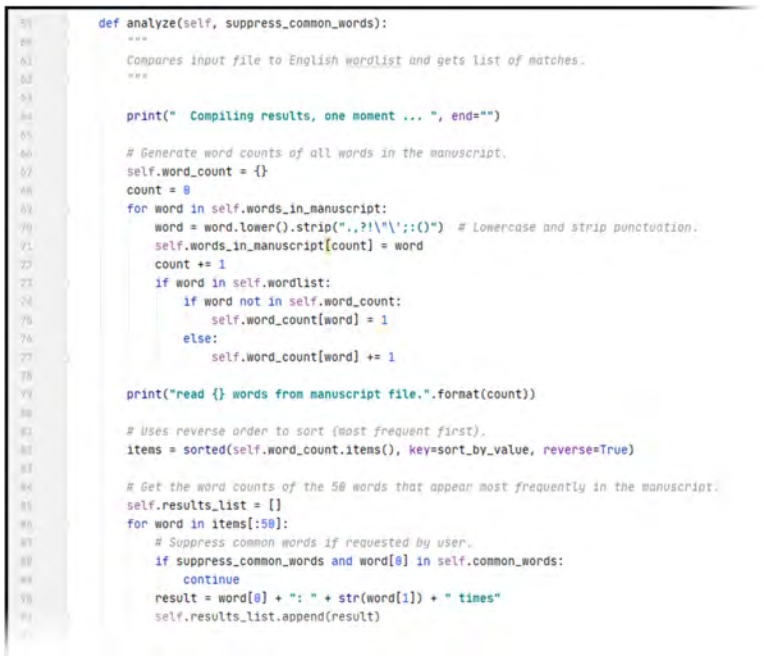
b) Scroll to line 45, and examine the method that reads the input file.



```
45      def read_input_file(self, input_file_name):
46          """
47          Read the specified input file.
48          Returns True if file can be read; False if it cannot.
49          """
50
51          # Open and read the input file.
52          self.manuscript_file = input_file_name
53
54          print("  TO-DO: Open and read the manuscript from", self.manuscript_file)
55          self.words_in_manuscript = ["one","two","three","four","TWO","Three","four","THREE", "Four","four"]
56
57          return True
```

- This method accepts a parameter of `input_file_name`, which is the name of the file the user wants to read.
- Line 52 keeps a copy of the `input_file_name` value in the object's `manuscript_file` property. This will enable other methods in the `WordCount` object to identify the name of the original file used to read in the manuscript text.
- If the method reads the input manuscript file successfully, it will return a value of `True`. Otherwise, it will return the value `False`.
- Lines 54 and 55 contain placeholder code that you must replace with code to actually read the input file. For development and testing, for now the program manually creates a list of ten words to simulate a very small manuscript containing 4 instances of the word "four", 3 instances of "three", 2 instances of "two", and 1 instance of "one".

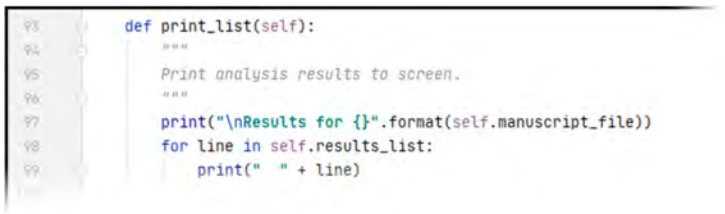
c) Scroll to line 59, and examine the method that analyzes the input file.



```
59      def analyze(self, suppress_common_words):
60          """
61          Compares input file to English wordlist and gets list of matches.
62          """
63
64          print("  Compiling results, one moment ... ", end="")
65
66          # Generate word counts of all words in the manuscript.
67          self.word_count = {}
68          count = 0
69          for word in self.words_in_manuscript:
70              word = word.lower().strip(".,?!\"\';:()")  # Lowercase and strip punctuation.
71              self.words_in_manuscript[count] = word
72              count += 1
73              if word in self.wordlist:
74                  if word not in self.word_count:
75                      self.word_count[word] = 1
76                  else:
77                      self.word_count[word] += 1
78
79          print("read {} words from manuscript file.".format(count))
80
81          # Uses reverse order to sort (most frequent first).
82          items = sorted(self.word_count.items(), key=sort_by_value, reverse=True)
83
84          # Get the word counts of the 50 words that appear most frequently in the manuscript.
85          self.results_list = []
86          for word in items[:50]:
87              # Suppress common words if requested by user.
88              if suppress_common_words and word[0] in self.common_words:
89                  continue
90              result = word[0] + ": " + str(word[1]) + " times"
91              self.results_list.append(result)
92
```

- This method accepts a parameter of `suppress_common_words`, a `True` or `False` value. If this value is `True`, then common words should be suppressed from the list of word counts.
- After the word counts are completed (lines 66 through 77), the list is sorted (line 82), the 50 most frequently appearing words selected and (optionally) the word counts are suppressed (lines 85 through 91), the results will be contained in the object's `results_list` property, from which they can be printed to screen using the `print_list` method or saved to log files using the `save_list` method.
- As you saw when you tested the program, the `analyze` method already contains the functionality required to perform the word count analysis. It will just be more useful when actual files can be processed.

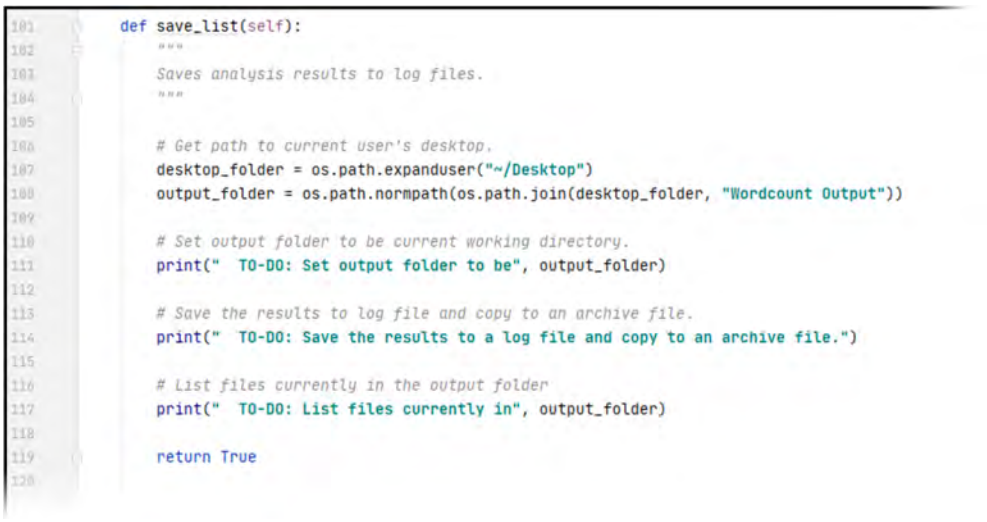
d) Scroll to line 93, and examine the method that prints results to the screen.



```
93      def print_list(self):
94          """
95          Print analysis results to screen.
96          """
97          print("\nResults for {}".format(self.manuscript_file))
98          for line in self.results_list:
99              print("  " + line)
```

- Like the `analyze` method, the functionality for `print_list` is essentially complete.
- Line 97 prints a message showing the name of the file that was analyzed.
- Lines 98 and 99 iterate through each line in the list of results produced by the `analyze` method, and print them to the screen. Each line is padded with two spaces so the output will appear indented on screen.

e) Scroll to line 101, and examine the method that saves the results to log files.



```
101     def save_list(self):
102         """
103         Saves analysis results to log files.
104         """
105
106         # Get path to current user's desktop.
107         desktop_folder = os.path.expanduser("~/Desktop")
108         output_folder = os.path.normpath(os.path.join(desktop_folder, "Wordcount Output"))
109
110         # Set output folder to be current working directory.
111         print("  TO-DO: Set output folder to be", output_folder)
112
113         # Save the results to log file and copy to an archive file.
114         print("  TO-DO: Save the results to a log file and copy to an archive file.")
115
116         # List files currently in the output folder
117         print("  TO-DO: List files currently in", output_folder)
118
119         return True
120
```

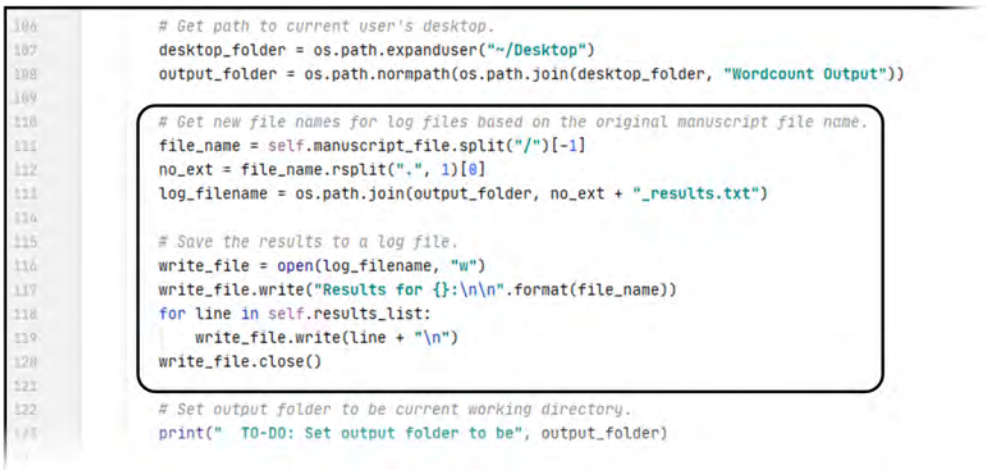
- Lines 107 and 108 identify the path to the output folder on the desktop.
- Placeholders in lines 110 through 117 show where you must add code to set the output folder as the current working directory, save the results to a log file, copy the log file to a dated archive file, and list the files currently in the output folder.
- If the files are saved successfully, the method returns `True`.

ACTIVITY 6-2
Writing to a Text File

Scenario
Now you can begin implementing code that enables your program to work with files. When the program is done, it must be able to read text in from two files: one that contains the list of common words that can be excluded from the results, and the actual manuscript text that it will analyze. The program also must be able to save its screen output to a log file. This last feature—writing to an output log—is what you will add in this activity.

---

1. Add code to format the input file name.
   a) Position the insertion point at the end of line 108, and press **Enter** twice.
   b) Enter the code as shown, including the blank line at the end.



```
106     # Get path to current user's desktop.
107     desktop_folder = os.path.expanduser("~/Desktop")
108     output_folder = os.path.normpath(os.path.join(desktop_folder, "Wordcount Output"))
109
110     # Get new file names for log files based on the original manuscript file name.
111     file_name = self.manuscript_file.split("/")[-1]
112     no_ext = file_name.rsplit(".", 1)[0]
113     log_filename = os.path.join(output_folder, no_ext + "_results.txt")
114
115     # Save the results to a log file.
116     write_file = open(log_filename, "w")
117     write_file.write("Results for {}:\n\n".format(file_name))
118     for line in self.results_list:
119         write_file.write(line + "\n")
120     write_file.close()
121
122     # Set output folder to be current working directory.
123     print("  TO-DO: Set output folder to be", output_folder)
```

Lines 110 through 113 get the name for the log file.

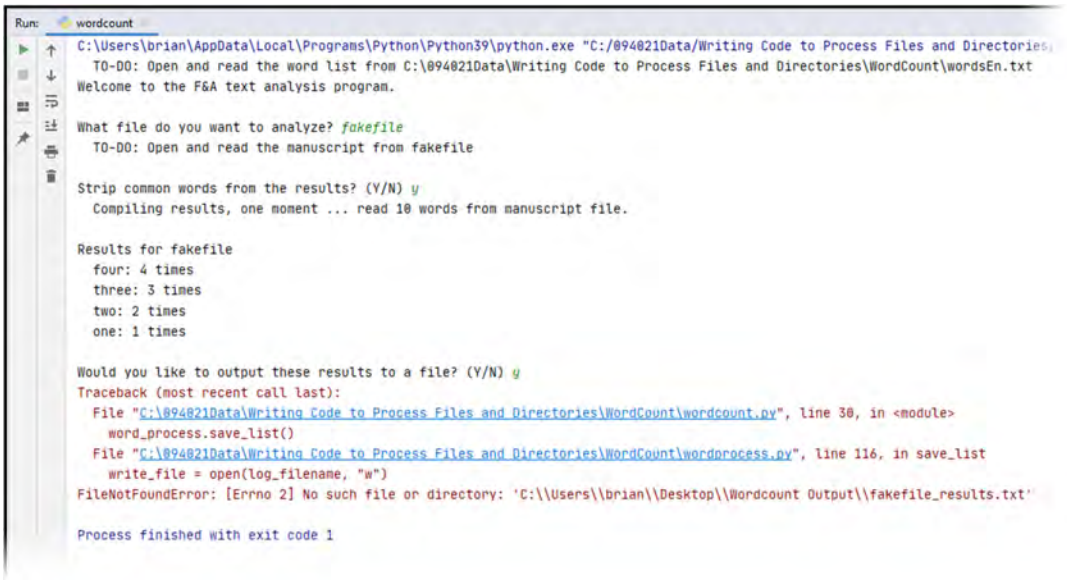
- The file name will be based on the name of the original manuscript file.
- Line 111 splits the user's input. When you provide an argument to the `split()` method, you can tell Python where to start splitting, rather than the default of using spaces. The `[-1]` tells Python to start at the end of the string, so in this case, Python will split everything before the *last* forward slash, leaving only the file name and its extension.
- Line 112 uses `rsplit()`, which differs from `split()` in that it starts splitting to the right. The first argument tells Python to split at periods, and the second argument tells Python how many times to do this split. So, Python will only split the very last period in `file_name`, removing the file extension.
- Line 113 appends "_results.txt" to the base name extracted from the manuscript file. The join method is then called to append the file name to the output folder path. The log file will be saved in the **Wordcount Output** folder on the Desktop.

Lines 115 through 120 actually save the file.

- Line 116 opens a file for writing. The first argument in `open()` is the file name, and the second argument tells how the file will be used ("w" for writing to the file).
- Line 117 uses the `write_file` file object to write the first line of text to the output file. This will act as a header for anyone who needs to read the file.
- Line 118 and 119 iterate through each item in the results list, writing it out to the file in its own line.
- Line 120 closes the file.

2. Test the program with a missing output folder.
   a) On the Windows Desktop, make sure the **Wordcount Output** folder doesn't exist.

      If the directory does exist, delete it or rename it so you can test what happens if the output folder is missing.
   b) Select the **wordcount.py** tab, and run **wordcount.py**.
   c) At the three prompts, enter ***fakefile***, ***y***, and ***y***, as shown.



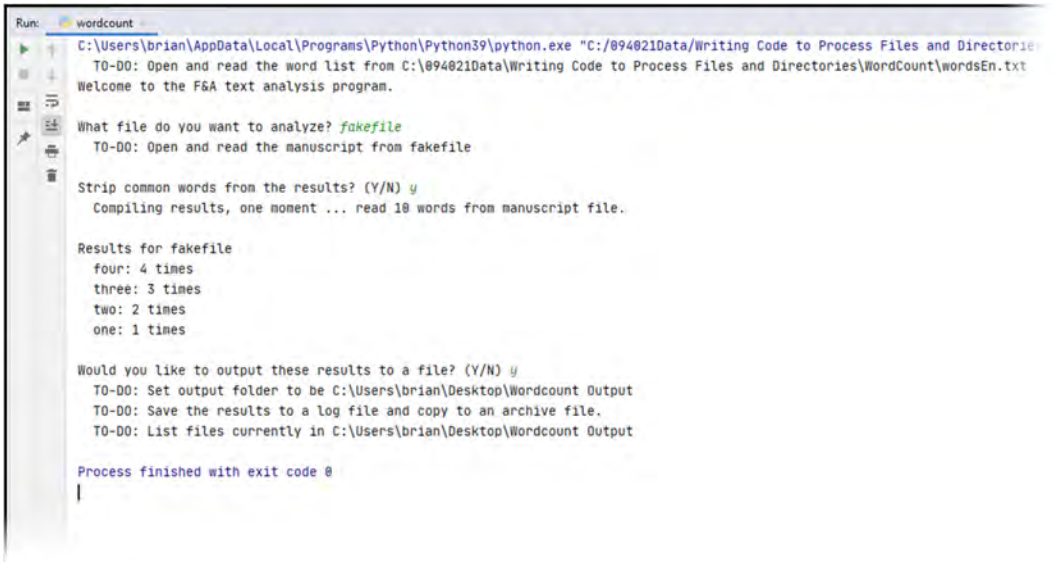
   - The program runs and processes the word counts, but it fails when it attempts to save the log file.
   - File operations are especially prone to runtime errors since the ability to complete an operation is subject to missing directories and files, file permissions, and various other problems, so it is very important to add good error-handling capabilities to routines like this.
   - For now, you can temporarily work around this flaw by ensuring the output folder exists.

   d) Create a new folder on the Windows Desktop, and name it **Wordcount Output**.

Note: To avoid errors, make sure the folder is named exactly as shown.

3. Test the program.
   a) In PyCharm, select the **wordcount.py** tab, and run **wordcount.py**.

b) At the three prompts, enter ***fakefile***, ***y***, and ***y***, as shown.



- This time, the program saves the log file successfully, and finishes with an exit code of 0.

	Note: You still have a TO-DO placeholder for "Save the results to a log file and copy to an archive file." The code you have added here partially completes this task. It saves a log file, but it does not yet copy the log file to a dated archive file. You won't delete this placeholder until you complete the entire task.

4. Confirm that the output log has been written.

a) On the Windows Desktop, open the **Wordcount Output** folder, and examine the log file.
   - The output file has been saved.
   - The file name is based on the name of the manuscript file you passed into the input prompt.

b) Double-click **fakefile_results.txt** to open it in your text editor.

   The results have been written to the file.

c) Exit the text editor.

d) In File Explorer, delete the **fakefile_results.txt** file. Leave the **Wordcount Output** folder in place, though.

e) Close the File Explorer window and return to PyCharm.

TOPIC B

Read from a Text File

More than just writing text to files, you'll want to go in the opposite direction—reading from text files. Python's code to read from files is just as easy and intuitive as writing to files.

File Information

Once you've established that a file exists, you may want to collect some information about that file before opening and reading from it. The `os.path` module also provides functions for accessing file information. For example, say you want to verify a file's size before reading from it. Because you plan on reading all of its contents, you don't want to waste time and resources reading a huge file with millions of characters. To check a file's size, use the `getsize()` function:

```
import os

if os.path.getsize("names.txt") < 1e8:
```

Like `isfile()`, the `getsize()` function takes the file name/path as its argument. It then returns the size of the file, in bytes. In the above code, if the size of the file is less than 100 MBs (`1e8` bytes in scientific notation), the conditional statement will proceed.

Note: This file size is an arbitrary example; you can supply any value you want.

You may also want to get the times a file was created, last modified, and last accessed. You can do this by using the `getctime()`, `getmtime()`, and `getatime()` functions, respectively. All three functions will return a value in Unix time.

```
import os

file_created = os.path.getctime("names.txt")
print("Created {} seconds after the Unix epoch.".format(file_created))

file_modified = os.path.getmtime("names.txt")
print("Last modified {} seconds after the Unix epoch.".format(file_modified))

file_accessed = os.path.getatime("names.txt")
print("Last accessed {} seconds after the Unix epoch.".format(file_accessed))
```

Note: On Unix operating systems, `getctime()` returns the time of the last metadata change (e.g., changing permissions), not the time of the file's creation. Unix file systems do not typically store creation time, and in those that do, it is usually not accessible.

Functions for Reading

You can read from a file in Python by using the `read()` function with your file object. After opening the file in a read mode, the default behavior for `read()` is to read from the entire file. So, assume you have a file called **names.txt** in your source code folder. The text in this file reads:

```
John
Terry
Terry
Graham
Eric
Michael
```

To open and read the entire file:

```
names_file = open("names.txt", "r")
names = names_file.read()

names_file.close()
```

This returns a string containing all of the text in the file. You can also supply an argument to tell `read()` to stop reading after a certain number of bytes. For example, `names_file.read(4)` will simply read "John", as that takes up four bytes of the file.

The readline() Method

What if you only want to read certain parts of a file, instead of the whole thing? The `readline()` method allows you to read from a file line-by-line. The first time you call `readline()`, Python will read the first line of the file. The next time you call `readline()`, Python will read the second line of the file, and so on.

```
names_file = open("names.txt", "r")
print(names_file.readline())
print(names_file.readline())
print(names_file.readline())

names_file.close()
```

This outputs:

```
John

Terry

Terry
```

The extra line breaks are due to the fact that `readline()` adds a newline character (`\n`) to the string.

The readlines() Method

You can use a `for` loop or a `while` loop to iterate over each line in a file with the `readline()` method. This is useful if you need to process each individual line. Likewise, the `readlines()` method (note the plural) will read *every* line in the file separately and place each line at its own index in a list. So:

```
names_file = open("names.txt", "r")
names = names_file.readlines()

print(names[3])

names_file.close()
```

This would print `Graham`, as that is the text on the fourth line of the file (index `3`).

Guidelines for Reading from Text Files

Use the following guidelines to help you read from text files in your Python program.

Reading from Text Files

To help you read from text files in your Python program:

- Import the `os` module to use some of these functions.
- Check to see if the file exists first by calling `isfile()`.
- Capture information about the file before opening it.
  - Call `os.path.getsize()` on the file to get its size in bytes.

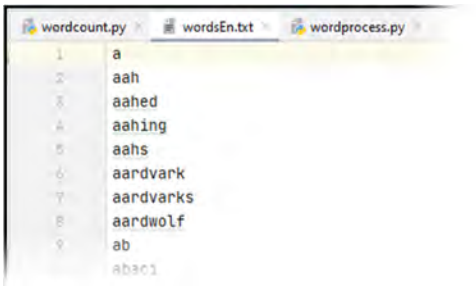
  - Call `os.path.getctime()` on the file to get its creation date on Windows file systems (Unix time).
  - Call `os.path.getmtime()` on the file to get the time it was last modified (Unix time).
  - Call `os.path.getatime()` on the file to get the time it was last accessed (Unix time).
- Use the `read()` function to read all data in a file at once.
- Supply an argument in the `read()` function to tell Python how many bytes it should read.
- Use the `readline()` function to read individual lines of a file. Each successive call reads the next line.
- Iterate over `readline()` in a loop to efficiently process individual lines in a file.
- Use `readlines()` to place every line as its own value in a list.

ACTIVITY 6-3
Reading from a Text File

Scenario
Now you're ready to enable your program to do what it was designed to do—read a manuscript from an input text file, compare it to an English word list, and calculate how many times each word appears. Once you've implemented the reading code, you'll test it out using various manuscript files that have already been copied into the project folder.

---

1. Examine the WordCount project.
   a) In the **Project** pane, double-click **wordsEn.txt**.
   b) Scroll down, and quickly scan the words in this file.



   This file contains more than 100,000 English words, each of which is on a separate line. This is the word list. Only the words that are contained in this file will be counted within the manuscript.
   c) From the **Project** pane, open and examine **empty.txt** and **small.txt**.
      - These manuscript files have been provided for testing during development.
      - The **empty.txt** file is useful to see what would happen if an empty manuscript file were analyzed.
      - The **small.txt** file is useful to verify that the program produces expected word counts. In this file, there is 1 instance of "one", 2 instances of "two", and so forth. Because it is small, it takes little time to load, which is convenient during development, when you may run a program many times to test code that you're adding.

Note: When developing and testing a program, it is helpful to have good testing data. Create various data sets that enable you to verify correct program logic and behavior, and that the program can deal acceptably with bad input.

   d) From the **Project** pane, open and examine **bartleby.txt** and **open-boat.txt**.
      These are normal manuscripts you can also use for testing.
   e) From the **Project** pane, open **large_file.txt**.
      You may see a warning in PyCharm because this is a large file. This file can be useful for testing how the script would respond to opening a very large manuscript. In some situations, very large input files may consume a lot of memory and may cause your program to become unstable. You may want to limit the size of files the program allows.
   f) Right-click one of the editor tabs, and select **Close All**.

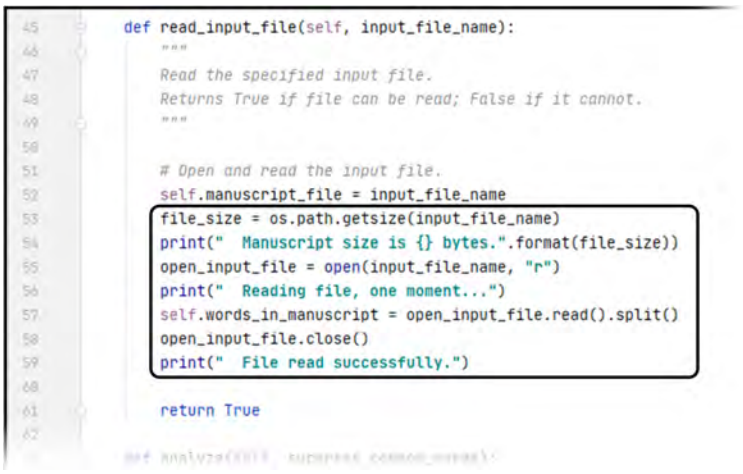
2. Write code to read the manuscript file.
   a) Open **wordprocess.py** for editing.

b) Delete lines 53 through 55, and position the insertion point at the end of line 52 as shown.

```
45      def read_input_file(self, input_file_name):
46          """
47          Read the specified input file.
48          Returns True if file can be read; False if it cannot.
49          """
50
51          # Open and read the input file.
52          self.manuscript_file = input_file_name
53
54          return True
55
56      def analyze(self, suppress_common_words):
57          """
```

You will add code here to read the input file.

c) Press **Enter** and enter the code as shown.



```
45      def read_input_file(self, input_file_name):
46          """
47          Read the specified input file.
48          Returns True if file can be read; False if it cannot.
49          """
50
51          # Open and read the input file.
52          self.manuscript_file = input_file_name
53          file_size = os.path.getsize(input_file_name)
54          print("  Manuscript size is {} bytes.".format(file_size))
55          open_input_file = open(input_file_name, "r")
56          print("  Reading file, one moment...")
57          self.words_in_manuscript = open_input_file.read().split()
58          open_input_file.close()
59          print("  File read successfully.")
60
61          return True
62
```

- Lines 53 and 54 get the input file's size and display it in the output.
- Line 55 opens the input file for reading.
- Line 56 displays a message to keep the user informed, since reading a large file may take several seconds.
- Line 57 reads the entire text file, then splits its words into a Python list to make it easy to analyze each word. The list is stored in the `WordCount` object's `words_in_manuscript` property.
- Line 58 closes the file.
- Line 59 displays a progress message to the user.
- For the moment, you are not concerned with handling error situations. The `read_input_file` method will always return `True`. Later, you will add code to return `False` if an error occurs reading the file.

3. Write code to read the word list.

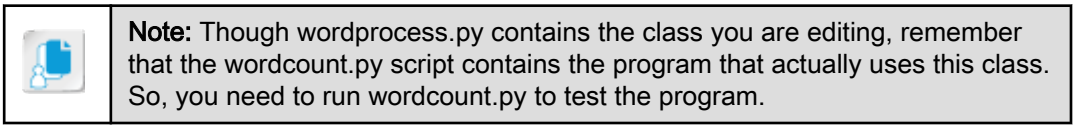
a) Delete lines 41 and 42, and enter the code to read the word list as shown.

```
26      def __init__(self):
27          """
28          Construct class instance and load the word list.
29          """
30
31          self.wordlist = []
32          self.words_in_manuscript = ""
33          self.manuscript_file = ""
34          self.word_count = {}
35          self.results_list = []
36
37          # Open and read the word list from the directory the script is in.
38          wordlist_filename = "wordsEn.txt"
39          wordlist_path = os.path.join(sys.path[0], wordlist_filename)
40
41          open_wordlist_file = open(wordlist_path, "r")
42          self.wordlist = open_wordlist_file.read().split()
43          open_wordlist_file.close()
```

- Line 41 opens the wordsEn.txt file for reading.
- Line 42 reads the entire text file, then splits its words into a Python list, which is stored in the `WordCount` object's `wordlist` property.
- Line 43 closes the file.

4. Test the program.

a) In the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.



Note: Though wordprocess.py contains the class you are editing, remember that the wordcount.py script contains the program that actually uses this class. So, you need to run wordcount.py to test the program.

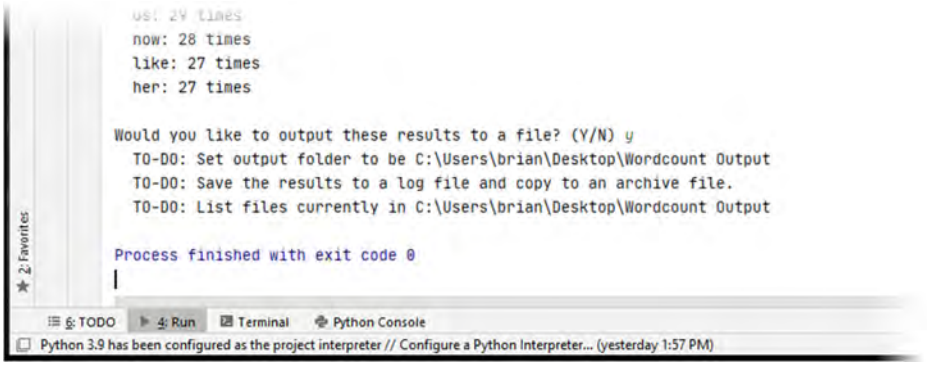
b) When prompted for the file to analyze, enter ***open-boat.txt***

```
Run:  wordcount
C:\Users\brian\AppData\Local\Programs\Python\Python39\python.e
Welcome to the F&A text analysis program.

What file do you want to analyze? open-boat.txt
  Manuscript size is 51461 bytes.
  Reading file, one moment...
  File read successfully.

Strip common words from the results? (Y/N)
```

- When you provide only the file name (not including a directory path), the file is read from the current directory. By default, the current directory is the directory the script is running from, so in this case, the file is read in from the project directory.
- The file size is shown, and the manuscript is read in from the file.

c) At each of the remaining two prompts, enter ***y*** to respond.

d) Verify that the read operations worked and the results were printed to the console.



You can scroll up to see lines that have scrolled out of view. The most frequently appearing words are summarized at the top of the report.

TOPIC C

Get the Contents of a Directory

Besides files, the other significant element of most file systems is the directory. In this topic, you'll retrieve the contents of a directory in order to learn more about the files it holds.

Directory Exists

You can determine whether or not a given directory exists similarly to individual files. If your program ends up taking a lot of input or producing a lot of output, you'll want to know for certain if the affected directory exists or not. Otherwise, your program may run into problems. Users, as well as the operating systems they use, often move, delete, or rename folders for various reasons. You don't want your program to break from the very start simply because it can't find the directory it needs. So, to check for the existence of a directory, you can use the `os.path` module. The syntax after importing is as follows:

```
os.path.isdir("Path")
```

Like `isfile()`, the `isdir()` function takes your path argument and checks whether it is on the system. Also like searching for a file, if the directory you want to search for is within the same folder as the program itself, you don't need to type a complete path. The following code searches the system for a folder at **C:\Users\You\Documents**:

```
if os.path.isdir("C:/Users/You/Documents") is False:
```

Note: Remember that backslashes in string literals indicate escape codes. Therefore, to prevent errors, you should either use forward slashes or double backslashes in a path.

Directory Contents

The `os` module also has a useful function called `listdir()` which gets the content of a directory. The function places each file/folder name in the directory as a string value in a list, which you can process however you choose. This can help you verify the existence of a certain number of files/folders or the existence of groups of files/folders. It can also help you begin managing files and folders when you don't know their exact names. The syntax for listing a directory's contents is as follows:

```
os.listdir("Path")
```

As before, you should supply the entire path in the argument, unless the folder resides in the same directory as your program.

In the following example, Python creates a list out of the directory **C:\Users\You\Documents** and assigns that list to variable `folder_contents`:

```
folder_contents = os.listdir("C:/Users/You/Documents")
```

Assume that this folder contains a bunch of loose files and folders. Printing the `folder_contents` variable would produce something like this:

```
["Business contacts.xlsx", "Cover letter_old.docx", "Invoices", "Resume.docx",
"Temp Files", "Work Samples"]
```

As you can see, the `listdir()` function captured file extensions as well as file names. Also keep in mind that this function will retrieve file *and* folder names, not just files. So, the names in the list that have no extension are likely to be folders.

File Exists

Sometimes, you may need to just confirm whether a file already exists in a directory. For example, before you open and begin reading from a file, you may need to confirm that the file actually exists. Otherwise, if the file doesn't exist, Python will fail to execute with a `FileNotFoundError`. You can check a file's existence by using the `isfile()` function from the `os.path` module. Here's an example:

```
import os

if os.path.isfile("names.txt") is False:
```

First, you must import the `os` module to gain access to the necessary function. Then, in the `isfile()` function, you supply the name of the file. Like opening a file, you can specify a directory path if you want Python to look outside of the program's current directory. When run, `isfile()` returns either `True` or `False`. So, in the above code block, if the file does not exist in the current directory (`False`) the code under the conditional statement will run.

Note: `os.path` is a submodule of the more general `os` module. In this case, you can import either and the code will work, but Python documentation suggests importing `os`.

Guidelines for Reading a Directory

Use the following guidelines to help you read directories in your Python program.

Reading a Directory

To help you read directories in your Python program:

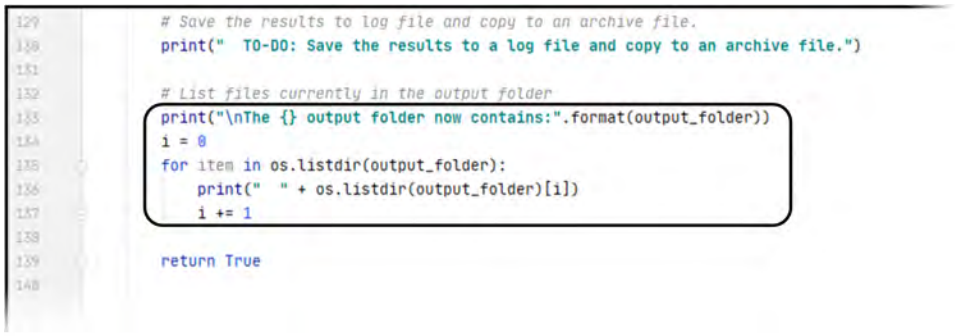
- First, determine whether the directory you want to work with actually exists.
  - Use the `isdir()` function of the `os.path` module.
- Provide the full path as an argument, or the relative path, if your target is in the same directory as your program.
- Use the `listdir()` function of the `os` module to get the file names and folders in a directory.
- Specify the path of the target directory as the argument. Use a relative path, if applicable.
- Process with the results of `listdir()` as a list to:
  - Discover the number of files and folders in a directory.
  - Identify groups of files and folders.
  - Begin managing files and folders when you don't know their exact names and locations.
- Keep in mind that `listdir()` retrieves file *and* folder names.
- Look for file extensions or a lack thereof to determine if an item is a file or folder.

ACTIVITY 6-4
Getting the Contents of a Directory

Scenario
After you output your results to a file (assuming the user chooses this option), you also want to tell the user what files currently exist in the output directory. This way, they can immediately know the names of all the files that have been analyzed and saved to the output directory. So, you'll write code to get that directory's contents.

---

1. Enumerate the files in the output folder.
   a) In **wordprocess.py**, delete the placeholder print statement in line 133, and enter the code as shown.



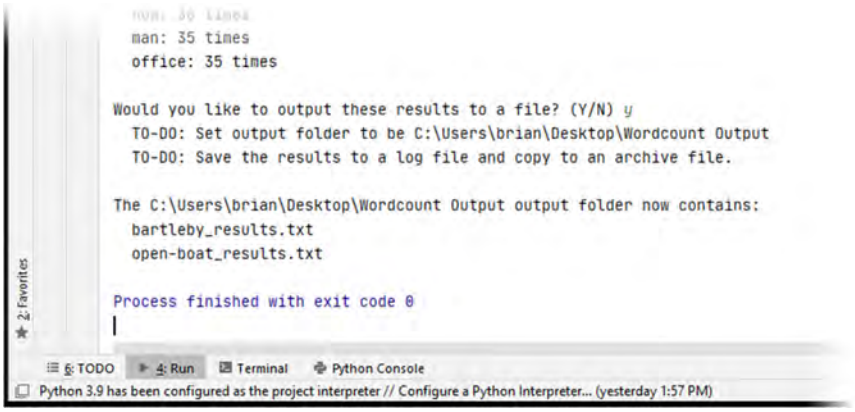
```
129         # Save the results to log file and copy to an archive file.
130         print("  TO-DO: Save the results to a log file and copy to an archive file.")
131
132         # List files currently in the output folder
133         print("\nThe {} output folder now contains:".format(output_folder))
134         i = 0
135         for item in os.listdir(output_folder):
136             print("  " + os.listdir(output_folder)[i])
137             i += 1
138
139         return True
140
```

   - Line 134 initializes the counter *i* to use in the loop.
   - Line 135 iterates through each item in the directory listing.
   - Line 136 prints the name of the item, using the counter i as an index to access the item from the list.
   - Line 137 increments the counter.

2. Test the program to get the current directory's contents.
   a) In the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.
   b) At the first prompt, enter ***bartleby.txt***

      Because you provide no path, the manuscript file will be read from the current directory (the project folder).
   c) At each of the remaining prompts, enter ***y***

d) Verify that the program has listed the contents of the **Wordcount Output** folder.



- In various test runs of the program, you have analyzed two different manuscripts, and their results are stored in the output folder.
- Note that if you were to run another analysis of open-boat.txt or bartleby.txt, the new log file would overwrite the old one.

e) In the Run window, close **wordcount**.

TOPIC D

Manage Files and Directories

When you directly use Windows®, Linux®, or any other operating system, you're able to perform a number of management tasks on your files and folders. Python gives you the tools to do all of these things within your application.

File Operations

The four most common file management operations you'll write in your Python programs are:

- Renaming
- Deleting
- Moving
- Copying

File Renaming and Deleting

To rename a file on the system, you can use the `rename()` function from the `os` module. This function takes two basic arguments: a source and a destination. The source is the full path including the file you want to rename, and the destination will generally be the same path, but with a different file name. If the source and destination paths are different, the `rename()` function will essentially work like a rename *and* move operation (that is, it will remove the file from the source and put it in the destination with the new name). The following code demonstrates renaming a **Business contacts.xlsx** file to **Business contacts_old.xlsx**:

```
os.rename("C:/Users/You/Documents/Business contacts.xlsx", "C:/Users/You/
Documents/Business contacts_old.xlsx")
```

To delete a file on the system, use the `os` module's `remove()` function. This function takes one argument, which is the path and file name of the file you want to delete. This delete operation bypasses the Recycle Bin on the operating system and deletes the file outright. The following code demonstrates removing the **Cover letter_old.docx** file from **C:\Users\You\Documents**:

```
os.remove("C:/Users/You/Documents/Cover letter_old.docx")
```

File Moving and Copying

To move files, you'll actually need to import the `shutil` module. The function you're looking for is `move()`, and it takes two arguments: a source and a destination. You'll specify the path and file name in the destination; the path will usually be different than the source, but the file name will typically be the same. However, `move()` can double as a move *and* rename operation if you provide a different file name for the destination. The following code moves **Business contacts_old.docx** from **C:\Users\You\Documents** to **C:\Users\You\Documents\Archive**:

```
shutil.move("C:/Users/You/Documents/Business contacts_old.docx", "C:/Users/You/
Documents/Archive/Business contacts_old.docx")
```

Note: If your destination's path has folders that don't exist, the operation will create those folders for you.

Lastly, you can copy a file if you want it to remain at its source. You can do this with the `shutil` module's `copyfile()` function. As you might expect, the two arguments are the source and destination. Changing the file name in the destination will also perform both a copy *and* rename operation. The following code copies **Resume.docx** to **C:\Backup**:

```
shutil.copyfile("C:/Users/You/Documents/Resume.docx", "C:/Backup/Resume.docx")
```

Note: You can also use the `copy()` and `copy2()` functions instead. The former does not require you to name the destination file, only its path. The latter does the same as `copy()`, only it also copies all of the source file's metadata.

File Search

You've seen how to find the true/false value of a file's existence, but what if you want to search for files and do more with the results? Python has a module called `glob`, which has a function also called `glob()` that helps you search for patterns in file names. If `glob()` finds any file names that match the pattern you specify, it places the path and file names in a list that you can then process. The `glob()` function takes one argument, which is the path with the file name pattern you want to use in the search.

A useful application of `glob()` involves using a wildcard character (`*`). Assume that you only want to collect the names of XML-based Microsoft® Word documents. It wouldn't be very efficient to collect every name using `listdir()`, then write more code to parse only the items that end in **.docx**. Instead, you could do the following:

```
file_list = glob.glob("C:/Users/You/Documents/*.docx")
```

This will automatically return a list of every file in that directory that ends in **.docx**, because the asterisk indicates that *any* characters before **.docx** are acceptable. You can also use a wildcard character in other places, like to retrieve files that have the same name, but different extensions:

```
file_list = glob.glob("C:/Users/You/Documents/contacts.*")
```

You're also not just limited to one wildcard character:

```
file_list = glob.glob("C:/Users/You/Documents/*contacts.*")
```

This will ensure that the list includes both **Business contacts.xlsx** and **contacts.docx**.

Folder Operations

You can write Python code to manage folders in much the same way as you would to manage files. Some of the common folder operations are as follows:

- Creating
- Renaming
- Deleting
- Moving
- Copying

Folder Creation, Renaming, and Deleting

To create a new folder in the file system, you can use the `mkdir()` function from the `os` module. This function takes one argument, which is the directory path that you want to create. Note that the path you provide must already exist, other than the folder you're actually creating. In other words, you can't create **C:\New Folder\New Subfolder** if **New Folder** doesn't exist. The following code creates a new folder in **C:\Users\You\Documents** called **Resources**:

```
os.mkdir("C:/Users/You/Documents/Resources")
```

Renaming a folder is essentially the same as renaming a file. Use the `os.rename()` function and supply both the source and destination folder. The following code renames the **Invoices** folder to **March Invoices**:

```
os.rename("C:/Users/You/Documents/Invoices", "C:/Users/You/Documents/March
Invoices")
```

Deleting directories is a little different. You can use the `os.rmdir()` function from the `os` module to delete an empty directory, but if the directory you specify isn't empty, Python will return an error. Deleting a non-empty directory is possible with `rmtree()` from the `shutil` module. This will delete the directory you specify, all files inside that directory, and all subfolders in that directory. As with `remove()`, this operation will bypass the Recycle Bin. The following code deletes everything in **C:\Users\You\Documents**:

```
shutil.rmtree("C:/Users/You/Documents")
```

Folder Moving and Copying

You can move a directory by using the `move()` function from `shutil`, just like a file. Supply the source and destination as two arguments. If the destination path does not exist, Python will create it automatically. The following code moves **Temp Files** from **C:\Users\You\Documents** to **C:**:

```
shutil.move("C:/Users/You/Documents/Temp Files", "C:/Temp Files")
```

To copy a directory and all of its contents, use the `shutil` module's `copytree()` function. The following code copies **C:\Users\You\Documents\Work Samples** to **C:\Backup**:

```
shutil.copytree("C:/Users/You/Documents/Work Samples", "C:/Backup")
```

Current Directory

By default, your current working directory is wherever your Python program is located. To change this, you can use the `chdir()` function from the `os` module. So, if you don't want to keep typing out **C:\Users\You\Documents** in your file/folder operations, you can just change to that directory:

```
os.chdir("C:/Users/You/Documents")
```

You can also verify which directory you're currently working in with the `getcwd()` function:

```
>>> os.getcwd()
"C:\\Users\\You\\Documents"
```

Guidelines for Managing Files and Directories

Use the following guidelines to help you manage files and directories in your Python program.

Manage Files

When managing files:

- Import the `os` and `shutil` modules before managing files and folders.
- Use `os.rename()` to rename a file.
  - Supply the source file as the first argument, and the file you want to rename it to as the second argument.
- Use `os.remove()` to delete the file from the path you provide.
- Use `shutil.move()` to move a file.
  - Supply the source file as the first argument, and the destination file as the second argument.
- Use `shutil.copyfile()` to copy a file.
  - Supply the source file as the first argument, and the destination file as the second argument.
- Use `shutil.copy2()` to copy a file and all of its metadata.
- Import the `glob` module to use the pattern search function.
- Use `glob.glob()` to run a pattern search and return a list of each result.
  - Use a wildcard character (`*`) before the file extension to get all files in that path with that extension.
  - Use a wildcard character after the file name to get all files with that name, even if they have different extensions.
  - Use multiple wildcard characters to include more results in your search pattern.

Manage Directories

When managing directories:

- Use `os.mkdir()` to create a folder with the path you specify.
- Use `os.rename()` to rename a folder.
  - Supply the source folder as the first argument, and the folder you want to rename it to as the second argument.
- Use `shutil.rmdir()` to delete an empty directory.
- Use `shutil.rmtree()` to delete a directory and all of its contents.
- Use `shutil.move()` to move a folder.
  - Supply the source folder as the first argument, and the destination folder as the second argument.
- Use `shutil.copytree()` to copy a folder.
  - Supply the source folder as the first argument, and the destination folder as the second argument.
- Use `os.chdr()` to change the current working directory to the one you specify.
- Use `os.getcwd()` to retrieve your current working directory.

ACTIVITY 6-5
Managing Files and Directories

Scenario
Earlier, you found that the program throws an exception if you attempt to save an output log when the **Wordcount Output** folder doesn't already exist. To make the program a bit more robust, you will revise it to create the output folder if it doesn't already exist.

Also, the project requirements call for a dated archive copy to be created for each output file you generate. These copies will be placed in the same **Wordcount Output** folder, will include the date and time they were generated in the file name, and they should be clearly marked by file name as being archive copies.

---

1. Check to see if the output directory already exists.
   a) Position the insertion point at the end of line 112, and press **Enter** twice.
   b) Type the following code:

```
105     def save_list(self):
106         """
107         Saves analysis results to log files.
108         """
109
110         # Get path to current user's desktop.
111         desktop_folder = os.path.expanduser("~/Desktop")
112         output_folder = os.path.normpath(os.path.join(desktop_folder, "Wordcount Output"))
113
114         # Set output folder to be current working directory.
115         if os.path.isdir(output_folder) is False:
116             os.mkdir(output_folder)
117         os.chdir(output_folder)
118
119         # Get new file names for log files based on the original manuscript file name.
```

   - Line 115 checks to see if the output folder already exists on the user's computer.
   - If it doesn't, line 116 will create the folder.
   - Line 117 sets the current working directory to the output folder. This will assume that all files used in file operations such as writing, copying, and moving will be located in the output folder, unless a complete path is provided when referring to the file.

   c) Delete lines 131 through 133.

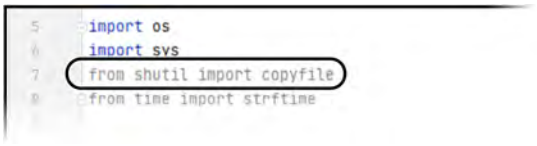
   These lines contain a placeholder comment, print statement, and trailing blank. You just completed the task identified by this placeholder.

   d) Examine the one remaining TO-DO placeholder in line 132.

```
128             write_file.write(line + "\n")
129         write_file.close()
130
131         # Save the results to log file and copy to an archive file.
132         print("  TO-DO: Save the results to a log file and copy to an archive file.")
133
134         # List files currently in the output folder
135         print("\nThe {} output folder now contains:".format(output_folder))
```

   You've already written the code to save the results to a log file. Now you will copy the log file to a dated archive file.

2. Write code to copy the output log to a dated archive file.
   a) Position the insertion point at the end of line 6, and press **Enter**.
   b) Type the code as shown.

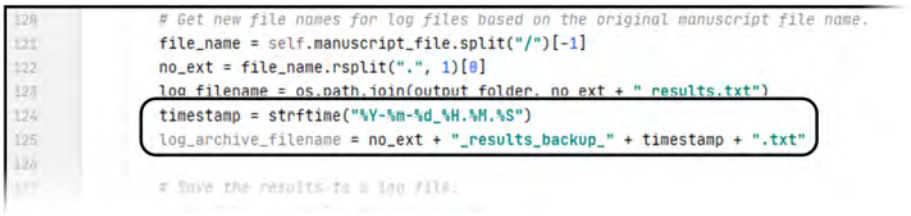


   - Line 7 imports the `copyfile` method from the `shutil` module.
   - In line 8, an `import` statement for the `strftime` method has already been added in preparation for generating a file name that contains the date and time.

   c) In lines 120 through 123, examine the code that produces the log file name.

   - This is the code you added to get the base name of the input file, to which you append "_results.txt" to produce a name for the log file.
   - You will build upon this code to produce a dated name you will use for the archive copy of the log file.

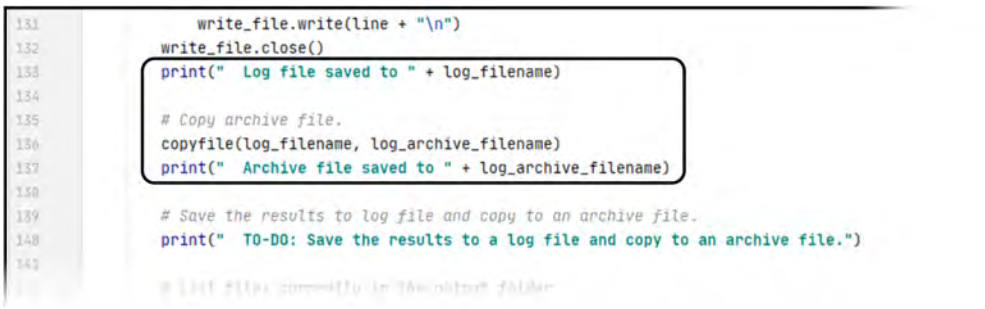
   d) Position the insertion point at the end of line 123, and press **Enter**.
   e) Enter two more lines of code as shown.



   - Line 124 uses the `strftime` method to create a formatted text string containing the date and time. This is used to create a time stamp in the file name of the copied file.
   - Line 125 creates a name for the archive file using the base file name, the text "_results_backup_", the time stamp, and the file extension for text files.

   f) Position the insertion point at the end of line 132, and press **Enter**.

g) Enter the code as shown.

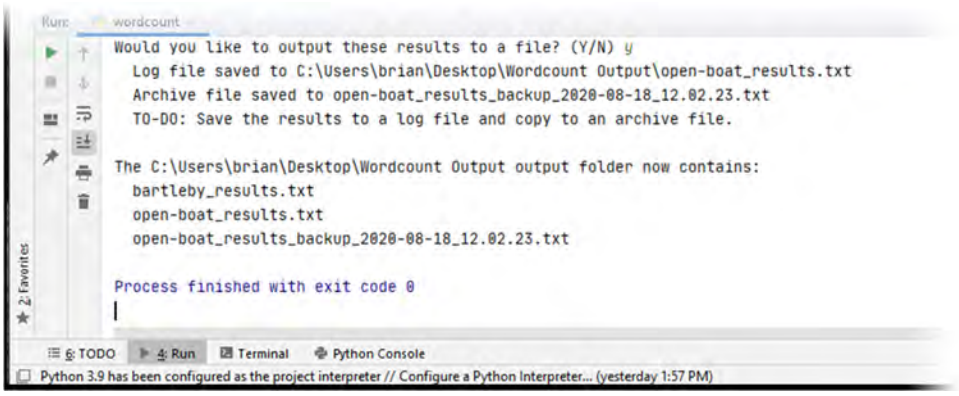


- Line 133 displays a message to show that the log file has been saved.
- Line 136 copies the log file to create an archive file.
- Line 137 displays a message to show that the archive file has been saved.

3. Test the archive copy functionality.

a) In the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.

b) At the three prompts, enter ***open-boat.txt***, ***y***, and ***y***

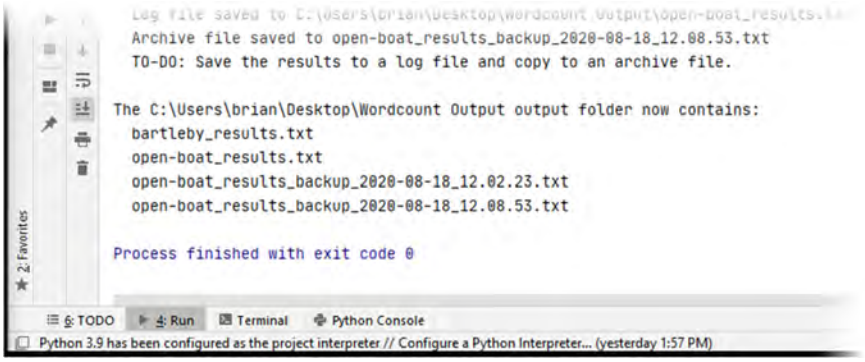


The manuscript is analyzed, and the output files are generated and listed. Files from previous runs are listed along with the dated archive file.

c) Re-run **wordcount.py**.

d) At the three prompts, enter ***open-boat.txt***, ***y***, and ***y***

e) Examine the new directory listing.



A new log file has been created, overwriting the old `open-boat_results.txt` file. However, there are two `open-boat_results_backup` files because the file name includes a time stamp that means a unique file will be created each time. These backup files will continue to accrue each time the program runs.

f) Delete the TO-DO placeholder statements in lines 139 through 141.

4. Test the program's ability to create a new output folder.

a) On the Windows Desktop, delete the **Wordcount Output** folder along with the files it contains.

b) In PyCharm, in the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.

c) At the three prompts, enter ***open-boat.txt***, ***y***, and ***y***

The missing output folder is created automatically, and the new log and archive file are saved to it.

5. Clean up the workspace.

a) From the Run window, close the **wordcount** tab.

b) Close the project, and exit PyCharm.

Summary

In this lesson, you used Python to read from and write to text files and work with your computer's file system. Input/output operations are some of the most fundamental in any program, and knowing how to process files and folders is an important component of these operations.

What are some of the challenges of writing file manipulation Python apps for different operating systems?

How do you plan on testing file management operations like copying, moving, deleting, and renaming files and folders?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

7 Dealing with Exceptions

Lesson Time: 1 hour, 30 minutes

Lesson Introduction

Dealing with exceptions in Python® is a bit like life. With some foresight and careful planning, you can prevent some kinds of problems from happening. But other kinds of problems can't be prevented, and you must deal with them as they arise.

Lesson Objectives

In this lesson, you will:

- Handle exceptions.
- Raise exceptions.

TOPIC A

Handle Exceptions

Exceptions are commonplace in the programming process, and they can often break an application in unforeseen ways. Thankfully, Python® has built-in functionality for addressing exceptions, and in this topic, you'll leverage this functionality.

Error Prevention

You might begin development of a particular routine by writing code to perform a task assuming all goes well. For example, you write code to save a file to a particular output directory. If all goes well, the code runs its normal flow of execution and the file is written. However, in the real world, things do not always go well, so you must employ various strategies to make your code reliable and problem free.

One strategy is *error prevention*. You test for a possible problem (such as a missing directory) in advance, and then make corrections as needed (for example, creating the directory that was missing). In that way, your code can resolve a problem before it has a chance to happen. The code may be able to resolve some problems on its own (such as creating a directory), but it may have to prompt the user to resolve other problems, such as logging onto a network or inserting a storage medium such as a USB drive.

Exceptions

It may not be possible or expedient to detect every potential problem situation in advance, so sometimes a program may encounter problems that get in the way of the normal flow of execution. When something prevents the normal flow of execution from completing, an ***exception*** is said to have occurred. Python understands and is able to interpret the code, but is unable to execute it for whatever reason. These types of errors are common in programs that access external resources. If, for example, your code contains a statement that reads a file in a storage location that doesn't exist or the program doesn't have adequate rights to access, an exception will be thrown when that statement is run.

Exceptions may or may not result in the program crashing. They may just make a certain part of a program non-functional while allowing the rest of the program to run just fine. Some exceptions are unavoidable and don't even require major action on the programmer's part.

For a basic example of an exception, check out this simple arithmetic operation:

```
a = 3
b = 0

c = a / b
```

This code is syntactically sound, and most IDEs may not even pick up on the problem before execution. However, attempting to run the program (from the console) will result in the following error:

```
Traceback (most recent call last):
    File "<input">, line 1, in <module>
ZeroDivisionError: division by zero
```

This is an exception because Python understood the arithmetic, but the arithmetic itself (dividing by zero) is mathematically undefinable. Python has no choice but to produce an exception. In this case, Python will likely stop executing any code that follows this operation. However, any code that executes before this operation may still work.

Note: Python will usually point to one or more line numbers where it encountered any exceptions. This can help you debug your code, but keep in mind that these line numbers do not always indicate where you need to actually apply a fix.

Types of Exceptions

When Python encounters an exception, it usually categorizes it in one of many ways. This makes it easier for you to handle exceptions when you write the code to do so. The following table lists some of the more common exceptions that may affect your program.

Exception	*Occurs When...*
`AttributeError`	Python fails to reference or assign an attribute.
`ImportError`	Python fails to find the module or object named in an `import` statement.
`IndexError`	You attempt to find an index (e.g., in a list) that doesn't exist.
`KeyError`	You request a key that is not in a dictionary.
`MemoryError`	The program runs out of memory.
`NameError`	You request a name (e.g., a variable) that is not defined.
`OSError`	Python encounters a system-related error, like an I/O error. Has several subclasses, like `FileNotFoundError`.
`RuntimeError`	Python detects an error that does not fit in any of the other categories.
`TypeError`	You attempt to perform an operation on a data type that isn't compatible with this operation.
`ZeroDivisionError`	You attempt to divide by zero.

Note: Some exceptions, like `IOError`, were merged with `OSError` in Python 3.

Note: For an exhaustive list of exceptions, navigate to **https://docs.python.org/3/library/exceptions.html**.

Try ... Except

In programming languages like Python, there are built-in tools that allow you to defend against the unwanted effects of an exception. This process is called ***exception handling***, as the programmer writes code to anticipate an exception and resolve it. In Python, you can handle exceptions by implementing the `try ... except` statement.

Note: This statement is known in many other languages as `try ... catch` and does essentially the same thing.

Basic exception handling code is structured with one `try` branch and one `except` branch after it. Within each of these branches is applicable code. The code that you actually want to test for errors, or the code that you think might cause issues, goes inside the `try` branch. Within the `except` branch is the code that Python executes if indeed there is an error. The following example demonstrates exception handling for possible errors in file input/output:

```
try:
    my_file = open("names.txt", "r")
```

```
    read_file = my_file.read()
except:
    print("An unspecified error has occurred.")
```

Assuming there's no **names.txt** file in the current directory, Python will print to the console "An unspecified error has occurred." So, in the `try` branch, you tried to open a file, and in the `except` branch, you output a warning in case that didn't work. If there was a **names.txt** file in the local directory, the code in the `except` branch wouldn't execute.

Code in the `except` branch can be a simple warning, like above, or it can be more complex code that attempts to keep the programming running as normal. How you handle exceptions will depend heavily on the exception itself, as well as the nature of the code you're trying.

Advanced Exception Handling

You can add a little complexity to the exception handling process by actually specifying exception types. So, let's say you want Python to respond differently based on what type of exception it encounters. At the `except` branch, you can actually reference the type, and you can even include multiple `except` branches. Like an `elif` branch in a conditional statement, Python will go through each branch until it finds the value that matches the exception it found. So, in the following example, the code first checks to see if the open and read processes can't find the file. If they can't, the warning message is output. If that wasn't the error Python encountered, it then checks if the error was due to a lack of memory. If so, it produces a memory-related error:

```
try:
    my_file = open("names.txt", "r")
    read_file = my_file.read()
except FileNotFoundError:
    print("File not found!")
except MemoryError:
    print("Out of memory!")
```

If you don't define the exception code in your `except` branches, then Python will execute the code for *any* exception. This isn't an ideal way to handle exceptions, as it can make it very difficult to pinpoint the root cause of the problem. You can also add multiple codes to one `except` branch, in case these different codes should all produce the same result.

Also like a conditional statement, you can terminate a `try ... except` statement with an `else` branch. This branch will execute if no exceptions were found:

```
try:
    my_file = open("names.txt", "r")
    read_file = my_file.read()
except FileNotFoundError:
    print("File not found!")
except MemoryError:
    print("Out of memory!")
else:
    print("File read successfully!")
```

You can also use an argument with an `except` branch. The argument typically gives you more information about that particular exception. To use an argument, after the exception code, write `as arg`. The `as` statement is necessary, but you can name the argument variable anything you choose. For example:

```
try:
    my_file = open("names.txt", "r")
    read_file = my_file.read()
except FileNotFoundError as arg:
    print(arg)
```

The argument variable is `arg`, and if Python encounters the `FileNotFoundError` exception, it will print this argument variable. In this case, the output is:

```
[Errno 2] No such file or directory: 'names.txt'
```

The actual value of this argument variable will vary depending on the exception. Rather than just using your own error warnings, you can leverage an exception's argument to produce that warning for you.

The `finally` Branch

With `try ... except` statements, you can also use a `finally` branch. The `finally` branch defines code that executes whether there is an exception or not. For example:

```
try:
    my_file = open("names.txt", "r")
    read_file = my_file.read()
except FileNotFoundError as arg:
    print(arg)
else:
    print("File read successfully!")
finally:
    print("Moving on...")
```

Whether or not the `FileNotFoundError` exception is produced, "Moving on..." will always print to the screen. When **names.txt** doesn't exist, this is the output:

```
[Errno 2] No such file or directory: 'names.txt'
Moving on...
```

When **names.txt** does exist:

```
File read successfully!
Moving on...
```

Guidelines for Handling Exceptions

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following guidelines to help you handle exceptions in your Python programs.

Handle Exceptions

When handling exceptions:

- Familiarize yourself with the different types of exceptions in Python.
- Wrap any code you want to test for errors with a `try ... except` statement.
- Place the code to be tested in the `try` branch.
- Place the code to execute if there is an exception in the `except` branch.
- Handle exceptions in a way that is most appropriate to the exception itself, as well as the nature of your program.
- Specify exception types in your `except` branches to pinpoint the root cause of the problem.
- Use multiple `except` branches to catch multiple exception types.
- Use an `else` branch at the end of a `try ... except` statement in case there are no errors.
- Add `as arg` to your `except` branches to define an argument variable.
- Use argument variables to produce specified warning messages.
- Use a `finally` branch to define code that must be executed whether or not an exception was found.

ACTIVITY 7-1
Handling Exceptions

Data Files

C:\094021Data\Dealing with Exceptions\WordCount\bartleby.txt

C:\094021Data\Dealing with Exceptions\WordCount\empty.txt

C:\094021Data\Dealing with Exceptions\WordCount\large_file.txt

C:\094021Data\Dealing with Exceptions\WordCount\open-boat.txt

C:\094021Data\Dealing with Exceptions\WordCount\small.txt

C:\094021Data\Dealing with Exceptions\WordCount\validate.py

C:\094021Data\Dealing with Exceptions\WordCount\wordcount.py

C:\094021Data\Dealing with Exceptions\WordCount\wordprocess.py

C:\094021Data\Dealing with Exceptions\WordCount\wordsEn.txt

Scenario

You have written code to perform various file operations. File operations are prone to various runtime problems, like trying to open a file that doesn't exist, navigating to a directory that doesn't exist, or accessing directories or files you don't have permission to access.

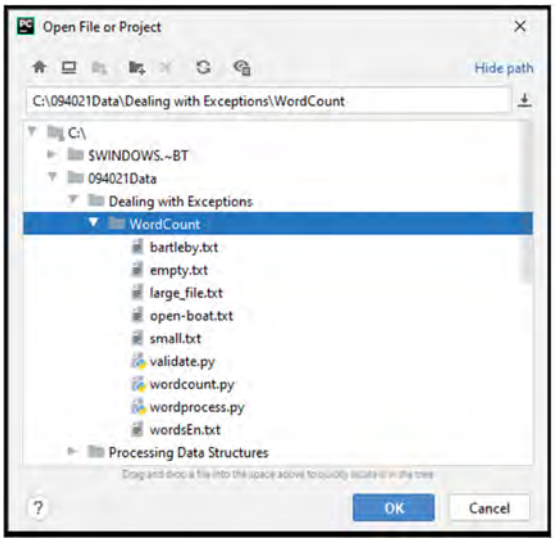
Some code has already been added to the WordCount project to deal with various types of exceptions a user might encounter when trying to save the output log. You will examine the code that has been added, and you will test it.

Then you add an exception handler to the method that reads the input file.

---

1. Launch PyCharm and open the WordCount project.
   a) Launch PyCharm.
   b) In the Welcome to PyCharm window, select **Open**.
      The **Open File or Project** dialog box is shown.

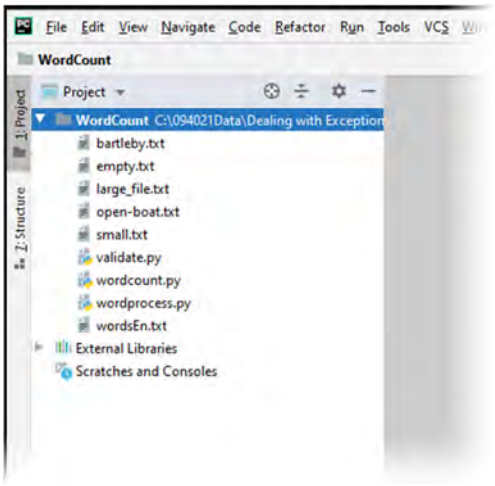
c) Beneath the **094021Data** folder, expand **Dealing with Exceptions**, and select the **WordCount** project folder.



d) Select **OK**.

The project is loaded and configured to use the Python 3.9 interpreter.

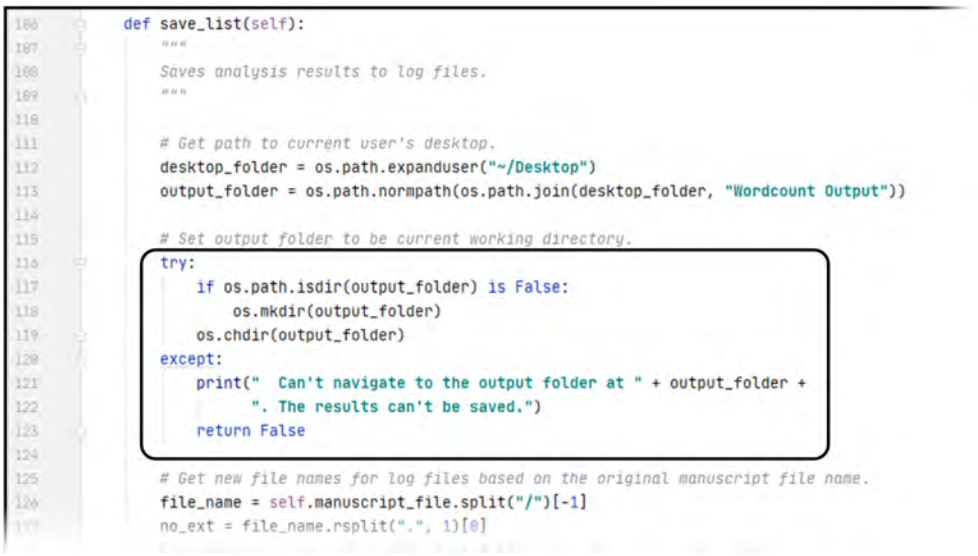
e) In the **Project** pane, expand the **WordCount** folder, if necessary.



f) Double-click **wordprocess.py** to show the source code in the editor.

2. Examine exception handlers that have been added to the code.

a) Examine the code in lines 115 through 123.



```
def save_list(self):
    """
    Saves analysis results to log files.
    """

    # Get path to current user's desktop.
    desktop_folder = os.path.expanduser("~/Desktop")
    output_folder = os.path.normpath(os.path.join(desktop_folder, "Wordcount Output"))

    # Set output folder to be current working directory.
    try:
        if os.path.isdir(output_folder) is False:
            os.mkdir(output_folder)
        os.chdir(output_folder)
    except:
        print("  Can't navigate to the output folder at " + output_folder +
              ". The results can't be saved.")
        return False

    # Get new file names for log files based on the original manuscript file name.
    file_name = self.manuscript_file.split("/")[-1]
    no_ext = file_name.rsplit(".", 1)[0]
```

- Lines 117 through 119 have been indented under `try:`. These are the statements that create the output folder if necessary and set it to be the current working directory.
- Lines 121 through 123 are indented under the `except:` block. They report a problem with the output folder and will cause the `save_list` method to return `False`.
- If any sort of exception occurs within the code under the `try:` block, the code under the `except:` block will be executed to handle the exception.
- This will enable the program to deal with any sort of exception that is thrown when the statements try to create or navigate to the output folder. For example, although the code in lines 112 and 113 should succeed in getting the path to the desktop directory on most Windows and Linux computers, there are some unusual circumstances in which the desktop directory can't be identified this way. Also, in some circumstances, it is possible that the program might not have adequate rights to create the output folder.
- Wrapping these statements in an exception handler provides a fail-safe for this routine, and provides helpful information to a user encountering the problem.

b) Examine the exception-handling code starting in line 132.

```
132     try:
133         # Save the results to a log file.
134         write_file = open(log_filename, "w")
135         write_file.write("Results for {}:\n\n".format(file_name))
136         for line in self.results_list:
137             write_file.write(line + "\n")
138         write_file.close()
139         print("  Log file saved to " + log_filename)
140
141     except PermissionError:
142         # What happens when file can't be saved due to a permissions error.
143         print("  Can't save the results file due to a permissions error.\n" +
144               "  The following file or the directory that contains it may be locked:\n  " +
145               os.path.join(output_folder, log_filename))
146
147     except:
148         # What happens when any other type of exception is thrown.
149         print("  Can't save the results file:\n  " +
150               os.path.join(output_folder, log_filename))
151
152     else:
153         # What happens only if log file was successfully saved.
154         try:
```

- The try statement protects the code in lines 134 through 139, where the output log file is saved.
- The `except:` block in lines 141 through 145 will execute if there is an exception due to file permissions. This exception handler can be specific about the fact that there was a problem related to file permissions.
- If an exception not due to file permissions occurs, the more general handler in lines 147 through 150 will execute. This is a more general handler, so it provides less specific information about the nature of the problem.

c) Examine the block starting in line 152.

```
152     else:
153         # What happens only if log file was successfully saved.
154         try:
155             # Attempt to copy to archive file.
156             copyfile(log_filename, log_archive_filename)
157
158         except:
159             print("Can't create the archive file:\n  " +
160                   os.path.join(output_folder, log_archive_filename))
161
162         else:
163             print("  Archive file saved to " + log_archive_filename)
164
165     # List files currently in the output folder
166     print("\nThe {} output folder now contains:".format(output_folder))
```

- The `else:` block executes only if the entire `try:` block succeeded and no `except:` blocks had to execute.
- Nested under the `else:` block is another complete `try ... except ... else` handler, in lines 154 through 163. This nested code tries to copy the archive file, and provides its own exception handler should something go wrong.

3. Run the program to create a log file.

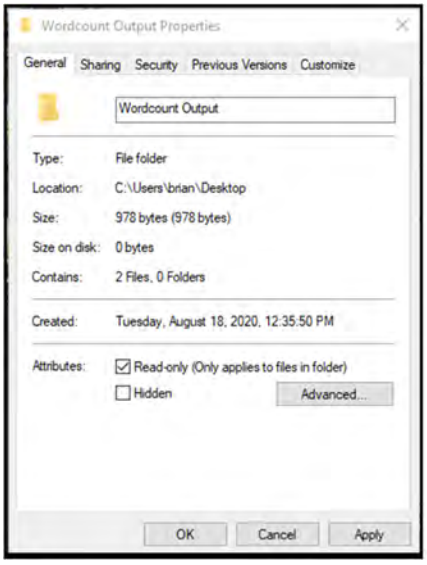
a) In the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.

b) At the prompts, enter ***bartleby.txt***, ***y***, and ***y***

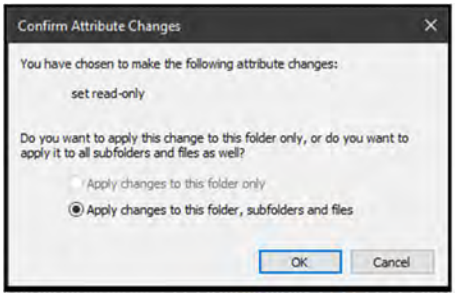
The program runs, producing a log file named **bartleby_results.txt**, as well as a backup file. The next time the program processes **bartleby.txt**, it will have to replace the **bartleby_results.txt** log file.

4. Test the exception-handling code by marking the log file as read-only.

a) On the desktop, right-click the **Wordcount Output** folder, and select **Properties**.
b) In the **Attributes** section, check the **Read-only** attribute as shown.

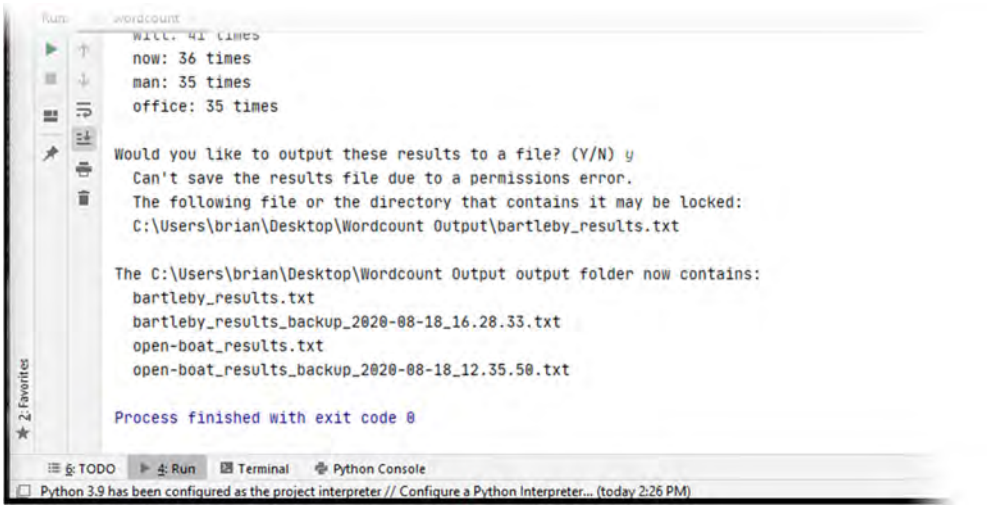


You may have to select the check box twice to change it to a check mark.
c) Select **OK**.



d) Select **OK**.
The output folder and files it contains will all be marked as read only.
e) In PyCharm, in the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.
f) At the prompts, enter ***bartleby.txt***, ***y***, and ***y***

g) Examine the Run console.



- Because the old `bartleby_results.txt` file is marked read only, it can't be opened for writing, and it produced a `PermissionError` exception.
- Because the `PermissionError` exception is handled, this problem did not cause the program to end abnormally. It simply reported the problem to the user, skipped past any remaining statements in the `try:` block, and resumed with the next block after the `try:` block and its exception handlers, producing a list of files currently in the output folder.

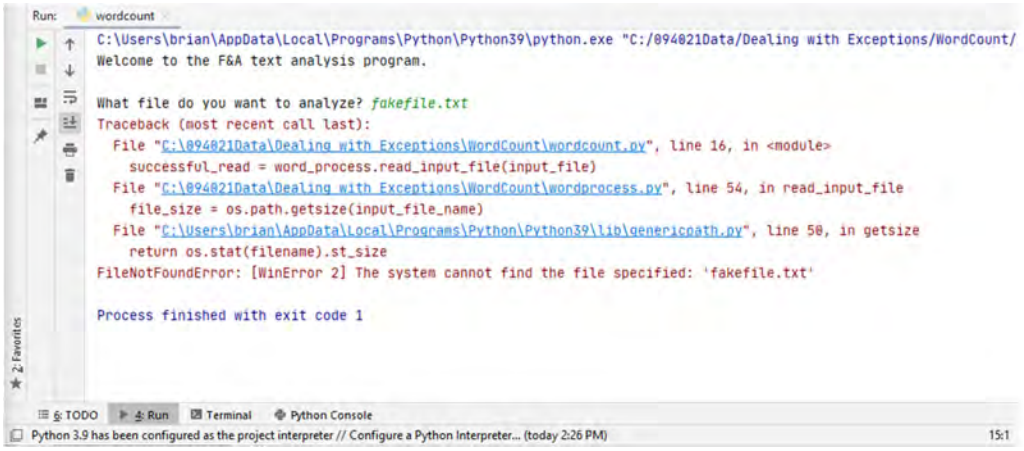
5. Revert the output folder's read-only attribute.

a) On the desktop, right-click the **Wordcount Output** folder, and select **Properties**.
b) In the **Wordcount Output Properties** dialog box, uncheck the **Read-only** attribute, and select **OK**.
c) In the **Confirm Attribute Changes** dialog box, select **OK**.

6. Verify that an exception handler would be useful in the routine that reads the input file.

a) In PyCharm, in the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.

b) At the prompt, enter ***fakefile.txt***



- This problem could have been avoided if the code included an error prevention routine (e.g., using `os.path.isfile` to check for an existing file before trying to open it) or by using `try ... except` to handle the exception.
- Because the exception is not handled by the WordCount program, the Python interpreter reports it to the user. Often, this way of dealing with exceptions is harsh and potentially confusing to an end user.

7. Wrap the code that reads the input file within a `try ... except` handler.

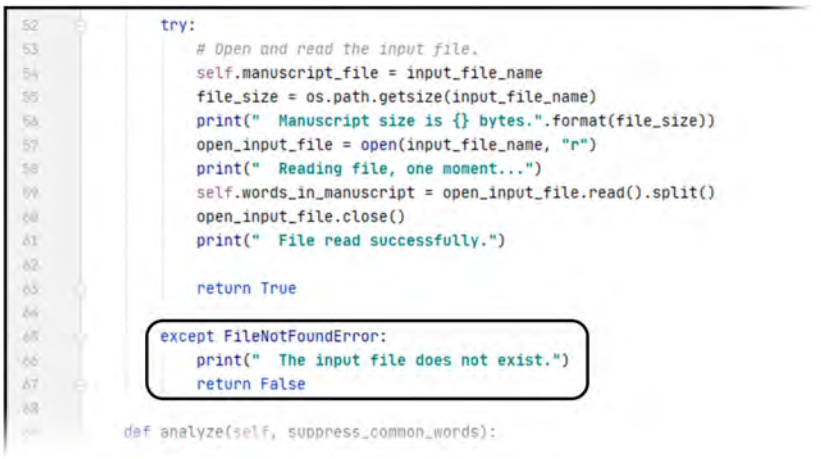
a) Position the insertion point at the end of line 51 and press **Enter**.

b) Type ***try:***

c) Select the statements in lines 53 through 63, and press **Tab** to indent them as shown.

d) Position the insertion point at the end of line 63, and press **Enter** twice.

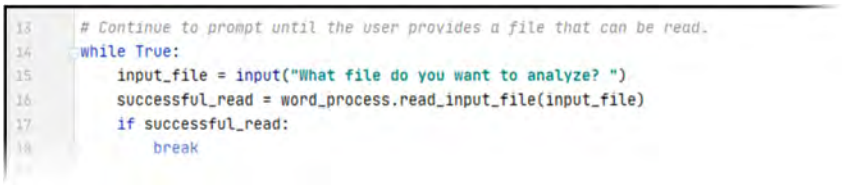
e) Enter the code as shown.



- If the code under `try:` executes without causing an exception, the `read_input_file` method will return `True`.
- If any statement in the `try:` block causes an exception, execution of the `try:` block will cease with that statement. If the exception is a `FileNotFoundError`, then execution of the code will resume at line 66, printing a message (line 66) and causing the `read_input_file` method to return `False` (line 67).

8. Identify what the main script does when the `read_input_file` method returns `False`.

a) In the **Project** pane, double-click **wordcount.py**.
b) Examine the code that calls the `read_input_file` method.



- Line 14 begins a loop that will be used to continue prompting the user for input until an acceptable value is entered.
- Line 15 places the user's input into a variable named `input_file`.
- Line 16 places the results of calling the `read_input_file` method (`True` or `False`) into a variable named `successful_read`.
- If `successful_read` is `True`, then the `break` statement in line 18 executes, exiting the `while` loop. Otherwise (not a successful read), the routine loops again, prompting the user to enter another file name.

9. Test the new exception handler.

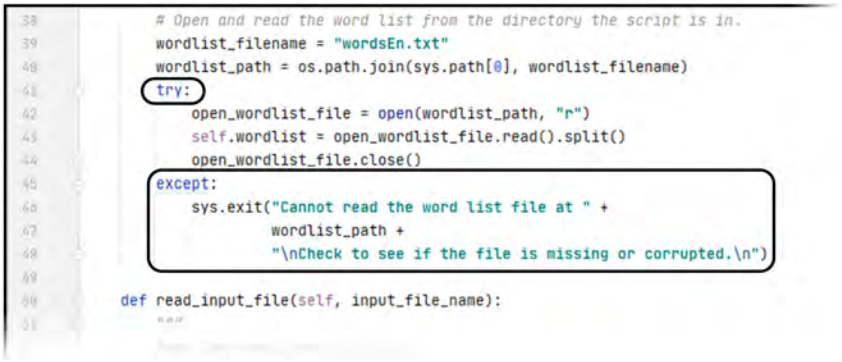
a) In PyCharm, in the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.
b) At the prompt, enter ***fakefile.txt***
The read operation fails since this file doesn't exist. You are prompted again to enter a file name.
c) At the prompt, enter ***small.txt***
The file is read successfully, and the input loop is exited.
d) In the **Run** window, close the **wordcount** tab, and select **Terminate**.

10. Add code to handle an exception when reading the word list file.

a) Select the **wordprocess.py** editor tab.
b) Starting in line 27, examine the operations performed in the `__init__` function.

- When a new `WordProcess` object is created, various attributes (`wordlist`, `words_in_manuscript`, and so forth) are initialized in lines 32 through 36.
- In lines 42 through 44, various file operations are performed to load the word list from wordsEn.txt.
- The wordsEn.txt file is an essential component of the WordCount program. It is always expected to be found in the same directory as the script it's loaded from, which is obtained by getting the value in `sys.path[0]` in line 40.
- There is a possibility, of course, that the WordCount program may not be installed correctly on another user's computer, and the word list file might be missing. If this happens, it would be helpful to inform the user what the problem is. Also, the program should not be allowed to run until the problem is corrected, since it is required in order to function.

c) Indent the code in lines 42 through 44 and add the code shown here.

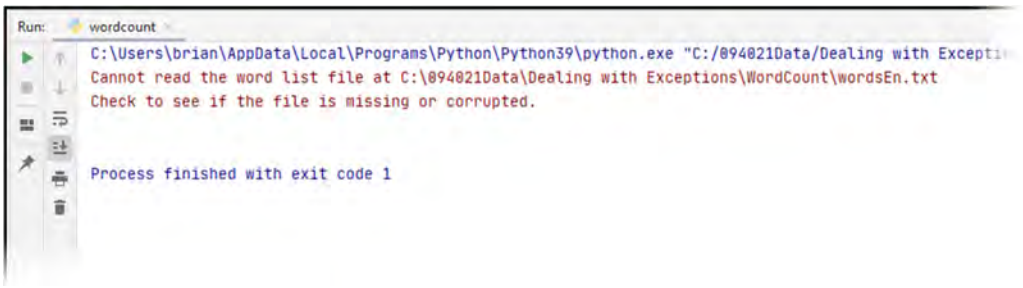


- Lines 41 through 44 try to read the word list file.
- If there is an exception when reading the word list, the program is exited in line 46, explaining that the program can't run without the word list file.

11. Test the exception handler.

a) In the **Project** pane, right-click **wordsEn.txt**, and select **Show in Explorer**.
b) In the **File Explorer** window, rename the **wordsEn.txt** file as ***XwordsEn.txt***
This will enable you to simulate what would happen if the file is not found.

c) In PyCharm, in the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.



The program immediately exits with a code 1. The message explains to the user what went wrong, providing a clue on how to resolve it.

d) In the File Explorer window, rename the **XwordsEn.txt** file back to ***wordsEn.txt***

e) In PyCharm, in the **Project** pane, right-click **wordcount.py** and select **Run 'wordcount'**.

The program starts successfully, and prompts the user for a file to analyze.

f) In the Run window, close the **wordcount** tab, and select **Terminate**.

12. **What are the advantages of handling an exception yourself, rather than just letting the exception be reported by the Python interpreter?**

13. **Why is it a good idea to specify the type of exception you want to handle, rather than just handling all exceptions in one `except` statement?**

TOPIC B

Raise Exceptions

Aside from leveraging Python's built-in exception handlers, you may find it beneficial to customize this exception handling process. Python makes it easy to do this, and in this topic, you'll raise your own unique exceptions.

Raise

Instead of letting Python produce an exception on its own, you can do so yourself with the `raise` statement. The `raise` statement can force Python to encounter an exception, which you can then handle. The syntax for the `raise` statement refers to the specific exception you want to raise, and then takes an optional exception argument within parentheses:

```
raise NotInRange("Number is not in range!")
```

This code raises a `NotInRange` exception and passes in the string as an argument.

You can raise an exception anywhere in your code, but you can also place the `raise` statement within a `try` branch in order to then catch it with an `except` branch:

```
try:
    raise NotInRange("Number is not in range!")
except NotInRange as arg:
    print("Error: ", arg)
```

Custom Exceptions

Very rarely will you need to raise one of Python's built-in exceptions. Instead, you'll most likely raise exceptions that are specific to your code. This way you can cover any program failures that Python doesn't specifically define, so that it's easier to debug your code. In order to do this, you need to first create a custom exception.

To start, you need to create your own exception class. This class should inherit the general `Exception` class or one of its subclasses. Within your custom exception class, you can define any sort of parameters you want. However, the purpose of exception classes is usually just to provide information about an error in code, so it's best to keep them simple.

For the following example, assume that you've created a number guessing game. You take a user's guess between 1 and 10, and let them know whether or not they've guessed the correct number. However, the user might enter a number that isn't between 1 and 10. How would you handle this situation? One way is to categorize this unwanted input as an exception. The following code block creates a custom exception class for this very purpose:

```
class NotInRange(Exception):
    def __init__(self, value):
        self.value = value
    def __int__(self):
        return self.value
```

The custom exception `NotInRange` inherits from the general class `Exception` and has its own initialization method. It takes one argument, and the `__int__()` method returns the argument as an integer (this will be the user's input).

This other section of code raises the exception when it's needed and handles everything in a `try ... except` statement:

```
try:
    user_guess = int(input("Guess a number between 1 and 10: "))
```

```
    if user_guess < 1 or user_guess > 10:
        raise NotInRange(user_guess)
except NotInRange as arg:
    print("Error: The number {} is not between 1 and 10.".format(arg))
```

If the user's guess is not between 1 and 10, the `NotInRange` exception is raised with the user's guess as the argument. The `except` branch prints an error message back to the user with their guess, assuming the guess was not between 1 and 10.

A simple program like this can also control for bad input with a few conditional statements. However, more complex programs will benefit from custom exceptions because they're easy to reuse and track.

Note: Custom exceptions are very useful when you are creating code libraries, which contain functions and classes you developed to be used by other programmers. (For example, in this course, you have used other people's code when you have written `import` statements in your programs.) When you create code libraries, your direct "customer" is another programmer—not an end user. When a code library implements custom exceptions, they provide a "hook" that other programmers can take advantage of to handle exceptions in a way that is appropriate for the program they are developing based on your code.

Guidelines for Raising Exceptions

Use the following guidelines to help you raise exceptions in your Python programs.

Raising Exceptions

When raising exceptions:

- Use the `raise` statement to force Python to encounter an exception.
- In a `raise` statement, provide the exception class and, optionally, an argument.
- Consider placing `raise` statements in `try ... catch` statements so you can handle the exception.
- Raise your own custom-defined exceptions, and not Python's built-in exceptions.
- Use custom exceptions that are unique to your program.
- Define your own exceptions as a class that inherits from the `Exception` class or one of its subclasses.
- Keep this custom class simple and informational.
- Use custom exceptions in larger, more complex programs to more easily track unwanted behavior.

ACTIVITY 7-2
Raising Exceptions

Data File

C:\094021Data\Dealing with Exceptions\WordCount\large_file.txt

C:\094021Data\Dealing with Exceptions\WordCount\empty.txt

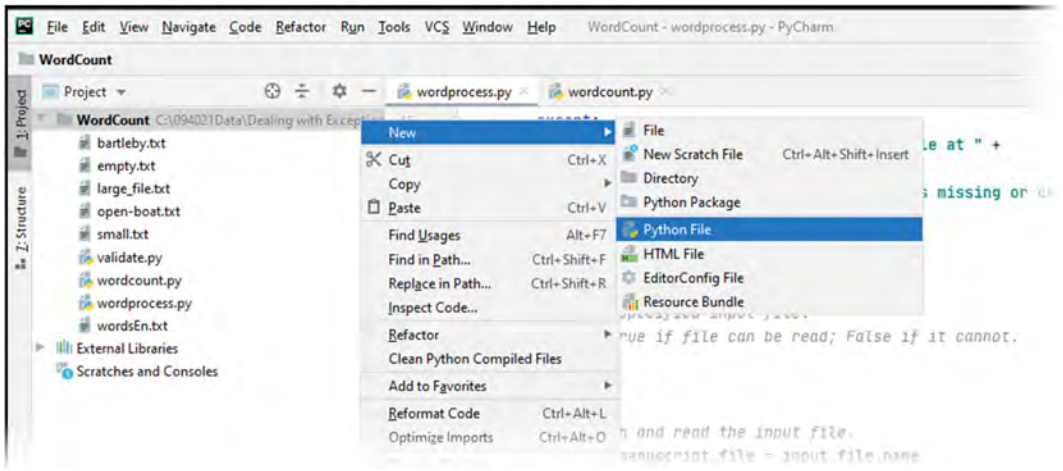
Scenario

In the WordCount program, you want to add exception handlers to deal with situations in which the input file is empty or is larger than 10 megabytes. While Python's built-in `Open` method raises a `FileNotFoundError` exception when you try to open a file that doesn't exist, it doesn't implement exceptions for empty files or files larger than 10 megabytes. In this activity, you'll define your own custom exceptions to account for these situations.

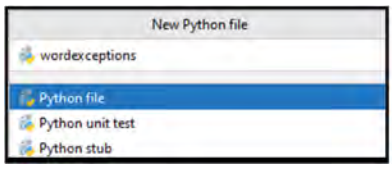
---

1. Create a custom exception module.
   a) In the **wordprocess.py** tab, scroll as needed to view the `read_input_file` method, which starts at line 50.

      You have already added an exception handler to deal with `FileNotFoundError` exceptions.
   b) In the **Project** pane, right-click **WordCount** and select **New→Python File**.



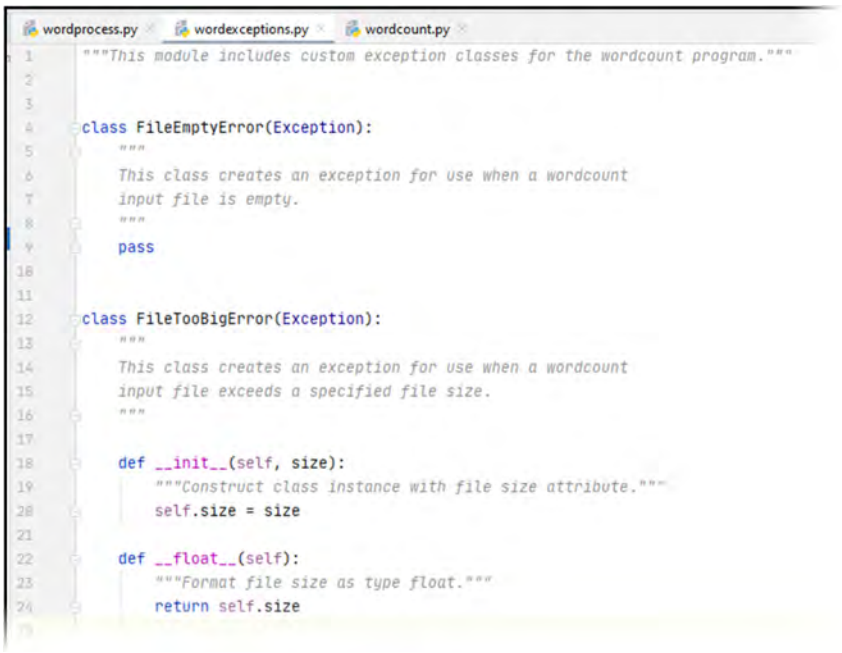
   c) In the **New Python file** dialog box, in the **Name** text box, type ***wordexceptions*** and press **Enter**.



   d) Verify that **wordexceptions.py** has opened in the editing area.

2. Create two custom exception classes.

a) Enter the code as shown.



Two custom exception classes are defined in this script.

- Line 4 creates the class `FileEmptyError` by inheriting from Python's general `Exception` class.
- A class is not allowed to be empty, so line 9 uses a `pass` statement as a placeholder. The `pass` statement will do nothing when executed.
- This approach enables you to define a barebones custom exception.

A slightly more complex custom exception is defined starting in line 12.

- Line 12 creates the class `FileTooBigError` by inheriting from Python's general `Exception` class.
- Lines 18-20 construct an instance of the custom exception class, taking in a file size argument that is assigned to the class's size attribute when the exception is raised. In the exception handler, this value can be read to determine the size of the file that was too large.
- Lines 22-24 implement a method that returns the size attribute as a value of type `float`.

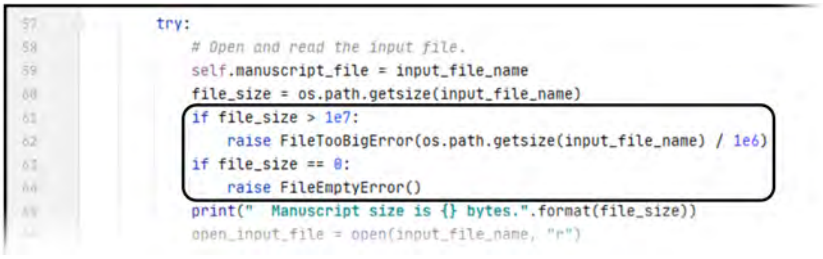
3. Import the new custom exception.

a) Select the **wordprocess.py** tab.
b) To the left of the `import` statement in line 5, select the **+** button to show all of the `import` statements.
c) Position the insertion point at the end of line 8, and press **Enter**.
d) Type **`from wordexceptions import FileEmptyError, FileTooBigError`**
This imports both exception classes from the wordexceptions.py file.

4. Raise the new custom exceptions in your code.

a) Position the insertion point at the end of line 60.
b) Press **Enter**.

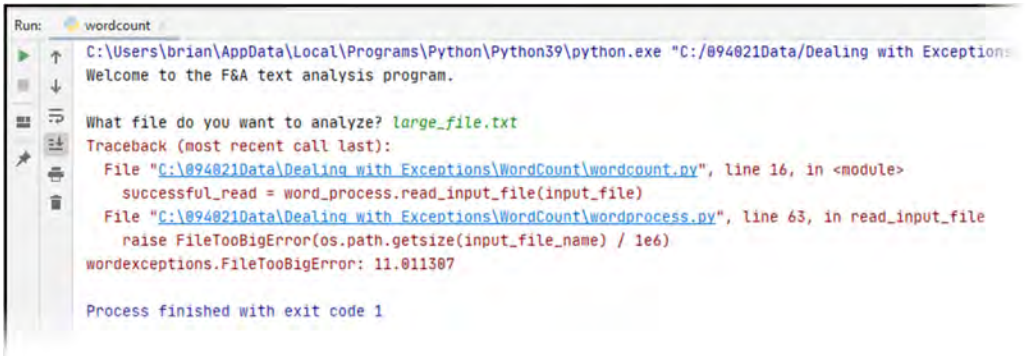
c) Enter the code shown here.



- If the size of the user's input file (`self.file`) exceeds 10 megabytes, then the `FileTooBig` exception will be raised, constructing an instance of the class and passing in the file size in megabytes. The argument passed into this class instance is the user's file size formatted into MBs.
- If the size of the file is zero bytes (empty file), then the `FileEmptyError` exception will be raised.
- If you were planning to handle the exception outside of the class (in wordcount.py, for example), you would be done working in wordprocess.py at this point. The exceptions will be raised, and if not handled within this class or in outside code that uses the class, then the exceptions will be reported by the Python interpreter.

5. Test your custom exception with bad input files.

a) Run **wordcount.py**.
b) For the input file, enter ***large_file.txt***
c) Verify that Python encountered your custom exception and reported the exception.



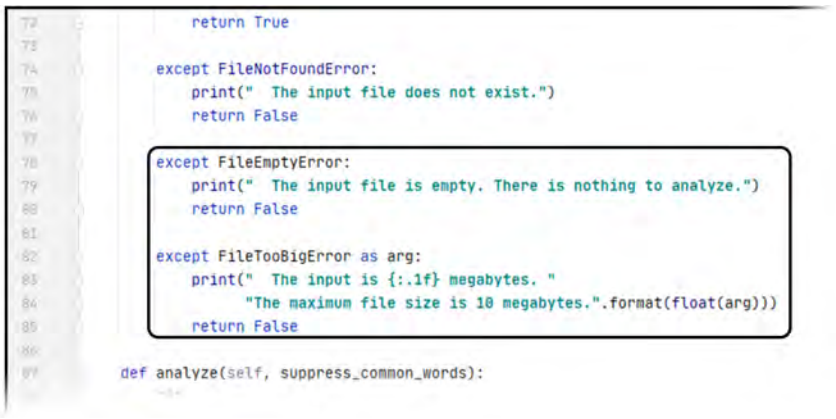
The exception is unhandled, so the program exits with a code 1.

d) Run **wordcount.py** again.
e) For the input file, enter **empty.txt**.
Again, the exception is unhandled, so the program exits with a code 1.

6. Write code to handle the custom exceptions.

a) Position the insertion point at the end of line 76.
b) Press **Enter** twice.

c) Type the code shown here.



```
            return True

        except FileNotFoundError:
            print("  The input file does not exist.")
            return False

        except FileEmptyError:
            print("  The input file is empty. There is nothing to analyze.")
            return False

        except FileTooBigError as arg:
            print("  The input is {:.1f} megabytes. "
                  "The maximum file size is 10 megabytes.".format(float(arg)))
            return False

    def analyze(self, suppress_common_words):
```

- If one of these exceptions occurs, then a message will be shown to the user and `read_input_file` will return `False`.
- In the `FileTooBigError` exception handler, the message will display the size of the file that was too large.

7. Test your custom exception handler with a large input file.

a) Run **wordcount.py**.
b) For the input file, enter ***large_file.txt***
c) Verify that Python encountered your custom exception and printed the warning you defined in your exception handling code.

8. Clean up the workspace.

a) In the Run window, close the **wordcount** tab, and select **Terminate**.
b) Close the project, and exit PyCharm.

Summary

In this lesson, you dealt with the exceptions that are inevitable in any programming project. With the proper handling techniques, you'll be able to minimize program failures and improve your debugging process.

What are some of the most common errors that you make while programming? Do you think you'll make more or less of these errors when coding in Python?

What kinds of exceptions might your apps encounter? Will you need to raise your own custom exceptions?

	**Note:** To learn more about the world of Python, check out the LearnTO **Become a Member of the Python Community** presentation from the **LearnTO** tile on the CHOICE Course screen.
	**Note:** Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Course Follow-Up

Congratulations! You have completed the *Introduction to Programming with Python®* course. You have successfully applied the fundamental elements of the Python programming language to create a desktop application. Python is a versatile language and understanding the basics will help you eventually develop more complex apps for desktop, mobile, and web platforms.

What's Next?

The next course in this series is *Advanced Programming Techniques with Python®*. For programmers, cybersecurity is a major concern, so you might also consider taking *CertNexus Certified Cyber Secure Coder® (Exam CSC-210)*.

You are encouraged to explore Python further by actively participating in any of the social media forums set up by your instructor or training administrator through the **Social Media** tile on the CHOICE Course screen. You may also wish to explore the official Python documentation website at **https://docs.python.org/3/** for more information about the language.

A Major Differences Between Python 2 and 3

While Python® 3 is very similar to its Python® 2 predecessor, there are some key differences:

- Python 3 is not backward compatible with Python 2. Application code written in Python 2 must be ported to Python 3 before it can run from the Python 3 interpreter.
- Python 3 currently has support for most, but not all, of the libraries associated with Python 2.
- The `print` statement in Python 2 is now a `print()` function in Python 3.
- In Python 2, dividing integers (`/`) always resulted in an integer. In Python 3, dividing integers this way results in a float.

  For example: In Python 2, `3 / 2` would result in the integer `1`. In Python 3, it results in the float `1.5`.
- The `raw_input()` function in Python 2 was renamed to `input()` in Python 3.
- In Python 2, strings could be formatted in both Unicode and non-Unicode. In Python 3, all strings are by default Unicode.
- Raising exceptions in Python 3 requires parentheses around the exception argument, whereas Python 2 does not.

Some of these changes have been backported to more recent Python 2 versions such as Python 2.6 and 2.7.

Note: For more information about the differences between Python 2 and 3, navigate to **https://docs.python.org/3/whatsnew/3.0.html**.

B Python Style Guide

Appendix Introduction

Whether you write and maintain your own software or share that responsibility with other developers, it's important that you follow best coding practices. Python® has an official style guide that will help you write clean, well-formatted code that makes it easier to understand the purpose and function of that code.

TOPIC A

Write Code for Readability

In terms of style, the primary factor of well-written Python code is its readability. The easier it is to read a piece of code, the quicker and easier it is for a programmer to grasp the intent behind it. This makes maintaining, debugging, modifying, and extending code much less of a hassle. So, you'll explore how to write Python code so that your programs will be as easy to read as possible.

PEP

The Python Software Foundation maintains a list of various Python Enhancement Proposals or PEPs. A ***PEP*** is a public document that either describes some element of Python's design, or proposes a new feature be added to the language. Community members can write a PEP based on the existing standard, and, if approved, it will become officially part of Python's documentation. Python creator Guido van Rossum has the final say on whether or not a PEP is approved.

Python has many active PEPs, each of which is numbered. One of the most useful PEPs is ***PEP 8***, officially titled "Style Guide for Python Code." This style guide prescribes a wide array of best practices for writing code in Python to enhance readability, as decided by both van Rossum and community consensus. While adhering to these conventions is not required for programmers, the Python community generally follows them.

One of the most overarching ideas in PEP 8 is the importance of consistency. However, there are circumstances when you need to sacrifice consistency for the sake of your code. This is embodied in the expression, "a foolish consistency is the hobgoblin of little minds." In other words, you should try to follow the conventions in PEP 8, but you should know when to break these conventions. If a convention makes your code harder to read, avoid it. Circumstances like these are more likely to pop up in programs that need to maintain backward compatibility with older versions of Python. Ultimately, it comes down to your own personal judgment.

Note: For a listing of all PEPs, navigate to **https://www.python.org/dev/peps/**.

Note: For the full text of PEP 8, navigate to **https://www.python.org/dev/peps/pep-0008/**.

Code Layout

The following PEP 8 guidelines prescribe how to lay out your source code:

- Indent code with spaces instead of tab characters. In IDEs like PyCharm, pressing the **Tab** key automatically adds the required number of spaces.
- The required number of spaces is four at each level of indentation.
- The maximum suggested line length is 79 characters. Some leeway is given for individuals or teams that agree to increase the limit.
- Wrap code on the next line to avoid exceeding the line limit.
- Align wrapped code rather than letting it spill over to the beginning of the next line. For example:

```
my_input = input("This is the first line of the string."
                 "This wrapped text should be aligned.")
```

- Top-level functions and class definitions should be separated by two blank lines.
- Methods inside classes should be separated by one blank line.
- Use blank lines to separate logic groupings of code.

- Place `import` statements on separate lines.
- Place `import` statements at the beginning of a module, below any opening docstrings.
- Import modules in the following order:
  1. Standard library modules
  2. Third-party modules
  3. Local or application-specific modules
- Place global variables and constants after any `import` statements.
- Use general and selective imports and avoid universal imports.
- Be consistent with your use of single or double quote characters in string literals.

Whitespace

The following guidelines apply to your use of whitespace in Python code:

- Avoid whitespace after opening parentheses/brackets and before closing parentheses/brackets.

  Correct: `print("There is no space between quote and parentheses.")`

  Incorrect: `print( "There are spaces between quote and parentheses." )`
- Avoid whitespace before commas that separate values. Place them only *after* commas.

  Correct: `print(a, b, c)`

  Incorrect: `print(a , b , c)`
- Avoid whitespace before and after colons that separate range slice values.

  Correct: `my_list[0:5]`

  Incorrect: `my_list[0 : 5]`
- Avoid whitespace after the function name in a function call.

  Correct: `str(3.14)`

  Incorrect: `str (3.14)`
- Include one space before and after the = in a variable assignment.

  Correct: `a = 1`

  Incorrect: `a=1`
- Include one space before and after any operators.

  Correct: `if a < b:`

  Incorrect: `if a<b:`
- In a complex expression, consider using whitespace in operations with high priority, and no whitespace for operations of lower priority.

  Correct: `a = b*c - d`

  Incorrect: `a = b*c-d`
- Try to keep different statements on different lines, rather than joining them on the same line.

Comments

The following guidelines apply to your use of comments, whether block or inline.

- It's better to have no comments than comments that contradict the code or each other.
- Update comments as the code changes.
- Comments should usually be formatted as complete sentences, beginning with a capital letter and ending in a period.
- Short comments can end without a period.
- Use Strunk & White's *The Elements of Style* as an English language style guide.

- Always write comments in English, unless you're absolutely sure that only people who understand your language will be reading your code.
- Apply block comments above associated code.
- Indent these block comments on the same level as the associated code.
- Separate the `#` symbol from the first letter with a single space.

  Correct: `# Increment the loop's counter.`

  Incorrect: `#Increment the loop's counter.`
- Use inline comments sparingly.
- If you use inline comments, separate the comment from the code by at least two spaces.

  Correct: `count += 1 # Increment the loop's counter.`

  Incorrect: `count += 1 # Increment the loop's counter.`
- Don't add inline comments that state the obvious.

Docstrings

Docstring style is mentioned in PEP 8, but is actually elaborated on in ***PEP 257***: "Docstring Conventions." Beyond just readability, formatting docstrings in accordance with PEP 257 can actually make using docstring tools much easier.

- All modules should have docstrings.
- Module docstrings should go at the beginning of source code, before any `import` statements.
- All functions, public methods, and classes that may be imported by another program should include docstrings.
- Use the same English language conventions in docstrings that you would in comments.
- Always use triple quotes with the double quote character in docstrings.

  Correct: `"""This program counts the number of words in a file."""`

  Incorrect: `'''This program counts the number of words in a file.'''`
- Use one-line docstrings for simple cases.
- The opening and closing quotes go on the same line for a one-line docstring.
- No blank lines before or after a one-line docstring.
- A function/method docstring should *prescribe* the function/method, not *describe* it.

  Correct:

```
def my_func(x, y):
    """Calculate user's values."""
```

  Incorrect:

```
def my_func(x, y):
    """This function calculates a user's values."""
```
- Use multi-line docstrings for more complex objects.
- In a multi-line docstring, place the closing and opening quotes on their own lines.
- For a module multi-line docstring, list the classes, exceptions, functions, and public methods that are exported. Describe these objects briefly.
- For a class multi-line docstring, list any public methods, instance variables, inheritance, and summarize the class's behavior.
- For a function/method multi-line docstring, list any arguments, return values, and other important information about the object.
- Add a blank line between a class's docstring and the first method.
- Indent docstrings directly within the object they prescribe.

Note: For the full text of PEP 257, navigate to **https://www.python.org/dev/peps/pep-0257/**.

Names

The following are PEP 8's naming conventions.

- Use these conventions for new modules, but retain any agreed upon style that is already in place for existing code.
- Don't use "l" and "I" as single-character variable names, as these are often confused in some fonts.
- The same goes for "O," which is often confused for a zero.
- Module names should be short and in all lowercase. Underscores are allowed if they improve readability.

  Correct: `wordcount`

  Incorrect: `Word Counting Module`
- Class names (including custom exceptions) should use the CapWords format.

  Correct: `class MyClass`

  Incorrect: `class my_class`
- Function/method names should be in lowercase, with underscores between words.

  Correct: `my_function()`

  Incorrect: `MY FUNCTION()`
- Use `self` as the first argument of an instance method.
- Use `cls` as the first argument of a class method.
- Add a leading underscore to private methods/variables.
- Variable names should be in lowercase, with underscores between words.

  Correct: `my_variable = 1`

  Incorrect: `MyVariable = 1`
- Constant names should be in uppercase, with underscores between words.

  Correct: `MY_CONSTANT = "Blue"`

  Incorrect: `my_constant = "Blue"`

Miscellaneous Best Practices

Lastly, PEP 8 has some general recommendations for Python programmers.

- Write code so that it can be used with any of the available Python interpreters, not just CPython.
- Use `is not` as an identity operator instead of `not ... is`.
- When creating custom exceptions, inherit from existing exception classes in a way that benefits handling the exception.
- Handle specific exceptions whenever possible, rather than all exceptions at once.
- Place only the minimum amount of code needed in the `try` branch of an exception handler.
- Be consistent with return statements in functions; i.e., make them all return a variable or make them all return an expression, not a mix of both.
- Don't place too much trailing whitespace in string literals. Some IDEs will truncate these string literals.

C Mapping Python Course Content to Python Institute Certification Exams

Logical Operations is pleased that our Python® curriculum can be an important part of your preparation for professional certifications from our partner The Python Institute. These certifications include *Certified Entry-Level Python Programmer (exam PCEP-30-0x)*, *Certified Associate in Python Programming (exam PCAP-31-0x)*, and *Certified Professional in Python Programming 1 (PCPP-32-10x)*. For specific requirements for each exam, please visit The Python Institute website at **https://pythoninstitute.org/**.

To assist you in your preparation for the Python Institute exam of your choice, Logical Operations has provided reference documents that indicate where current exam objectives are covered in the Logical Operations Python curriculum series, *Introduction to Programming with Python®* and *Advanced Programming Techniques with Python®*.

The exam-mapping documents are available from the **Course** page on CHOICE. Log on to your CHOICE account, select the tile for this course, select the **Files** tile, and download and unzip the course files. The mapping references will be in a subfolder named **Mappings**.

Best of luck in your exam preparation!

Solutions

ACTIVITY 1-5: Preventing Errors

7. **Why does the program print the wrong value? How can you fix this statement to get the program to print the right value?**

   A: The print statement prints `user_input`, which is the first input. You can change the statement to say `print(common_word)` to get the program to print the second input.

8. **What kind of error is this? Why?**

   A: This is a logic error. Python is able to interpret and execute the code correctly, but the code doesn't do what the programmer intended.

ACTIVITY 2-1: Processing Strings and Integers

4. **Why can you concatenate `size_query`, but must use interpolation for `size_in_bytes`?**

   A: Only strings can be concatenated, and since `size_in_bytes` is an integer, it can use interpolation so that it's formatted properly in the string.

ACTIVITY 2-2: Processing Mixed Number Types

3. **Why did `size_in_gigabytes` turn into a float instead of an integer?**

   A: By default, the result of all regular division operations in Python 3 is always a float.

4. **Assume you want to convert `size_in_gigabytes` back to megabytes by multiplying `size_in_gigabytes` by `1000`. What would the result of this operation be and why?**

   A: The result would be `1.0`. In an operation with a mix of integers and floats, the result always widens to a float.

ACTIVITY 3-1: Processing Ordered Data Structures

8. **The word "test" still exists in the list. Why has it not been removed?**

   A: The `remove()` operation will always remove the first instance of an item in a list, while leaving the rest. The word "test" was in the list twice.

ACTIVITY 4-1: Writing Conditional Statements

7. **When you input "no" to the third question, why does it output nothing to the console?**

   A: The condition only handles the single characters "y" or "n". There is no explicit `else` statement that handles every other kind of input besides "y" and "n", so the program just continues on.

ACTIVITY 5-4: Importing and Using a Module

4. **In line 106 where you call `expanduser()`, why do you need to include `os.path` before it?**

   A: Because you used a general import, which requires you to explicitly reference the module when you call one of its methods.

5. **Which of the following statements would allow you to selectively import `expanduser()` so you don't need to reference the `os.path` module?**

   - ○ import expanduser from os.path
   - ○ import os.path with expanduser
   - ◉ from os.path import expanduser
   - ○ for os.path import *

ACTIVITY 7-1: Handling Exceptions

12. **What are the advantages of handling an exception yourself, rather than just letting the exception be reported by the Python interpreter?**

    A: Answers may vary. If you plan on just printing an error message, you might be able to tell your users (or testers) what went wrong better than Python. You can also do more than just print an error; you can execute some other code to keep your programming running, despite the exception. Additionally, handling exceptions makes it easier to identify and resolve bugs in your code, especially in larger programs.

13. **Why is it a good idea to specify the type of exception you want to handle, rather than just handling all exceptions in one `except` statement?**

A: When you specify the type of exception to handle, it'll be easier for you to identify the problem and you can be more specific in the type of response you apply.

Glossary

access mode
The argument that determines how a file will be opened and what kind of operations can be performed on the file.

argument
A value passed into a function that the function uses when called.

arithmetic operator
An operator that performs basic mathematics on number variables, including addition, subtraction, multiplication, and division.

attributes
Variables within a class's scope that are shared among the objects the class creates.

bytecode
Intermediary code designed to be efficiently executed by an interpreter, usually through a virtual machine.

calling
Invoking a function, usually by providing the values of the arguments that the function uses (if it has any).

class
An object used to combine similar code and attributes to create other objects.

class method
A method that typically takes a `cls` argument and that is attached to the class itself, not any particular instance.

class variable
A variable defined in a class that can be shared by all instances of the class.

command line argument
A form of input that the user specifies at a command line, before the program is executed.

command shell
Software that allows a user to directly interact with a computer's operating system by executing commands.

comparison operator
An operator that tests the relation between two values.

compiler
Software that translates programming code into machine code before it can be executed.

concatenation
The process of combining strings end-to-end.

conditional statement
An object that requires the program to make a decision based on certain conditions.

constant
An identifier with a value that does not change.

CPython
The default and most widely used Python interpreter.

data type
The way in which a variable is classified and stored in memory.

debugger
A software tool that helps a programmer detect and test errors in a program's source code.

decorator
An object in Python that can modify the definition of a function, method, or class.

dictionary
An unordered, mutable data structure that has key-value pairs.

docstring
Documentation placed within source code and formatted as a string literal. Reveals what a segment of code will do.

error
An incorrect or unintended result after the execution or attempted execution of code.

escape character
See *escape code*.

escape code
A way to represent characters in a string literal that can't be represented directly.

exception
When a programming language is able to understand code, but cannot complete the code's action due to numerous possible circumstances.

exception handling
The process of anticipating and responding to program exceptions.

float
A less precise version of a decimal number that can use scientific notation to represent a very large or very small number.

floating point number
See *float*.

function
A block of reusable code that performs a specific task.

general import
The process of importing all of a module's objects into a program.

global variable
A variable that is useable everywhere within a program's source code.

IDE
(integrated development environment) Software that assists programmers in writing code. Typically comes with a source code editor and a debugger.

identifier
The name attached to an object, like a variable, class, function, etc.

identity operator
An operator that checks to see if one object points to the same address in memory as another object.

IDLE
The default Python IDE that includes a command shell.

immutable
A description indicating that an object's value cannot change after it is created.

inheritance
The process of extending a class's capabilities into another class.

instance construction
The process of calling a class and creating a specific object from that class.

instance method
A method that typically takes a `self` argument and that is attached to a particular instance of the class.

instance variable
A variable that is local to a method and only applies to the current instance of the class.

integer
A whole number that can be either positive or negative and can't include a decimal.

interactive mode
When the Python interpreter presents a command prompt to the user for immediate code execution.

interpreter
Software that executes programming code itself, rather than translating it into machine code first.

list
A mutable sequence that can hold different types of values.

local variable
A variable that is bound to a specific object, such as a function, and cannot necessarily be used outside of that object.

logic error
When a program has unintended or unpredictable results due to improperly implemented code.

logical operator
An operator that connects multiple values together so they can be evaluated.

long integer
An integer that has an unlimited amount of precision. No longer applicable in Python 3.

loop
A statement that executes code repeatedly.

method
A function inside of a class.

module
A Python file that can be reused and imported into other Python files to take advantage of its functionality.

modulo
The arithmetic operation of finding the remainder in a division.

mutable
A description indicating that an object's value can change after it is created.

operand
The values that an operator evaluates.

operator
Objects that can evaluate expressions.

operator overloading
The process in which a class defines how it handles operators.

order of operations
The order in which operators in an expression are executed.

parameter
See *argument*.

PEP
(Python Enhancement Proposal) A public document that describes a design element of Python or proposes a new feature.

PEP 257
The official documentation that prescribes docstring conventions for Python code.

PEP 8
The official documentation that prescribes style guidelines for Python code.

private variable
A variable that cannot be used outside of a class.

project folder
A set of files used to develop a program in Python, typically stored within a directory that contains Python source code, configuration files that specify how the code should be processed to create the finished program, and other supporting files.

property
An attribute with getter, setter, and delete methods.

range
An immutable sequence of integers that is commonly used to iterate through a process.

range slice
A span of characters retrieved from a sequential variable like a string or list.

reserved word
A word that cannot be used as an object identifier because it has a fixed meaning.

scope
Where a particular object is valid in source code and where it is not.

selective import
The process of importing specific objects from a module into a program.

sequence
An ordered collection of elements.

set
An unordered, mutable data structure with individual entries that cannot be repeated.

slice
An individual character retrieved from a sequential variable like a string or list.

source code
The code written by a programmer in a programming language such as Python, which is then processed by an interpreter or compiler into code that can execute on the computer.

special method
Methods reserved by Python that perform certain tasks in a class.

standard library
The official collection of resources associated with a programming language.

string
A sequence of characters stored in memory.

string interpolation
The process of replacing variable placeholders in a string literal with their corresponding values.

string literal
A value that is enclosed in single or double quotation marks.

subclass
The class that is doing the inheriting.

superclass
The class that is being inherited from.

syntax
The rules that govern how you must write programming code for it to execute properly.

syntax error
When a programming language cannot understand a piece of code that the programmer has written.

tuple
An immutable sequence that can hold different types of values.

universal import
The process of importing all of a module's objects into a program so that calls do not need to directly reference the module.

Unix time
A measurement of time in which a computer calculates how many seconds have elapsed since the Unix epoch (00:00:00 on January 1st, 1970 [UTC]).

variable
A value that is stored in memory and is associated with an identifier or name.

Index

G

H

I

L

M

N