00000011

CHAPTER 3

Small Victories: Simple Python Programs

Learning Objectives

- List and describe the minimum development tools required to create Python programs
- Launch the Python interpreter in Interactive Mode
- Execute Python code in the REPL
- Execute a Python program from the command line
- Execute a Python program with Visual Studio Code
- Explain the differences between Python modules and packages
- Explain how the '+' operator behaves when applied to mixed types
- List and describe fundamental Python data types
- Use Python's print() and input() functions to perform console I/O operations
- Demonstrate your ability to perform type conversions on string and numeric types
- Explain the difference between syntax errors and runtime exceptions
- Use the try statement and the except clause to properly handle runtime exceptions
- Explain how Python modules and packages are loaded and executed
- List and describe the contents of the Python Standard Library
- Import and use standard library packages in your programs
- Install third-party packages with pip or pip3

INTRODUCTION

This chapter is all about *small victories*. Small victories give you the reassurance you so desperately need when taking your first tentative steps toward becoming a software developer. Small victories is about the planets beginning to move into alignment. Small victories is about your knowledge buckets beginning to fill. Small victories is all about the amazing feeling you get when you write and execute your first Python program. It's a great feeling!

You will learn a lot in this chapter. First, you learn how to enter and run short programs in the Python interactive interpreter or **R**ead, **E**valuate, *P*rint, *L*oop (REPL) environment. From there you'll create Python source files with a text editor and run them from the command line. Finally, I'll show you how to run and debug Python projects using Visual Studio Code.

Along the way you'll learn about the Python Standard Library, built-in functions, fundamental data types, and program structure and formatting. I'll show you how to perform console input and output (I/O) with the `print()` and `input()` functions. You'll learn proper identifier naming techniques and how to convert string values to numeric values and back again.

In this chapter you'll be using the Python interpreter, a text editor, and Visual Studio Code. If you haven't yet set up your baseline development environment I recommend you return to *Chapter 1: Part I Preliminaries: Baseline Development Environment*, and do so now. Also, I'll be conducting most of the operations in this chapter via a terminal. Be sure to write down the commands I use in your engineer's notebook.

The only tools you really *need* to create and run Python programs are a simple text editor and the Python interpreter, but you'll soon grow to appreciate an Integrated Development Environment like Visual Studio Code with its wide selection of plugins that provide helpful, convenient features. You can also interact directly with the Python interpreter running in interactive mode, also referred to as the *Read, Evaluate, Print, Loop* (*REPL*), which is a great way to learn about and experiment with fundamental Python concepts. So, let's start there.

1 PYTHON INTERPRETER INTERACTIVE MODE (REPL)

To run the Python REPL, launch a terminal and type either `python` or `python3` depending on whether you're running Windows (`python`) or macOS & Linux (`python3`). This launches the Python interpreter in *Interactive Mode*. **Note:** Windows users can run the Python interpreter in either a Command Prompt, Windows PowerShell, or Git Bash terminal. So, pick a terminal and launch the interpreter. Your terminal output should look similar to figure 3-1.

Referring to figure 3-1 — I'm running Python 3.10.8. Your version may be different and that's OK. The REPL prompt `'>>>'` indicates it's ready for commands. Let's start with some simple arithmetic.

1.1 THE ADDITION '+' OPERATOR

Refer to figure 3-2 to see the results of entering the following commands. Enter 2 + 2 and hit return. Notice how the REPL evaluated the *expression* and output the value 4 on the next line. The plus sign is referred to as the *addition operator*. The addition operator takes two *operands*, adds them together, and returns a result. In the case of 2 + 2 the addition operator is adding two

Figure 3-1: Running Python REPL (python3) on macOS in iTerm2

Figure 3-2: Applying the Addition Operator to Numbers and Strings

numeric types. Now enter `2.5 + 2.5`. The result is what you'd expect. Now, enter `'2.5' + '2.5'` and note the result. In this case you applied the addition operator to two *string* operands, which results in the *concatenation* of the two strings together. Concatenation is a ten-dollar word that means to join together. The result of the concatenation operation is a new string consisting of the concatenated values.

Python strings can be enclosed in single quotes `'string'` or double quotes `"string"`. Repeat the previous expression with double quotes like so: `"2.5" + "2.5"`. The result is returned in single quotes.
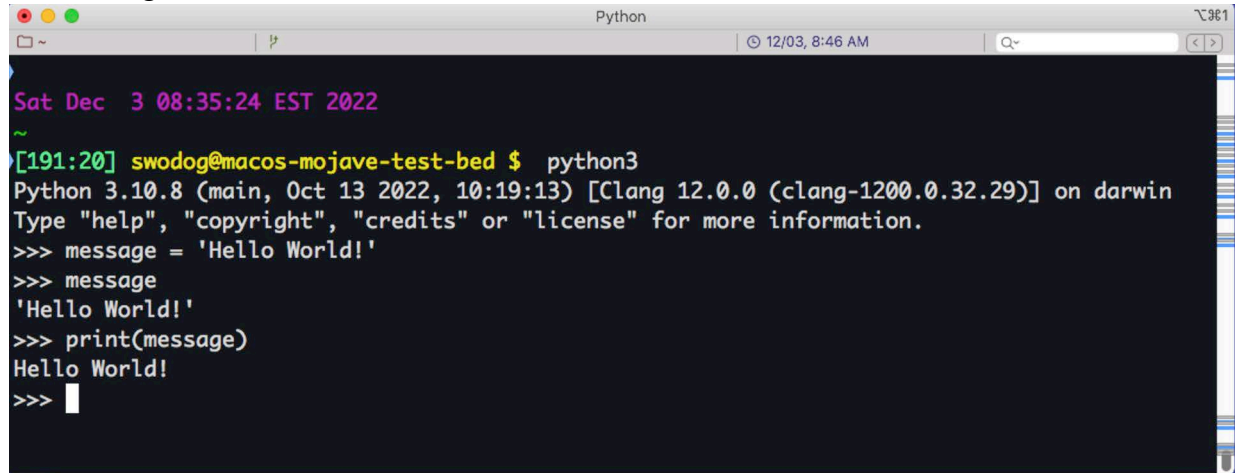
Single-quoted strings are the same as double-quoted strings. When you define a string within a Python program the string is referred to as a *string literal*. The same concept applies to numeric values that appear within your program. The expression `2.5 + 2.5` is using *numeric literals*. The expression `'2.5' + '2.5'` is using string literals. When using string literals in your programs it's good practice to pick either single-quoted strings or double-quoted strings and be consistent.

Pro Tip: Be consistent in your use of either single-quoted or double-quoted string literals.

I will use single-quoted strings for string literals going forward. Adopting a consistent string literal rule will make your programs easier to read, understand, and fix when something goes wrong. Something always goes wrong when you're just starting out!

1.2 VARIABLES

Enter the following command into the REPL: `message = 'Hello World!'` You've just defined a variable named `message`. A *variable* is a *named location in memory* that contains a value. In other words, the name `message` points to somewhere in memory that holds the string value `'Hello World!'`. To be even more precise, `message` points to the start of the string value or its first character. (*I dive deeper into Python code execution in Chapter 5: Computers, Programs, and Algorithms.*)You can now use the `message` variable anywhere you can use a string literal as shown in figure 3-3.
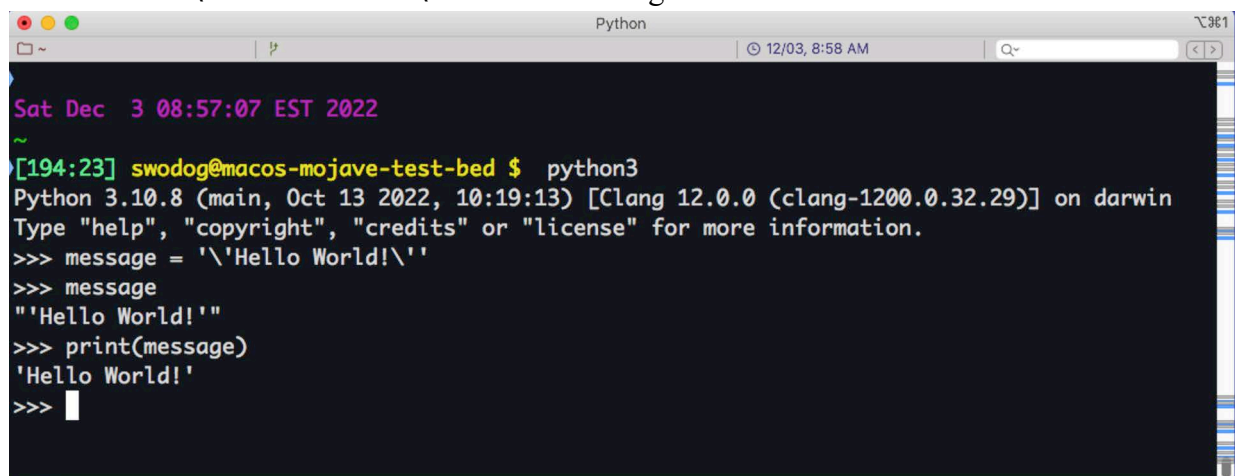

Figure 3-3: Defining and Using a Variable in the REPL

Referring to figure 3-3 — Note first that the variable `message` is defined. On the next line `message` is entered at the REPL prompt, which results in the output `'Hello World!'`. On the last line, I use Python's built-in `print()` function to print the value of the `message` variable. Note the difference between the two outputs. The first output `'Hello World!'` indicates that `message` is a string object. Printing `message` with the `print()` function just prints the characters without the quotes. If you wanted the single quotes to be part of the string you would have to define the string literal like so `'\'Hello World!\''` as shown in figure 3-4.
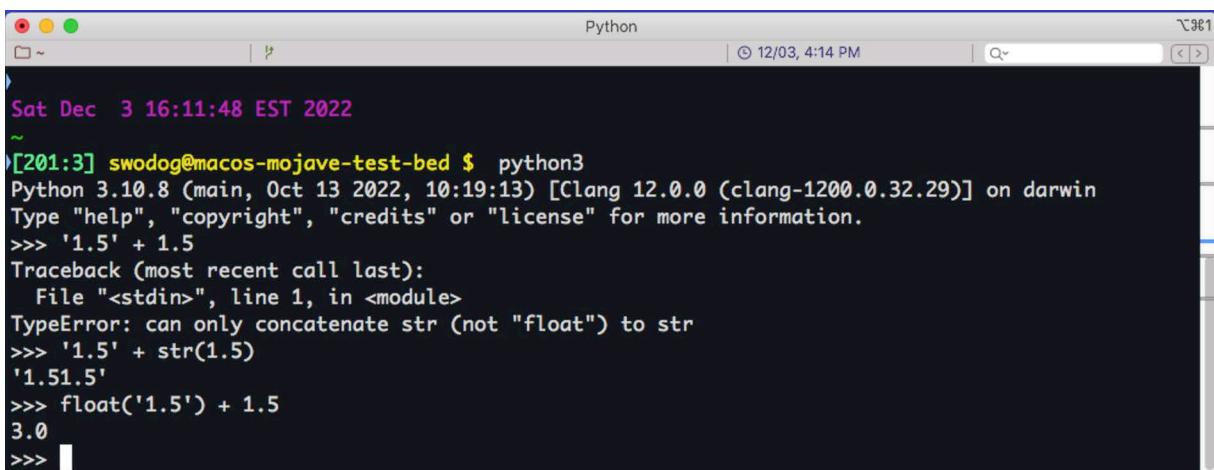

Figure 3-4: Using Embedded Quotes in a String Literal - Use the Escape Character '\'

Referring to figure 3-4 — The string literal assigned to the variable `message` contains *embedded single quotes*. The backslash character `'\'` represents the *escape character*. Together, the backslash character followed by the single-quote character `'\''` represents an *escape sequence*. The escaped single-quote characters are now part of the string literal `"'Hello World!'"` and is printed that way when the `message` variable is entered into the REPL. The `print()` function prints the string `'Hello World!'`.

1.3 Type Conversion

The `'+'` operator can add two numbers or concatenate two strings. What about mixing things up? Nope, that won't work. You'll need to pick a side, so to speak. By that I mean you'll need to decide what operation you want to perform, addition or concatenation, and either convert the strings to numbers or the numbers to strings as shown in figure 3-5.

Figure 3-5: Converting Types Before Applying `'+'` Operator

Referring to figure 3-5 — Trying to apply the `'+'` operator to mixed types results in an error with a helpful hint. Since there's a string in the mix the Python interpreter assumes you're trying to concatenate it with the number and suggests you convert the number to a string. To do so use the built-in `str()` function as shown on the next line. If instead you intended to perform addition, you'll need to convert the string to a number, and since the string is `'1.5'`, I've used the built-in `float()` function. Note that the string you're trying to convert to a number should parse to either a valid integer or float or you'll get an error.

Type conversion operations like the ones shown here happen frequently in Python code. Numbers entered via the console are read as strings and must be converted to their proper numeric type before use. Numbers must be converted to strings before being used in string concatenation operations. You'll see plenty more type conversion examples as you progress through the book.

1.4 Exiting The Python Interpreter

To exit the interactive Python interpreter do one of the following:
- On macOS and Linux type *Control-D, exit(), or quit()*
- On Windows type *Control-Z + Return, exit() or quit()*

1.5 Final Thoughts On The REPL

Using the Python interpreter in interactive mode is nice when you want to quickly run small snippets of code, but becomes cumbersome when you want to run larger and more sophisticated programs. So, I'll not spend any more time on it here. We have bigger fish to fry. I'll revisit the REPL in *Chapter 5: Computers, Programs, and Algorithms*, when I dive deeper into how Python loads and runs code.

Quick Review

The term Read, Evaluate, Print, Loop (REPL) refers to the Python interpreter running in *interactive mode*. The interpreter *read*s an expression from the >>> prompt, *evaluate*s the expression, and *print*s the results. The interpreter then returns to the >>> prompt and awaits further instructions, completing the *loop*.

When strings and numbers appear in programs they are referred to as *literals*. Choose single or double quotes to represent string literals in your programs and be consistent.

The '+' operator takes two arguments. If the arguments are strings it performs concatenation. If the arguments are numbers it performs addition. When applying the addition operator to mixed types you must decide which operation to perform and either convert the strings to numbers or convert the numbers to strings.

2 Run Python Programs From The Command Line

The Python REPL is helpful for experimenting with small snippets of code during your early forays into the language, but it quickly becomes tedious and ill suited for larger, more complex programs. Instead, put Python code into files called *modules* and run the files with the Python interpreter. Let's start with the universally accepted first program all budding programmers write — Hello World! Example 3.1 gives the code listing.

3.1 hello_world.py

```
1    print('Hello World!')
```

Referring to example 3.1 — The *hello_world.py* file contains one line of code that simply prints the string 'Hello World!' to the console. The .py file suffix indicates it's a Python source code file. OK, what do you do with it? First, using your preferred text editor, create the file named *hello_world.py* and enter the line of code shown above. Figure 3-6 shows how the editing session looks in Sublime Text.
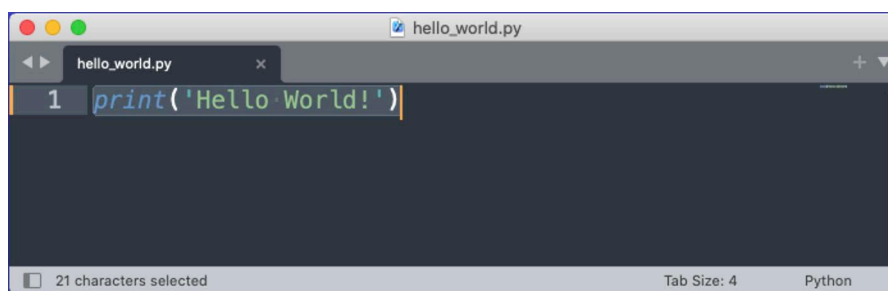

Figure 3-6: Creating hello_world.py with Sublime Text

Referring to figure 3-6 — Note that you DO NOT enter the line number shown in the example code and above. Sublime Text will automatically show line numbers. I add line numbers in example code to make it easy to reference a line when discussing the code. So, just enter the code `print('Hello World!')`. Now, before you save the file, think about where you want your Python source files to reside. In chapter 1, I recommended you create a folder in your home directory called *dev*. I recommend you create a subdirectory in the *dev* directory named *helloworld*, all lowercase. So, the path to that directory will be *~/dev/helloworld*. Save the *hello_world.py* file there. Now, open a terminal and navigate to the *~/dev/helloworld* directory.

To run the program, Windows users type `python hello_world.py` and UNIX/Linux users type: `python3 hello_world.py` The results of running example 3.1 on Windows and macOS are shown in figure 3-7.

Figure 3-7: Running hello_world.py on macOS (iTerm2 top) and Windows (Git Bash bottom)

2.1 Fixing Common Errors

If you managed to create and run the *hello_world.py* file without any problems, you're lucky. If you made a mistake or two then you're in good company. Common mistakes you might make, even with the simple *hello_world.py* example, include:

- Forgetting to surround the string `Hello World!` with single or double quotes
- Starting a string with a quote but omitting the ending quote
- Starting a string with one type of quote mark and ending with a different type
- Misspelling the `print()` method name
- Capitalizing the first letter of the `print()` method name
- Using a file suffix other than `.py` (*File will still run, but it's bad practice because development tools will use file suffixes to determine programming language.*)

If you do get an error when you try to run a program **fix the first error first**, as I advised in chapter 2. OK, let's take a look at a longer program whose source code is listed in example 3.2.

3.2 basic_io.py

```
1    message = 'Hello World!'
2    print(message.lower())
```

```
3    print(message.capitalize())
4    print(message.upper())
5    print(message.center(40, '-'))
6    print('*' * 60)
7    name = input('What\'s your name? ')
8    message = message.removesuffix('World!')
9    print(message + ' ' + name + '!')
```

Referring to example 3.2 — This program begins by creating a variable named `message` and assigning it the string `Hello World!`. On lines 2 through 5, I call various methods on the *message* string as it is passed to the `print()` method. For example, on line 2 I call the `lower()` method, which changes the string to lowercase. On the next line, I call the `capitalize()` method, which capitalizes the first character of the string. On line 4, I call the `upper()` method, which changes the `message` string to uppercase. On line 5, I call the `center()` method. This centers the string within the alloted space specified by the numeric argument, which in this case is `40`, and pads empty space with the `'-'` character. Line 6 may look strange. It illustrates a shortcut to print repeated characters to the console. In this case, I'm printing the asterisk character sixty times across the screen. On line 7, I use the built-in `input()` function to ask the user for input, which I assign to the `name` variable. I then make a call to the string `removesuffix()` method passing in the string `'World!'` to remove from the `message` string. This results in a new string with the value `'Hello '` being assigned to the `message` variable. Finally, on line 9, I concatenate the string variables `message` and `name`, and `print()` the string to the console.

Enter the code from example 3.2 into your text editor and save the file. Like the *hello_world.py* program shown earlier, create a subdirectory in your *~/dev* folder. I will create a subdirectory named *basicio*, so the full path to the file will be *~/dev/basicio/basic_io.py*. Open a terminal, navigate to the *~/dev/basicio* directory, and run the program. Figure 3-8 shows the results of running example 3.2.
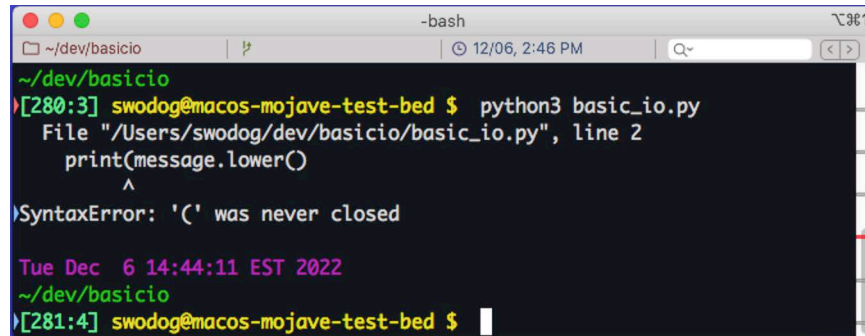
Figure 3-8: Results of Running basic_io.py

Referring to figure 3-8 — You can trace the effects of the code on the program's output. The `input()` function pauses execution while it waits until a user enters a string at the console. The important thing to note about the `input()` function is that it *returns a string*, which may need to be parsed and converted before use, depending on your needs. Also, I'm not doing any error checking on the input, so there's no telling what a user may actually enter.

Now, let's inject some faults into the code to see what errors look like. We'll do this in measured doses. Start by deleting the closing parenthesis from the end of the first `print()` statement. Save the file and run the program. The results should look similar to figure 3-9.

Figure 3-9: Syntax Error for Missing Closing Parenthesis
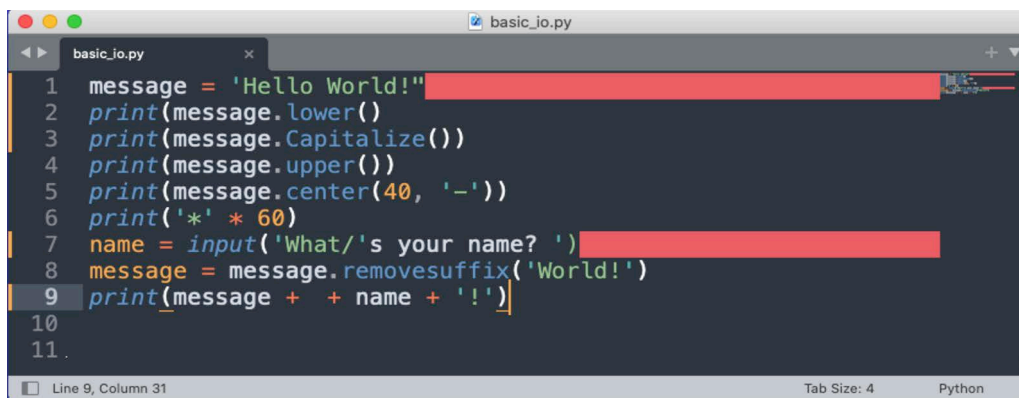
Referring to figure 3-9 — Python produces some fairly clear error messages. It gives you the full path to the file that produced the error and the line number on which the error occurred, and finally an error message. This one was easy, let's add several more. Figure 3-10 shows my Sublime Text editor with a handful of errors introduced to the *basic_io.py* file.
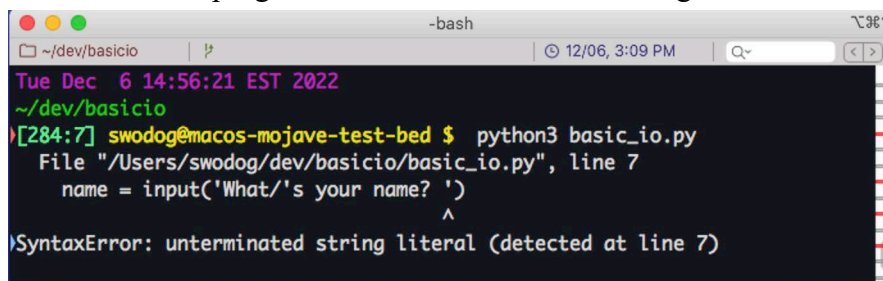
Figure 3-10: Multiple Errors Indicated in Sublime Text

Referring to figure 3-10 — This is another good argument for using a decent text editor or IDE. Sublime Text has highlighted a few of the lines containing errors. Some errors are easier to spot than others. For example, it's easy to see the mismatched quote types on line 1, but the capitalized letter 'C' on line 3 will not produce an error until the program actually runs. Well, to be more specific, a Python program *loads before it runs*. Syntax errors are checked during loading. Other types of errors called exceptions aren't detected until runtime. So, I'm going to fix the mismatched quote and reload the program. The results are shown in figure 3-11.

Figure 3-11: A Rather More Cryptic Error Message

Referring to figure 3-11 — OK, we see there's an error on line 7, but it says it's an unterminated string literal. This is something you may encounter fairly often if you use escape sequences in

strings and forget which slash character to use, a backslash `'\'` or a forward slash `'/'`. The correct escape character is the backslash.

Notice, however, that it skipped over the error on line 3. That's because Python is just trying to load, parse, and compile the program. It's not yet trying to run the program. (*I'll dive deeper into how Python loads and executes programs in Chapter 5: Computers, Programs, and Algorithms.*) The types of errors Python is catching during the program load and parse phase are *Syntax Errors*. You can see this in the error output of figure 3-11. If something goes wrong during program execution it will produce an *Exception*. We'll see this happen here in a second after we fix these syntax errors.

OK, change the forward slash back to a backslash, save the file and run the program once again. It should now catch the missing closing parenthesis on line 2. Go ahead and fix that error and repeat the save and run cycle. This is actually a good demonstration of what you'll be doing during development as discussed in chapter 2. You'll write some code, try to run it, fix any mistakes, and repeat the cycle until everything works fine.

Another common mistake you'll make is to forget to save the file when you make a change, and you'll wonder why none of your edits are taking effect when you run the program. That leads to some good advice: **Remember to save your changes whenever you edit a source file.**

Pro Tip: Remember to save your changes to a source file before running the program. If you're editing a source file and you don't see your changes take effect, you're either editing the wrong copy of the file or not saving your changes between edits.

Alright, that seems pretty funny for a Pro Tip, but believe me, even professional software engineers make these mistakes.

Continuing on, save your changes and run the program. You will now see a different kind of error as shown in figure 3-12.

Figure 3-12: AttributeError Exception Thrown on Line 3

Referring to figure 3-12 — Now Python is trying to run the program and when it attempts to call the Capitalize() method on a string object it throws an exception. An exception is a error that occurs during program runtime. Compare this error output with the one shown in figure 3-11 above. You notice here the message `hello world!` printed to the console, so the program was running up to the point it threw the exception. You can tell it's an exception because of the text `Traceback (most recent call last):` This is a fairly easy exception to decipher. Later in the book you'll see examples of exceptions that will really have you scratching your head. No worries. I'll show you how to read them.

OK, change the `Capitalize()` method back to `capitalize()` with a lowercase `'c'` and rerun the program. It should now run to the point where it asks you to input your name. Enter your name and hit return. Now it throws another exception as shown in figure 3-13.

Figure 3-13: TypeError Exception Thrown on Line 9

Referring to figure 3-13 — You'll make this mistake fairly frequently especially when you're just starting out writing your own code and it can be rather cryptic to figure out what's going on. Python is trying to apply the unary `'+'` operator to a string object. I haven't discussed the unary `'+'` operator yet but you apply it to a single numeric operand like so: +3. OK, fix that final error and run the program. See the last line of example 3.2 to see what's missing. Let's take a look now at how to run Python programs in Visual Studio Code

Quick Review

To run Python programs from the command line, first create a source code file that contains the Python code you want to execute and save it in a project directory. Next, launch a terminal, navigate to the project directory, and run the program using either the `python` or `python3` command depending on your operating system.

Don't be discouraged if your source code contains errors. Even professional programmers make coding mistakes. The first errors the Python interpreter will detect as it attempts to load a module are syntax errors. You'll need to fix all the syntax errors before a module will successfully load. The second type of errors the Python interpreter detects are runtime exceptions. These are errors that occur during program execution. A good text editor, like Sublime Text, provide a visual indicator when it detects Python syntax errors.

Like programming in general, the only way you get good at fixing errors is to write code, make mistakes, and figure out how to fix those mistakes.

3 Run Python Projects With Visual Studio Code

Launch Visual Studio Code. If you set it up as I suggested in chapter 1, then your start-up screen will look similar to figure 3-14.
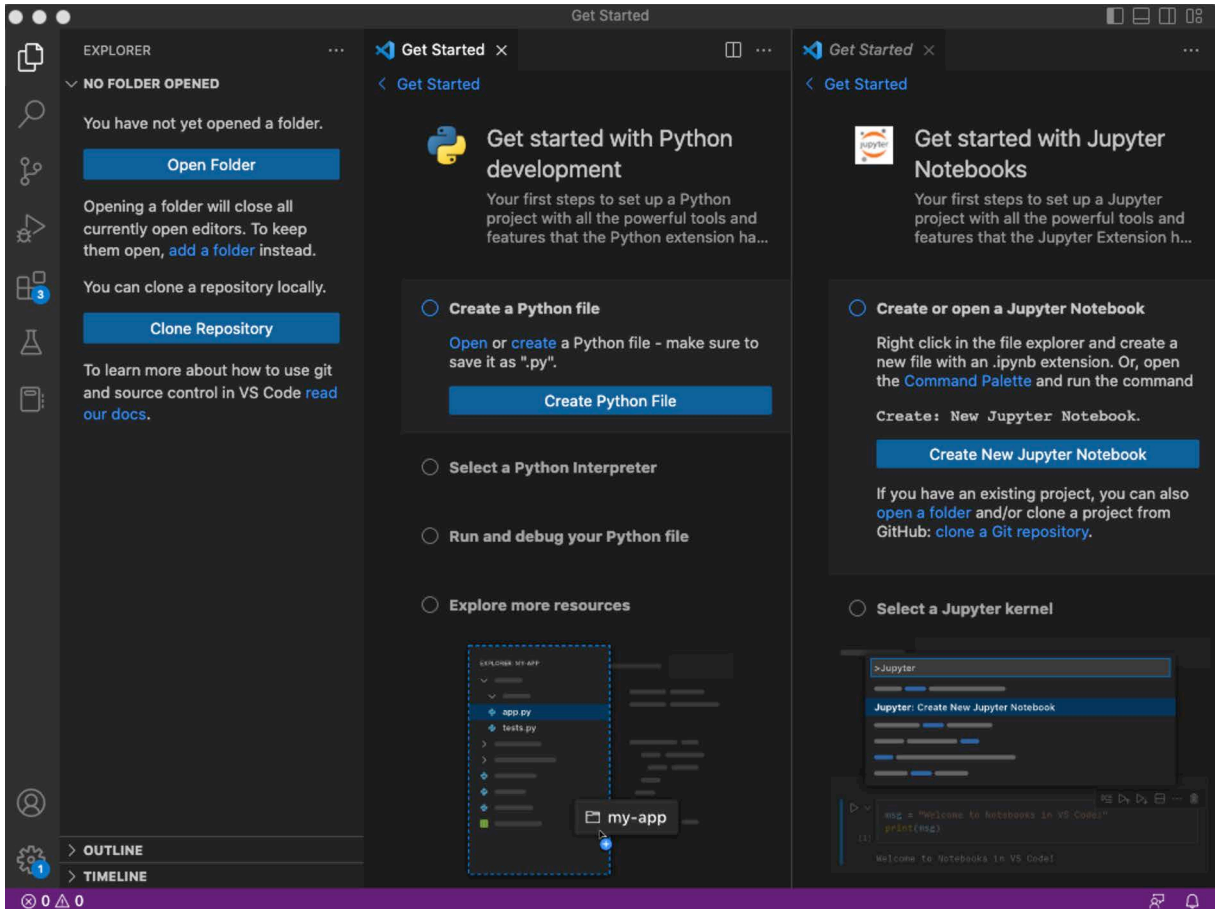
Figure 3-14: Launching Visual Studio Code with Recommended Python Plugins Installed

Referring to figure 3-14 — Starting from the left — the little blue circles on the icons along the left hand side indicate updates are available for extensions and for Visual Studio Code itself. In the EXPLORER column there are two blue buttons labeled **Open Folder** and **Clone Repository**. You'll learn more about *Git*, *GitHub*, and *repositories* in *Part II: Foundations, Chapters 7 & 8*, so you can ignore the Clone Repository button for now. Moving right, the first **Get Started** panel guides you on *Getting started with Python development*. The next **Get Started** panel guides you on *Getting started with Jupyter Notebooks*. You can ignore Jupyter Notebooks as I will not be using them in this book.

3.1 Update Visual Studio Code And Extensions

The first thing I always do when I launch Code is to install all pending updates to Code itself and to installed extensions. To update Code, click on the gear cog icon in the lower left corner of the screen and restart to install the update. You may see a new panel with Release Notes related to the update you just installed. I recommend reviewing them to see what's new as shown in figure 3-15.

Referring to figure 3-15 — Notice the update indicator on the extensions icon is still lit up. Reload Visual Studio Code once again to install the updated extensions. When you've completed the updates you'll be ready to proceed. This is something you'll go through fairly regularly with Visual Studio Code. It's good practice to keep everything up-to-date.
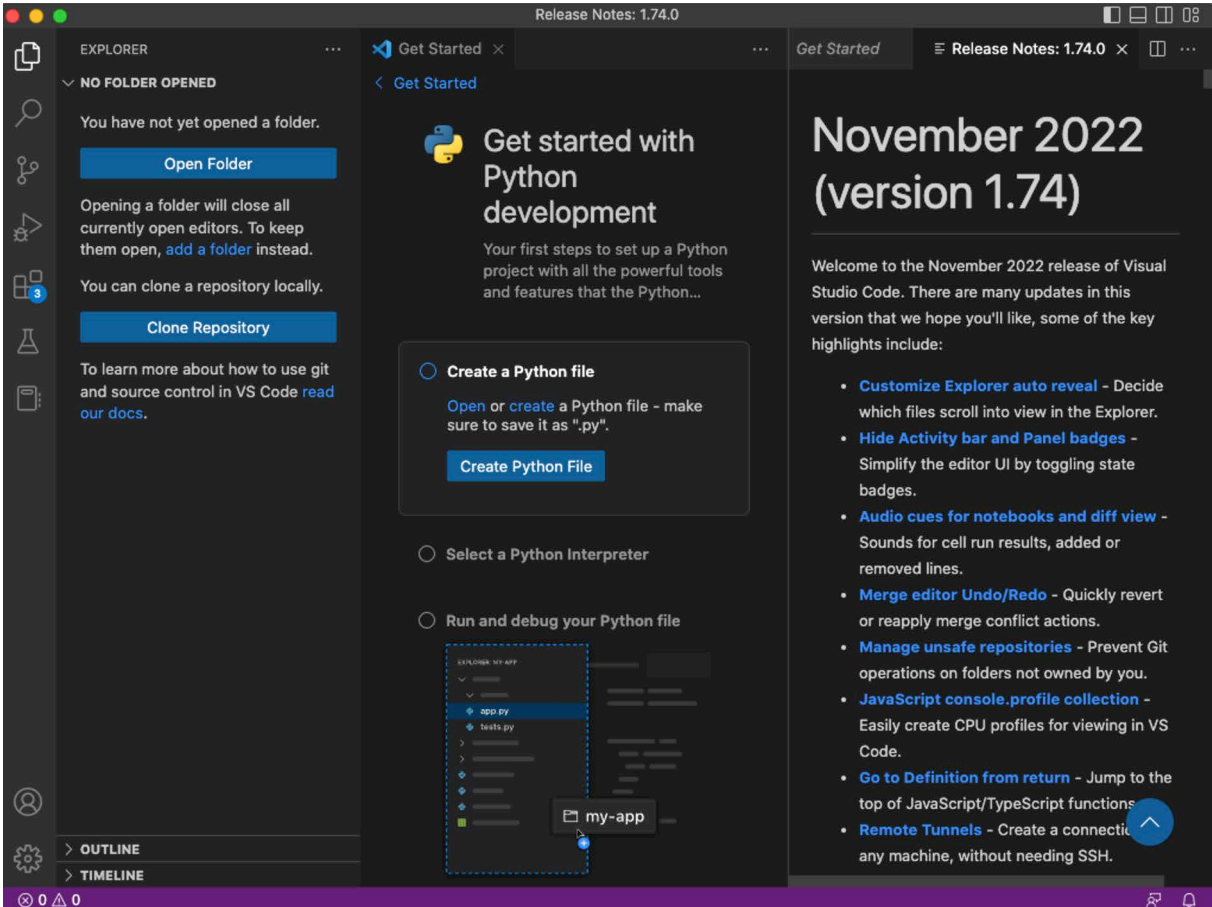
Figure 3-15: Release Notes Appear After Update

3.2 Think Project Folder

You really need to select a directory in which to store your programming projects. You need to *start organized*, *get organized*, and *stay organized*. Yes, I'm in lecture mode. Not knowing where you saved your project files will mess with your head. By now you know I recommend creating a *~/dev* folder in your home directory and creating subdirectories under it to store individual projects. Of course you can exceed my recommendations. As the number of programming projects you work on increases, the need for structure and organization will become painfully obvious. It's easier to stay organized if you start organized than it is to get organized from an absolute disorganized disaster.

Pro Tip: Start organized and stay organized. Organize your source code into project folders.

If you followed along in the previous section and created the ~/dev/basicio/basic_io.py file, then open that project folder with Visual Studio Code. You should first see a warning message as shown figure 3-16.

Referring to figure 3-16 — Unless you have a compelling reason not to do so, click **Yes, I trust the authors**. Your Visual Studio Code window will look similar to figure 3-17.
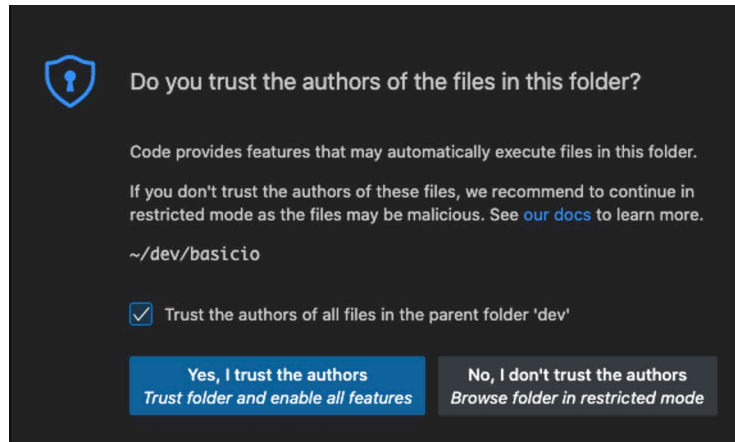
Figure 3-16: Trust Authors Warning when Opening Folder with Code
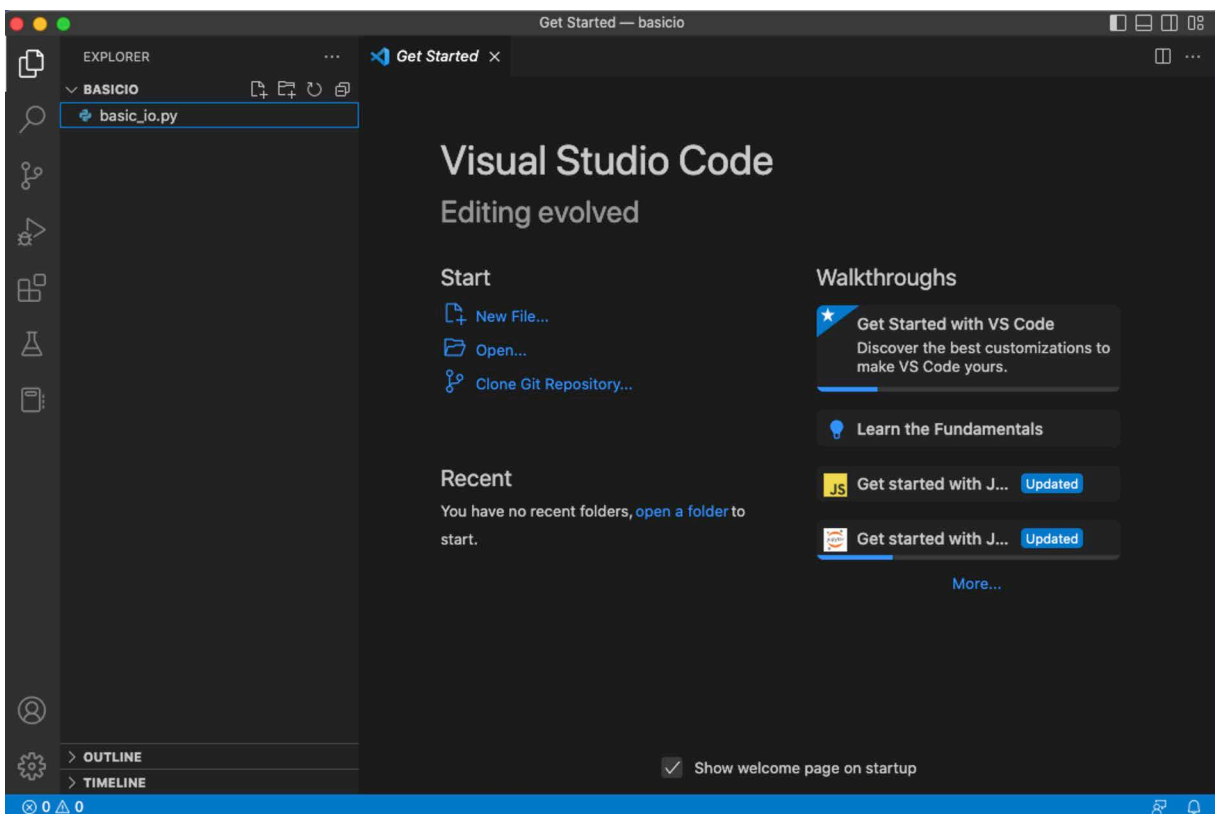
Figure 3-17: Visual Studio Code Listing Files in the basicio Project Folder

Referring to figure 3-17 — You can see in the EXPLORER panel the BASICIO project folder expanded to show the files it contains. In this case there's only one file *basic_io.py*. Double-click the file to open it. You should see something similar to figure 3-18.

Referring to figure 3-18 — I have increased the font size to make it easier on the eyes.
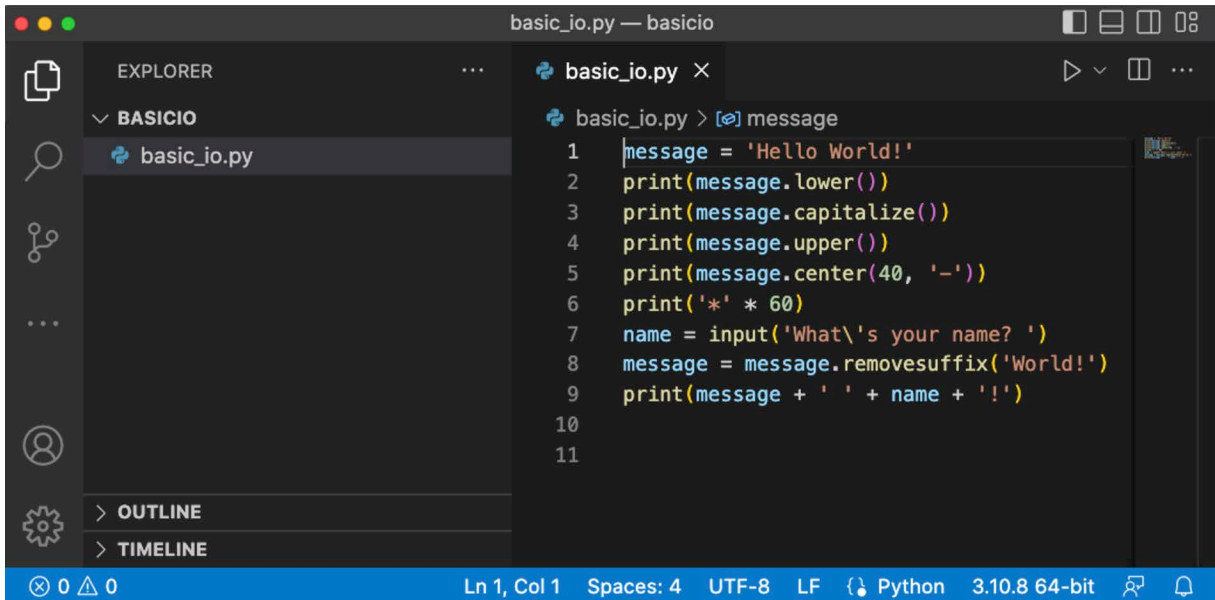
Figure 3-18: Editing the basic_io.py File

3.3 RUNNING A PROGRAM

You have several ways to run a Python program when using Code. You can run it directly in Visual Studio Code by selecting the file and clicking the triangle in the upper-right corner of the window. This will launch an embedded terminal within Code as shown in figure 3-19.
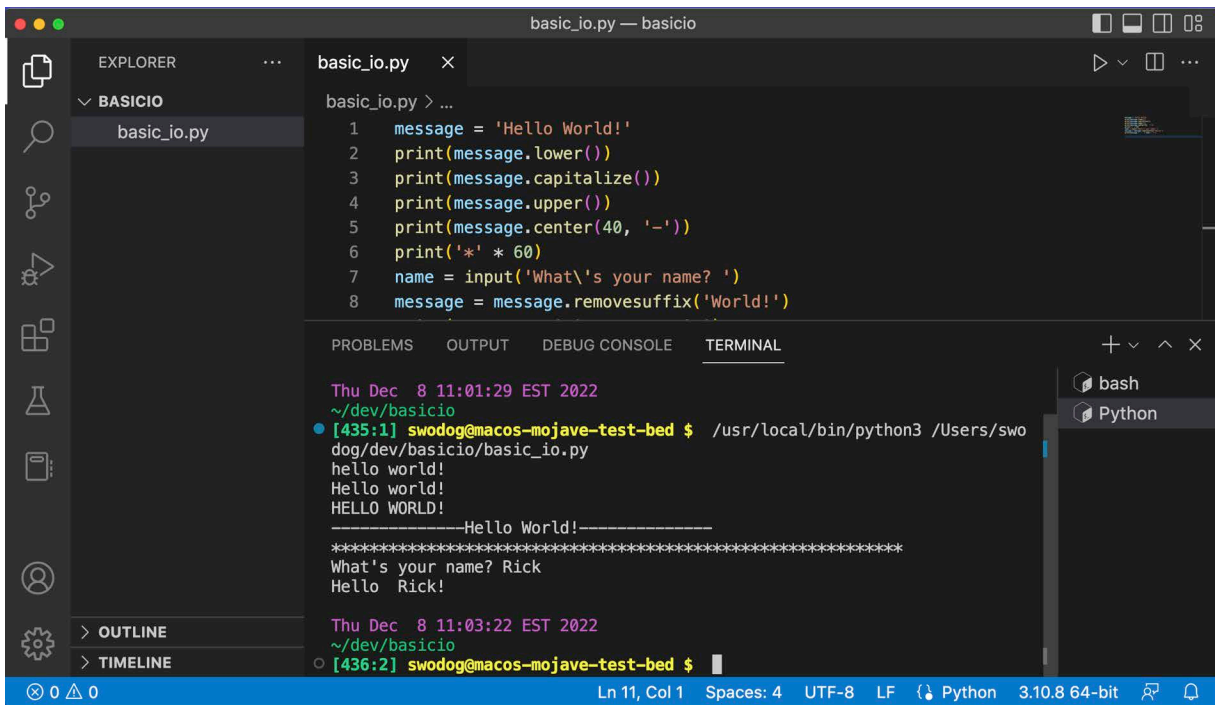
Figure 3-19: Running a Program in Code's Embedded Terminal

Referring to figure 3-19 — When you click the triangle, Code will run the selected file in an embedded terminal within the Visual Studio Code Workbench, which is Microsoft's name for the arrangement of Visual Studio Code's user interface panels. Refer to the VS Code documentation for a more detailed treatment: *https://code.visualstudio.com/docs/getstarted/userinterface*

Note: I've never experienced a problem launching and using Visual Studio Code's embedded terminal when running it directly on macOS, Linux, or Windows. However, when I ran Code in a Parallels virtual machine, which I do for the macOS examples in this book, I did have a problem which I resolved by launching Code from the command line like so:

```
code --disable-gpu
```

You can also create an alias in your *.bash_profile* or *.bashrc* like so:

```
alias code='code --disable-gpu'
```

You can avoid these issues altogether by simply running your Python programs the second way, which is just running them in a separate terminal. This is what I do because while using Code's embedded terminal does come in handy sometimes, I prefer to devote as much workbench space to the editor as possible as shown in figure 3-20.
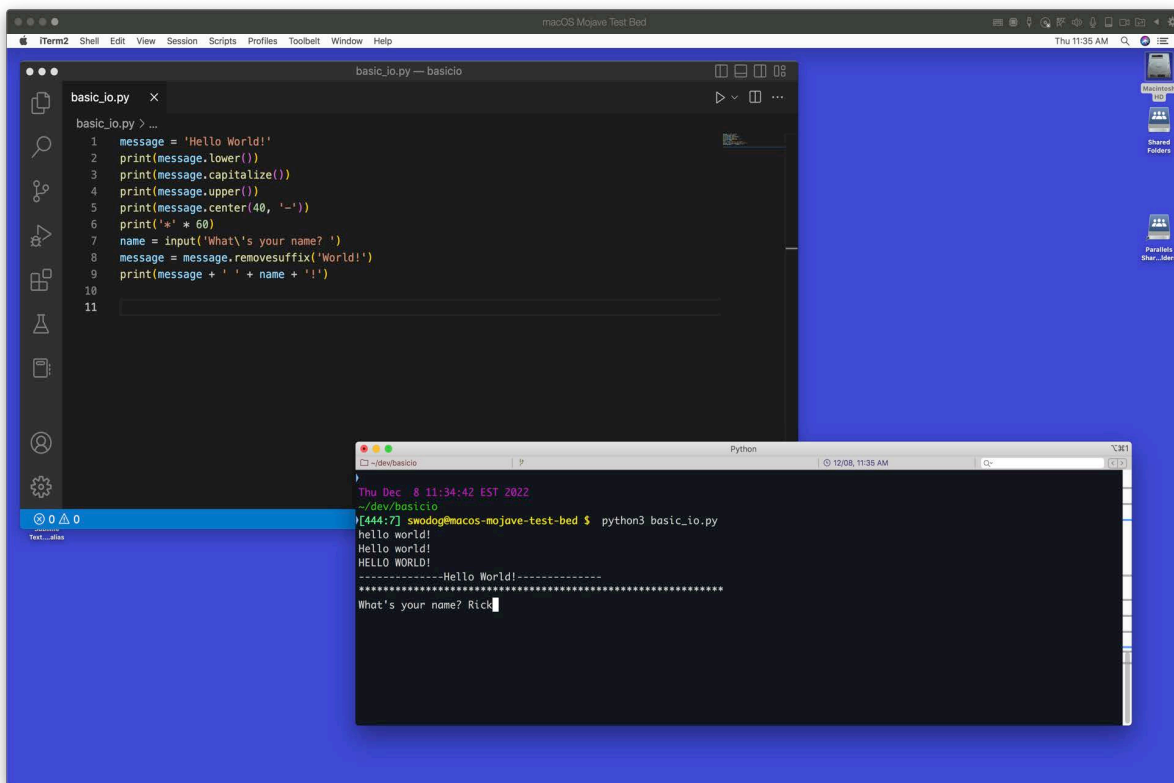


Figure 3-20: Coding in One Window and Running The Program in External Terminal

Referring to figure 3-20 — In reality, I have multiple monitors and I'll run my text editor or IDE on one screen and have the terminal running on another screen. When you're programming, you will also usually have a browser open to reference documentation or Google results, so you'll have multiple windows open simultaneously. I find one monitor too restricting, two is better, three or more is ideal. How you code and run your programs is largely a matter of personal taste.

3.4 Launch Git-Bash in Code's Embedded Terminal

This is for Microsoft Windows users. When you run Visual Studio Code in Windows, the embedded terminal is set to launch the command prompt (cmd.exe). I recommend you set the default profile to run Git-Bash. Here's how to do it. Launch Visual Studio Code and enter **Ctrl-Shift-P** and in the search box start typing **Select Default Profile**. You'll see **Terminal: Select Default Profile** come up. Click it to display a list of terminal profiles as shown in figure 3-21.
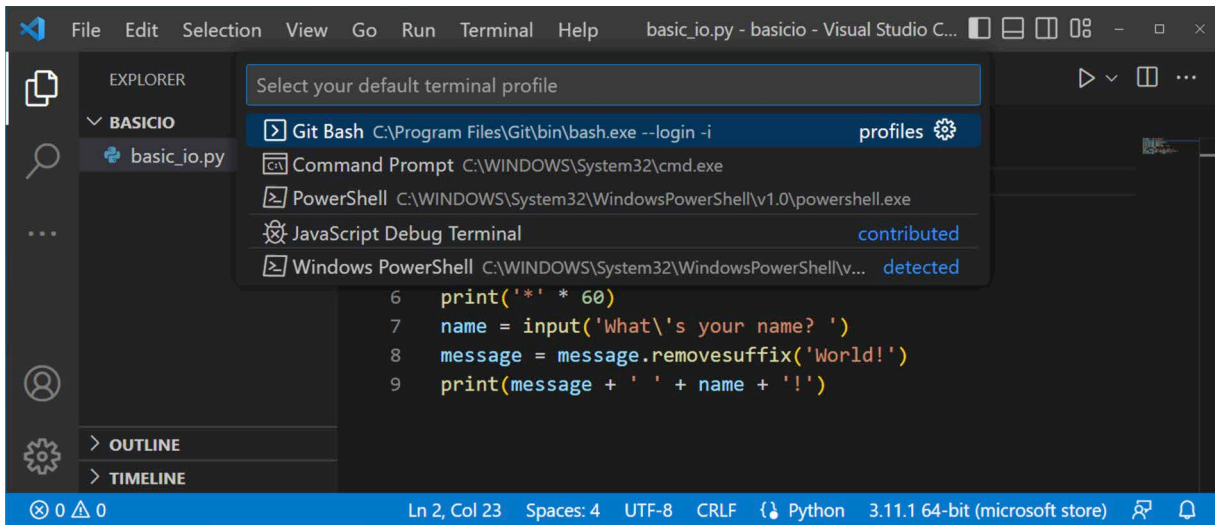
Figure 3-21: Select Git Bash as the Default Terminal Profile

Referring to figure 3-21 — Select the Git Bash terminal profile, then from the **Terminal** menu select **New Terminal** and ensure it launches Git Bash. If not, you may need to restart Code. Figure 3-22 shows Git Bash running in VS Code's embedded terminal.
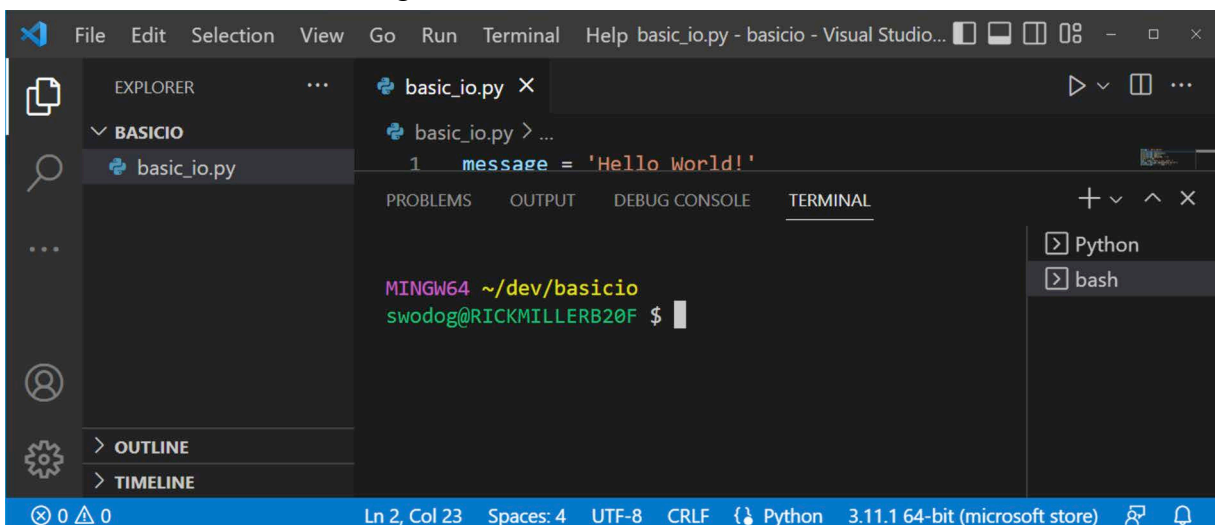
Figure 3-22: Git Bash Running in Visual Studio Code's Embedded Terminal

Referring to figure 3-22 — Click the triangle to run the program. OK, I still recommend running programs in a separate terminal. Visual Studio Code has plenty of nice features even without the embedded terminal.

3.5 Debugging Programs In Visual Studio Code

Still in the *~/dev/basicio* folder with *basic_io.py* open in an editor, click the little v-shaped icon to the right of the run triangle and select **Debug Python File**. Notice your Visual Studio Code workbench changes as shown in figure 3-23.
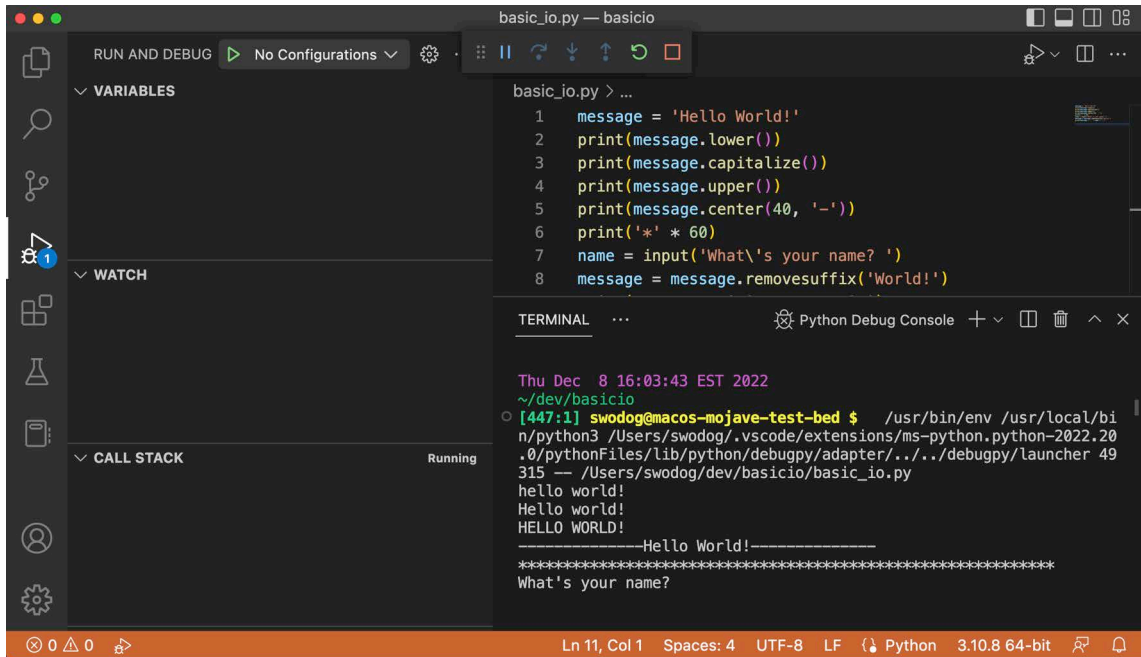
Figure 3-23: Visual Studio Code in Debug Mode

Referring to figure 3-23 — A debugger enables you to step through your code to help pinpoint problems. To stop somewhere in your code you need to set a *breakpoint*. To do this, click to the left of a line number in the editor panel as shown in figure 3-24.
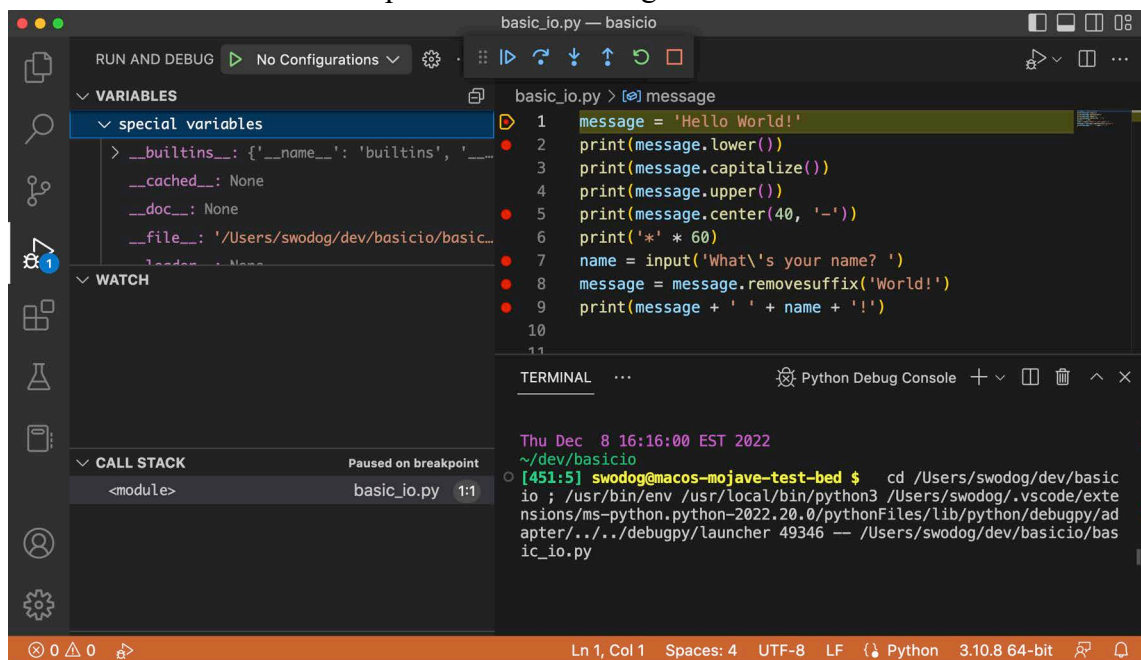
Figure 3-24: Breakpoints Set and Debug Session Active

Referring to figure 3-24 — I set several breakpoints as indicated by the red dots next to the line numbers. I then restarted the debug session to start at the top. Notice the **special variables** section of the RUN AND DEBUG panel is now populated. (*Don't worry if you don't understand what you see in the special variables section right now. You will by the time you finish this book.*) To interact with the debugger use the control strip located in the upper center of the workbench. You can *Continue*, *Step Over*, *Step Into*, *Step Out*, *Restart*, and *Stop* the session. Take time now to familiarize yourself with the debugger. For more information I recommend you consult the Microsoft Visual Studio Code documentation. *https://code.visualstudio.com/docs/editor/debugging*

QUICK REVIEW

Organize projects into folders and open the folders with Visual Studio Code. The code examples in this chapter are extremely simple but soon I'll introduce you to a formal project structure to apply to all of your programming projects.

You can run programs in Visual Studio Code's embedded terminal, which is convenient, but I recommend running programs in a separate terminal. If you're using Visual Studio Code in Microsoft Windows, set the default terminal profile to Git Bash.

If you have trouble with the embedded terminal on macOS (or any platform) try launching Visual Studio Code from the command line with the GPU disabled like so:

```
code --disable-gpu
```

Use Visual Studio Code's debug mode when you need to troubleshoot programs. Set breakpoints to signal to the debugger where to stop during code execution to allow for deeper fault analysis.

4 TYPE CONVERSION AND ERROR HANDLING

You saw in example 3.2 how the `input()` function is used to get console input from the user. The `input()` function takes a *string argument*, which is used to provide a *prompt* to the user, and *returns a string* containing what the user typed on the command line. You may need to convert the string into a different type, say, a numeric value, before using it in your program. Example 3.3 lists the code for a program that attempts to calculate the sum of two numbers entered via the console.

3.3 calc_sums.py

```python
1    """Demonstrates String to Number Conversions"""
2
3    def main():
4        # Read number strings from console
5        num1_string = input('Enter first number: ')
6        num2_string = input('Enter second number: ')
7        try:
8            # Convert number strings to integers
9            num1 = int(num1_string)
10           num2 = int(num2_string)
11           # Add the numbers
12           sum = num1 + num2
13           # Print results using formatted string
14           print(f'The sum of {num1} and {num2} = {sum}')
15       except Exception as e:
```

```
16          print(f'Problem converting strings to integers: {e}')
17
18
19   if __name__ == '__main__':
20       main()
```

Referring to example 3.3 — First, I want to explain this example's overall organization. At the top there's a documentation string comment or, just simply, a *docstring*. A docstring begins and ends with three double quotes. Docstrings are used to help automatically generate meaningful documentation for your projects. You'll learn how to effectively employ docstrings and generate project documentation later in the book.

Next, on line 3, a method named `main()` is defined with the `def` keyword. The *body* of the `main()` method begins on line 4, which is indented four spaces from the beginning of the method definition and ends on line 17. Line 4 contains a comment, which starts with the pound or hashtag `'#'` character. Everything past the hashtag character on that line is ignored by the interpreter. The comment here is important because it's describing the intention of the code on line 5 and 6, which is to read string input from the console. What could possibly go wrong?
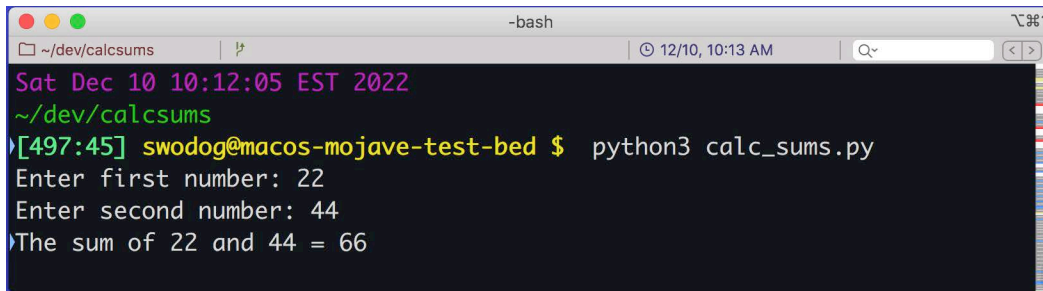
The rest of the `main()` function's code executes in the body of a *try statement* which begins on line 7. Anytime you do something in code that *could possibly throw an exception* you want to place that code in a *try clause*, which in this case includes all the code from lines 8 through 14. If the string-to-integer conversions on line 9 and 10 go wrong, the Python interpreter throws an exception and the remainder of the code within the try clause following the line that threw the exception is skipped and will not execute. It is the job of the *except clause* to catch the exception and do something meaningful with it so your code *behaves predictably*. In this example, the except clause will handle every type of exception that is thrown because it's looking for an *Exception* object, which is the base type of all exceptions. In this example, if an exception is thrown, the except clause catches it and prints a polite message to the console along with the original exception.

Pro Tip: Place code that may throw an exception in the body of a try clause and handle the exception with a except clause.

If all goes well (*The pathetic lament of a software engineer*), the strings entered by the user will convert to integers with the built-in `int()` function and the `sum` of the two values will be calculated and printed to the console. Note that this example uses *formatted strings*, also referred to as `'f'` strings. The string used as an argument to the `print()` function begins with an `'f'` and variables are placed in curly braces `"{ }"` within the string. If you used string concatenation instead of formatted strings, you would have to convert the numeric variables to strings using the `str()` function as shown earlier in this chapter.

Finally, on line 19, an `if` statement checks the `__name__` property of the *calc_sums.py* module and compares it to the string `'__main__'` using the equality operator `'=='`. If the `calc_sums.py` file is executed directly by the python interpreter then its module __name__ property will be set to `'__main__'` and the `main()` method will execute. You will frequently encounter this *pattern*, or *idiomatic usage*, represented by lines 19 and 20. I will use it throughout this book to create a *main entry point* for complex Python applications.

OK, to run this program, create the source file with your favorite text editor or IDE, and run it with either `python` or `python3`, depending on your operating system. Figure 3-25 shows the results of running this program.

Figure 3-25: Results of Running Example 3.3 with Inputs '22' and '44'

Referring to figure 3-25 — This shows the results of running calc_sums.py with inputs `'22'` and `'44'`. Try with different signed values like `'-33'` and `'+2433'`. Next, use decimal values and see the results of throwing an exception. I'll leave it as an exercise for you to make the program work with integers *and* floats.

4.1 How A Module Executes

Referring again to example 3.3 — As you saw earlier, when you run a module with the Python interpreter, it loads the module and performs a line-by-line syntax check. Upon passing, the interpreter then evaluates and executes each line of the module. Code that starts along the left-most side, in other words, code that is not indented, is evaluated, processed, and/or executed when the module is loaded. So, beginning at the top of the file, the docstring is evaluated and stored along with other docstrings found in the module. Next, the function definition of `main()` is evaluated and stored along with other function definitions that may be in the module. Finally, the `if` statement executes. If the module was executed directly by the Python interpreter then the `main()` function is called and runs as you see in figure 3-25. If instead the module was imported into another module, you'd see nothing happen until it explicitly called the `main()` function.

Note the difference between a function's *definition* and its *execution*. The `main()` function's definition starts with the `def` keyword on line 3 of example 3.3. The `main()` function does not execute until it is *called* in the body of the `if` statement.

Quick Review

Use the Python Standard Library built-in `input()` function to read a line of string input from the console. It takes a string argument that sets the prompt and returns a string value that may need to be converted into a different type depending on the needs of your program.

Any code that may throw an exception should be placed in the body of a `try` clause. This includes type conversion operations such as converting string values read from the console and converting them into numeric values. There's always the possibility a user will enter a string that does not represent a valid numeric value and will fail the conversion. Use an `except` clause to gracefully handle exceptions.

The `__name__` property of a module directly executed by the Python interpreter is set to the string value `'__main__'`. The idiomatic expression:

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

...checks the value of the module's `__name__` property and if it equals `'__main__'` it calls the `main()` method.

5 IMPORTING MODULES AND INSTALLING PACKAGES

So far in this chapter the examples have relied upon built-in functions to get things done. To write more serious programs, you'll need more help. The Python Standard Library comes with an assortment of modules you can import into your programs when you need extra muscle. However, these, too, will only get you so far. Many times you'll need to install one or more *third-party packages* to gain the necessary functionality. This holds especially true for performing certain types of date calculations. Example 3.4 gives a program that calculates a user's age given their birthday.

3.4 calculate_age.py

```
1    """Calculates user's age based on their birthday."""
2
3    # Import the date and datetime classes from datetime module
4    from datetime import date, datetime
5    # Import the relativedelta function from the third-party
6    # dateutil.relativedelta package
7    from dateutil.relativedelta import relativedelta
8
9    def main():
10       # Input birthday string in the format mm/dd/yyy
11       date_string = input('Enter Date of Birth (mm/dd/yyyy): ')
12       try:
13          # Split the input string along '/' boundaries
14          # This results in a list of strings
15          parsed_date_list = date_string.split('/')
16          # Convert month string to int
17          month = int(parsed_date_list[0])
18          # convert day string to int
19          day = int(parsed_date_list[1])
20          #convert year string to int
21          year = int(parsed_date_list[2])
22          # Create the birthdate date object
23          birthdate = date(year=year, month=month, day=day)
24          # Print birthday to console
25          print(f'Your birthday is {birthdate}')
26          age = relativedelta(datetime.now(), birthdate)
27          print(f'You are {age.years} years old.')
28          print(f'Your absolute age is: {age}')
29
30       except Exception as e:
31          print(f'Problem calculating date: {e}')
32
33    if __name__ == '__main__':
34       main()
```

Referring to example 3.4 — On line 4, I import the *date* and *datetime* classes from the *datetime* module. The *datetime* module is part of the Python Standard Library. While the Python date and datetime classes provide a lot of date manipulation features, it's rather clunky to calculate the

years between two dates, so this program requires the extra muscle provided by a third-party package named *python-dateutil*: *https://dateutil.readthedocs.io/en/stable/*

Before you can use the python-dateutil package in a program you'll need to install it using either `pip` or `pip3` from the command line like so:

```
pip3 install python-dateutil
```

You should now be able to run the program like so:

```
python3 calculate_age.py
```

On line 11, the program prompts the user to enter their date of birth in *mm/dd/yyyy* format. This is returned as a string to the `date_string` variable. From this point forward things could go horribly wrong, so the bulk of the code is located in the body of the `try` clause. The comments explain step-by-step what's happening in the code. The `date_string` is split into a *list* of strings by calling the `string.split()` method with a argument of `'/'` which tells the method to split the string along forward-slash boundaries. This results in a list of three strings. You'll learn more about lists in *Chapter 14: Lists and Tuples*, but in a nutshell, the `parsed_date_list` variable points to a list object that represents the result of the string.*split()* operation. To access the individual strings within the list use integer indices inside square brackets. For example, `parsed_date_list[0]` points to the first string, or the string that represents the month number *mm*; `parsed_date_list[1]` points to the second string, or the string that represents the day number *dd*; and finally, parsed_date_list[2] points to the third string, or the string that represents the year number *yyyy*. Each of these strings must be converted into integers before they can be used to create a date object.

On line 25, the `birthdate` variable is printed to the console. On line 26, I use the `relativedelta()` method to calculate the difference between the current date `datetime.now()` and the `birthdate`. On line 27, I print `age.years` to the console. On line 28, I print the complete `age` (*relativedelta*) object to the console. Figure 3-26 shows the results of running this program.

Figure 3-26: Results of Running calculate_age.py

Referring to figure 3-26 — If you were ever curious just how old you actually are this is a good way to find out.

QUICK REVIEW

The Python Standard Library contains modules you can import into your programs when you need specialized functionality. When the standard library falls short, you can install third-party packages with `pip` or `pip3` and import them into your program.

6 PYTHON STANDARD LIBRARY AND PYTHON PACKAGE INDEX

I've used several elements from the *Python Standard Library* to implement the examples in this chapter. These include the *built-in functions* `input()`, `int()`, `print()`, and `str()`, the *built-in types* *integer* (Numeric), *str* (Text Sequence), *list* (Sequence), and the *built-in Exception* class. I used `string.split()` from the *Common string operations* module to split a string into a list of substrings. Finally, I imported the *datetime* module to perform date manipulations. This represents only a small fraction of the functionality supplied by the standard library.

From the *Python Package Index (PyPI)* I installed the *python-dateutil* package and used the `relativedelta()` function to compute the difference between two date objects.

You'll find it helpful to browse the Python Standard Library and get a feel for what's included and available with your Python installation.

Pro Tip: Familiarize yourself with the Python Standard Library and the Python Package Index (PyPI)

A common mistake beginners make when learning modern programming languages that come with extensive supporting libraries or frameworks is to try to reinvent the wheel without first searching to see if what they are trying to do is already done for them by some component in a supporting library or framework. So learning how to program in Python is part learning Python and part learning what's in the Python Standard Library and the Python Package Index (PyPI).

SUMMARY

The term **R**ead, **E**valuate, **P**rint, **L**oop (REPL) refers to the Python interpreter running in *interactive mode*. The interpreter *read*s an expression from the `>>>` prompt, *evaluate*s the expression, and *print*s the results. The interpreter then returns to the `>>>` prompt and awaits further instructions, completing the *loop*.

When strings and numbers appear in programs they are referred to as *literals*. Use single or double quotes to represent string literals in your programs and be consistent.

The `'+'` operator takes two arguments. If the arguments are strings it performs concatenation. If the arguments are numbers it performs addition. When applying the `'+'` operator to mixed types you must decide which operation to perform and either convert the strings to numbers or convert the numbers to strings.

To run Python programs from the command line, first create a source code file that contains the Python code you want to execute and save it in a project directory. Next, launch a terminal, navigate to the project directory, and run the program using either the `python` or `python3` command depending on your operating system.

Don't be discouraged if your source code contains errors. Even professional programmers make coding mistakes. The first errors the Python interpreter will detect as it attempts to load a module are syntax errors. You'll need to fix all the syntax errors before a module will successfully load. The second type of errors the Python interpreter detects are runtime exceptions. These are errors that occur during program execution. A good text editor, like Sublime Text, provide a visual indicator when it detects Python syntax errors.

Like programming in general, the only way you get good at fixing errors is to write code, make mistakes, and figure out how to fix those mistakes.

Organize projects into folders and open the folders with Visual Studio Code. The code examples in this chapter are extremely simple but soon I'll introduce you to a formal project structure to apply to all of your programming projects.

You can run programs in Visual Studio Code's embedded terminal, which is convenient, but I recommend running programs in a separate terminal. If you're using Visual Studio Code in Microsoft Windows, set the default terminal profile to Git Bash.

If you have trouble with the embedded terminal on macOS (or any platform) try launching Visual Studio Code from the command line with the GPU disabled like so:

```
code --disable-gpu
```

Use Visual Studio Code's debug mode when you need to troubleshoot programs. Set breakpoints to signal to the debugger where to stop during code execution to allow for deeper fault analysis.

Use the Python Standard Library built-in `input()` function to read a line of string input from the console. It takes a string argument that sets the prompt and returns a string value that may need to be converted into a different type depending on the needs of your program.

Any code that may throw an exception should be placed in a `try` clause. This includes type conversion operations such as converting string values read from the console and converting them into numeric values. There's always the possibility a user will enter a string that does not represent a valid numeric value and will fail the conversion. Use an `except` clause to gracefully handle exceptions.

The `__name__` property of a module directly executed by the Python interpreter is set to the string value `'__main__'`. The idiomatic expression:

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

...checks the value of the module's `__name__` property and if it equals `'__main__'` it calls the `main()` method.

The Python Standard Library contains modules you can import into your programs when you need specialized functionality. When the standard library falls short, you can install third-party packages with `pip` or `pip3` and import them into your program.

SKILL-BUILDING EXERCISES

1. **Study Python Standard Library:** Study the *Python Standard Library*. Make this a standing assignment and focus on a particular section each day. For example, start with the built-in functions and note their usage, followed by the built-in constants, etc. Note that the goal is not to memorize the contents of the standard library, but rather to familiarize yourself with the contents so that when you need to do something in your code, you'll remember there was something in the library that could help you.

2. **Peruse The Python Package Index (PyPI):** Familiarize yourself with the Python Package Index (PyPI) website *https://pypi.org*.

3. **Study The Package Installer For Python (pip or pip3):** Study the purpose and use of the *Package Installer for Python* or Pip for short. (*On Windows you run the* `pip` *command; on Unix/Linux you run* `pip3`.) Where are these packages installed on your system?

3. **Verify Operation of Visual Studio Code Embedded Terminal:** If you're using Visual Studio Code, launch the embedded terminal and verify its operation. If it seems glitchy, try starting Code from the command line with the command `code --disable-gpu` and see if that corrects the issue.

4. **Visual Studio Code Embedded Terminal on Windows:** Change the terminal profile from the Command-Prompt to Git Bash as discussed in section 4.4. The reason you'll want to do this is so you can run bash scripts in later chapters.

5. **Research The Python Interpreter Interactive Mode (REPL):** Although we did not dwell too long on how to use the Python interpreter in Interactive Mode, it is a quite powerful learning tool. Try running examples 3.2, 3.3, and 3.4 in the REPL. Here's how I recommend you proceed. Change to a project directory that contains the Python file you want to run in the REPL. Launch the Python interpreter in Interactive Mode and at the `>>>` prompt type `import` *module*, where *module* is the name of the Python file you want to run **minus its extension**. (Very important!) **Example:** You want to import and run the basic_io.py example. Navigate to the basicio directory, launch the Python interpreter, and at the REPL prompt `>>>` type: `import basic_io`. Note the difference between the behavior you see when importing the basic_io module vs. importing the *calc_sums* module. Can you explain the difference? What must you enter at the REPL prompt to run the calc_sums program?

6. **PEP 8 — Style Guide For Python Code:** Browse and study the PEP 8 Style Guide for Python Code: *https://peps.python.org/pep-0008/*. Focus on the sections related to code formatting, and function and variable naming.

7. **The Python Debugger:** Python has a debugger, `pdb`, which can be used to debug Python programs from the command line. Research the Python Debugger and use it to debug a program you have written. Documentation for the Python Debugger can be found here: *https://docs.python.org/3/library/pdb.html*

SUGGESTED PROJECTS

1. **Alternative Python Integrated Development Environments:** Visual Studio Code is not the only IDE you can use to develop Python code. Research alternative IDEs like those offered by Jet Brains *https://www.jetbrains.com*, and Microsoft Visual Studio *https://visualstudio.microsoft.com*. Microsoft Visual Studio is like Visual Studio Code's big brother.

2. **Modify calc_sums.py:** Modify example 3.3, calc_sums.py, to convert user input to either integers or floats, depending on what they enter at the console. Enhance error and exception handling to validate user string input before attempting to perform a conversion. Note that making

these changes will require you to learn features of Python not discussed in this chapter but covered later in the book.

3. **Validate Date of Birth String:** Modify example 3.4, calculate_age.py, to validate the format of the date of birth string entered by the user. Enhance error checking and exception handling to ensure a user enters a properly formatted date string before attempting a conversion. Note that making these changes will require you to learn features of Python not discussed in this chapter but covered later in the book.

4. **Python Arithmetic Operators:** Write a short program that uses all the Python arithmetic operators listed in section 2.5 here: *https://docs.python.org/3/reference/lexical_analysis.html#operators*. If you're unsure of an operators meaning research its usage.

5. **Convert A String To Uppercase Characters:** Write a program that prompts a user to enter a message, convert the message to all uppercase characters, and prints the results to the console.

6. **Convert A String To Lowercase Characters:** Write a program that prompts a user to enter a message, convert the message to all lowercase characters, and prints the results to the console.

SELF-TEST QUESTIONS

1. What type of object does the input() method return?

2. What Python Standard Library string utility method do you use to split strings into substrings?

3. In the following string, 'Jane, Steve, Sapna, Kateryna, Coralie, Heather', what character would you supply as an argument to the string.split() method to divide the string into substrings of names.

4. What type of object does the string.split() method return?

5. Can the built-in int() function convert strings that start with '+' or '-' characters?

6. Which built-in function would you use to convert the following string to a numeric type: '-23.45'?

7. Explain in your own words the difference between string concatenation and formatted strings?

8. Describe in your own words how a Python module is loaded and executed.

9. What's the difference between a syntax error and an exception?

10. What is a module's __name__ property set to if the module is imported into another module? What's it set to if the module is executed directly by the Python interpreter?

REFERENCES

Official Python Website, *https://python.org*

PEP 8 — Style Guide for Python Code, *https://peps.python.org/pep-0008/*

Python Standard Library, Python 11, *https://docs.python.org/3/library/index.html*

Python Package Index (PyPI), *https://pypi.org*

Python DateUtil Package on PyPI, *https://pypi.org/project/python-dateutil/*

Python Interpreter Reference, *https://docs.python.org/3/tutorial/interpreter.html*

NOTES