Python Debugging

RL Zimmerman

Table of Contents

1. Introduction

Debugging is the process of finding and removing "bugs" or defects in a program. To help my daughter with her first Python class, I looked around for information on debugging that I could share with her. I wanted a simple guide with everything in one place and suggestions for how to go about the process of debugging. Initially, my research focused on gathering examples of common issues, but I knew something more was needed. After all, what happens if there is no example of the "bug" that you're experiencing?

I knew I needed to provide a debugging "foundation." Not just how to use debugging tools, but when to take action and why. With that goal in mind, Chapters 1 through 6 build a debugging arsenal, so you're ready to tackle the examples in Chapter 7. Each example begins with "References" to the related topics covered in earlier chapters. So theoretically, you could jump right into the examples in Chapter 7.

This book includes an extensive and detailed Table of Contents. I also made a point to cross-reference topics so you can easily locate whatever you're interested in from any point in the material. This approach means you can pick up the book at any time and quickly jump back in where you left off. Or if you prefer, you can hop around from topic to topic with as much detail as you want.

Hopefully, after reading this book, you won't feel like this man who posted a plea for help on a chat board. His frustration show's through

in his comment, “For the love of God, how is this done?” Instead, you’ll know exactly how it’s done and have fun doing it!

1.1 Overview

How you may ask, are we going to build your debugging arsenal? Let’s begin with these topics.

- How to use the debug environment.
- The Python Error Codes and specific examples of how they happen.
- Step-by-step instructions on the process of debugging code.
- Finding the information you need to **modify your program**: help on Syntax, Functions, Classes, and more.

The goal of debugging is a working program, and debugging is just part of the process of writing code. When I realize I have a “bug,” I’ll experiment and try a few things to find a clue where the issue is. You’ll see this process in the examples in Chapter 7, where I use different approaches from my “debugging toolbox” to isolate an issue. You might take a different approach to the sample problem, and there is no wrong approach. The idea is to try a few things and see what works.

In this book, I demonstrate Python using the open-source Anaconda Data Science Distribution that includes Python version 3.7. Spyder, the Scientific Python Development Environment, comes with Anaconda. The Spyder IDE, or Integrated Design Environment, includes an Editor, Console, and debugging tools. You may notice slight differences in screenshots, depending on whether I am using Spyder on my Windows or Mac computer.

1.2 What This Book is About

My intent in writing this book was not to provide a guide to Python Programming. Instead, this book is specifically about debugging Python with Anaconda’s Spyder application. The concepts around Python

debugging apply equally to other environments, but the screens and debugging tools may vary slightly.

You may wonder why I’ve included Python Basics in Chapter 3. I found it difficult to explain an IndexError without first explaining data structures and their indexes. Similarly, a Dictionary KeyError doesn’t mean much without an understanding of a Dictionary. Syntax errors are fairly obvious in Spyder, but it doesn’t hurt to have a brief explanation of the syntax the parser expects.

Finally, Chapter 6 demonstrates how to view values, types, and the length of objects. Since the syntax varies by the type of object, I wanted to provide a reference with the exact syntax for each object type.

1.3 What’s Next?

The next chapter walks you through installing Anaconda and the basic Spyder environment. We’ll also look at an overall plan for debugging code.

2. Debugging Overview

In this Chapter we discuss

Writing code begins with your vision of what the program should do. You write code, see what happens, and make changes along the way. When the code doesn't do what you want, debugging helps you zero in on what's happening while the code runs. In essence, you can pause program execution and "freeze" your program at that point in time, looking at variable and object values at that moment.

This Chapter outlines a few suggestions to approach programming and debugging. The Examples in Chapter 7 follow a similar methodology.

Intended Outcome: What I wanted the program to do.

Actual Result: What the program did.

Incorrect Code: A look at the code before any changes.

Debugging Experiment: What I suspect is wrong with the program, and the steps I tried to "debug" what the program is doing.

How to Resolve the Issue: A brief description of the change to the code to achieve my "Intended Outcome."

Correct Code: The finished code that works as I intended.

2.1 Plan for Debugging

Programming is not my primary job. Instead, programming is a tool I use for data mining or organizing projects. A day in my programming life includes lots of interruptions. It may be weeks or

months before I pick up a project and continue coding. For this reason, I've adopted a few suggestions from programming friends to make my life easier.

1. Work on small chunks of code, test, and then move on to the next piece.
2. Keep multiple backup versions of your files.
3. Have a clear idea of what you want your program to do.
4. Use small data file samples that you know have clean data to develop your code. When you've tested your code and are confident there are no bugs, use live data connections or real data files.
5. Keep notes of where you stopped programming and the next steps.

Start Small

Write small chunks of code. Test and validate that piece of code, then move on. This "**Correct Code**" is also a good baseline for backups.

Keep Multiple Versions of Your Code

Keep multiple backup versions of your files. My backup files often include the date and time in the filename. That way, if I really mess up the code, I can easily go back to the "**Correct Code**" that worked earlier today or last month.

Intended Outcome

While I'm not suggesting you have a vision statement for your program, it doesn't hurt to have an "**Intended Outcome**" of what you're trying to accomplish. This synopsis is beneficial in several ways:

- Pair programming, or asking for another opinion.
- When you check-in your code to a source control program.
- During peer review.
- In a Sprint Review, where you demonstrate your program to others.

In case you reach out to another programmer for assistance, share as much information as possible.

1. The incorrect code. If you have the last working version of your code, that might also be helpful.
2. Your **Debugging Experiment** methodology, and what you've already tried.
3. The "Actual Result." What happens when you run the program?

Test Data Files

Web scraping and external data files can be messy and huge. Take a moment to familiarize yourself with samples of the live data or data dumps. Make small "test data files" or mock-ups of the data. Scrub the data to ensure it's clean.

If you plan to code for blank data, hidden characters, or type conversions, set aside a version of the data for that purpose. Initially, keep the test data as simple as possible. Use these test data files to save time iterating through thousands (or millions) of rows of data.

Look for hidden characters and blank cells or data elements. Make notes of data types and other issues as a reminder to add logic to your code to handle the data correctly.

Often, when you dump database data to a CSV or Excel file, there is an error when you try to open the file. For example, if you open an Excel file, it may prompt you to "fix" the data on file open.

Chapter 4 has an example of a mock-up HTML data file in the topic "Create Test Data."

Plan for Tomorrow

When you're done for the day or decide to take a break, leave a note for yourself of where you stopped. Include what is or is not working and what you want to do next.

Review your pseudocode and **Intended Outcome** to be sure you're on the right track. Pseudocode is an outline of your program design in simple terms, often written in plain English. These types of notes remind me of where I left off programming and what I need to work on next.

2.2 Debugging Steps

In some ways debugging is more of an art than science. Since I'm analytical, I am more inclined to use the scientific method for my debugging. It's really up to you to decide on your style of debugging, and if you'll use any of these suggestions.

1. When debugging, keep a **logbook** of your experiments, so you know what you've already tried.
2. Divide and conquer. Divide the code in half and test each half to see which part has the error. Repeat these steps to drill down to the location with the error.
3. Make a backup of your files before starting your experiments.
4. Start with a clear **Problem Statement** of the defect.
5. Don't believe everything you hear. If the original defect is the program works with Oracle data and not Cassandra data, verify that is really the case.
6. Examine the environment.
7. Create a list of possible suspects.
8. In case you're out of ideas and haven't found the defect, take a break. Work on something else, go for a walk or come back to the problem tomorrow.

Logbook

Keep a logbook of your debugging experiments. Write down the steps and outcome for each task. I find writing down my issues frees my mind from worrying about the problem, and allows me to brainstorm at my leisure.

Divide and Conquer

When debugging, pick a logical point to divide the code in half. Use a process of elimination to drill down to the error in the code.

1. Divide the program into **Part 1 Code** and **Part 2 Code**.
2. Run **Part 1 Code**. If there are no errors, you know that Part 1 is working. If you have errors, divide **Part 1 Code** again. Repeat the process until you drill down to the root cause.
3. If **Part 1 Code** ran without errors, run **Part 2 Code**. If you find an error, divide **Part 2 Code** and repeat the steps.

Wherever possible, eliminate the code that is unrelated to the error. Chapter 4 has an example of skiping unrelated code in the topic "Focused Testing."

Backup Files Before Debugging

Create a backup of your files before you change anything.

Problem Statement

Develop a clear problem statement with as much detail as possible. Who can you contact for more details? When determining how critical the issue is, consider the impact to business and if there is a workaround.

Doubt Everything

Verify the accuracy of the original defect report by recreating the issue yourself.

Look Around Your Environment

Before creating a list of possible causes, gather background on the environment.

1. Has the program ever worked?
2. When was the last time the program ran successfully?
   - Did it work last month?
   - Is this the first time it ran on a Monday or on the first day of the month?
   - Is there heavy load on the environment because it's the end of the month or quarter?
3. Can you connect to devices outside of the program successfully? Can you query the Cassandra database outside of your program? Is the web server responding to requests? Is one of the integrated systems down?
4. Did the program encounter an Out of Memory error?

Create a List of Suspects

As a starting point for your experiments, make a list of components that could be causing the defect.

- Your app
- The last few lines of code you changed
- Python language

- OS
- Connection to a web page
- The format of a database table or web page changed
- More than one library with the same name in different paths
- One of your script files with the same name as a library

What Do You Think is the Cause?

Chances are, at this point you have some idea where you want to start investigating. Make a list of your ideas or hypothesis of what might be wrong.

Refine Your Experiment

As you refine your **Debugging Experiment**, you'll probably notice parts of the code that you don't need to test. Your goal is to narrow the search by removing things that don't contribute to your hypothesis. Chapter 4 has an example that narrows your search in the topic "Focused Testing." Modify your program to temporarily eliminate these items from your experiment. Consider hard coding values or using temporary mock-ups of data.

Experiment

Change one thing at a time, and observe what happens. Please, write everything down in your logbook, noting each step and the outcome. The simple act of writing down my experiment forces me to pause and consider what happened and why.

- What steps did you take?
- What did you expect to happen?
- What actually happened?

Review the experiment and see if you can come up with a theory about the cause of the defect.

- Is there something you should not see?
- Do you need to refine your experiment further?
- Do you have a theory about what might be causing the defect?
- Do all your test results fit in with your theory, or is there one result that doesn't quite fit? Don't ignore the evidence that contradicts your theory. If you you aren't sure how that piece of

code works, dig into the code because that might be where the problem lies.

> Keep a log of the things you've tried as you debug your program, to avoid repeating the same tests.

Success at Last

The last experiment you conduct that unequivocally works is the fix. The program does what you want, and you reach your "**Intended Outcome**."

2.3 The Debugging Environment

For this book, I am using the Anaconda Distribution that includes the Spyder application. My Anaconda programs support Python 3.7.

Python

Python is an open-source (free) programming language for Web Development, GUI development, Scientific and Numeric data science, Software Development, and System Administration. The examples use the open-source Anaconda Data Science Distribution that includes Python version 3.7.

Spyder, the Scientific Python Development Environment, comes with Anaconda, and I run Python scripts in Spyder primarily on a Windows machine. For variety, I've also included several examples on a MAC computer.

In this Chapter, we'll install Anaconda and set up your environment. If you are familiar with Python and want to jump into debugging, feel free to skip ahead to Chapter 4.

Anaconda

Download the Anaconda Distribution that includes Python version 3.7. Other Python versions may vary slightly compared to the examples in this manual. When prompted, update your path settings. The install takes a while, so you might want to grab a cup of coffee or something.

Spyder

Spyder is an Integrated Desktop Environment or IDE. Spyder includes an Editor, Console or Spyder Shell, Variable Explorer, Help module, and other tools. These modules are displayed in "Panes" in Spyder.

On a Windows machine, launch Spyder from the Start Menu, in the Anaconda folder. On a MAC computer, open the Anaconda Navigator and launch "Spyder."

The Spyder Default Layout has three panes, as shown below. You can return to this layout at any time from the View menu under Windows Layouts. You can close or open other panes to suit your preferences.

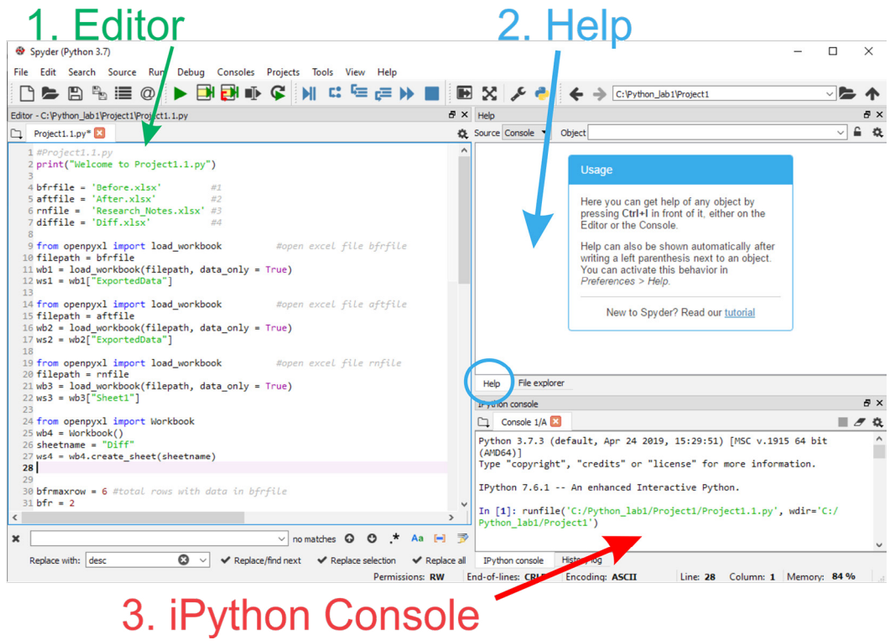


Figure 2.1 Project Files

1. The Editor window is where you type your code and create your script files.

2. The Help window displays syntax, function help, and more.

3. The iPython **Console** or Python shell. When you start Spyder the Console prompt is **In[1]:**.

   When you click "Run," the results are output to the Console. When you type a command in the Console, Python immediately runs the command. The Console is useful when debugging, or experimenting with different statements for your code.

   Results displayed in the Console include code output and error messages. For example, if you use the print() method, the

results are output to the Console window. In the example below, the **Console** displays "**Welcome to Project1.1.py**."

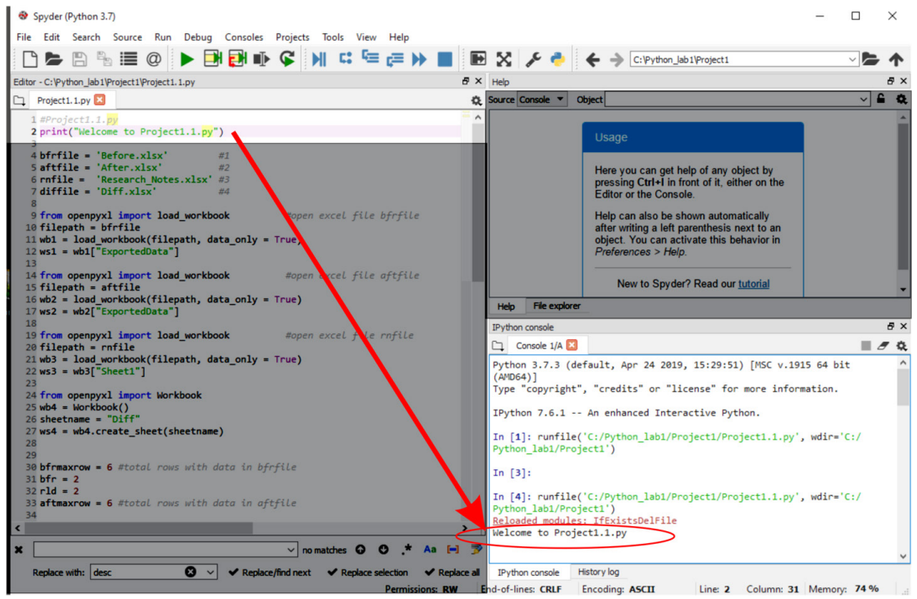


Figure 2.2 The iPython Console

Run a Script or Program

With Spyder open, click on the File menu and then click on "Open" and open, or create, a script file. In the next example, the "*Project1.1.py*" file is open in the Editor.

Click on the green arrow ▶ or use the **Run** menu, as shown below.

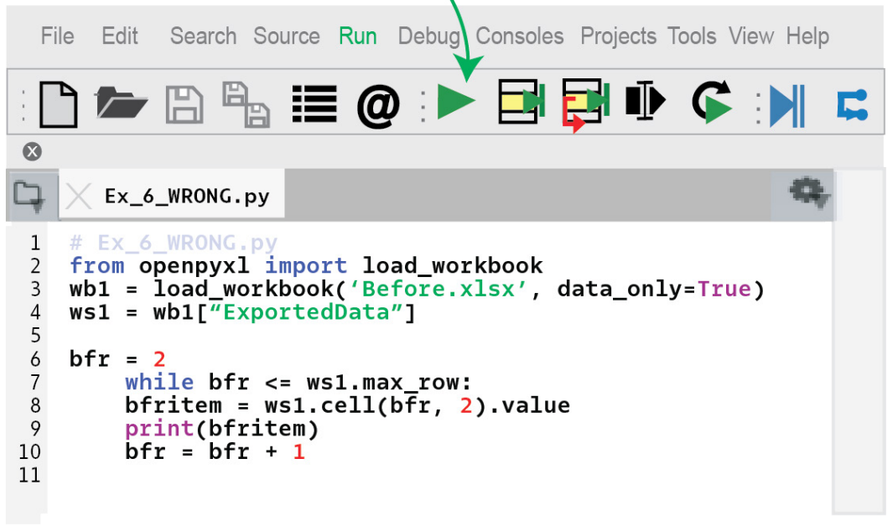


Figure 2.3 Run the Program

> In the Run menu select ▶ "Run selection or current line" to run only the selected lines of code.

In the next figure, I have two panes open. The Editor is on the left, and the **Console** window is on the right. Initially, the **Console** window displays the prompt **In [1]:**. After I click "**Run**," the Console window changes, as shown below. The first output line displays the name of the program file and the working directory.

In [1]: runfile('C:/Python_lab1/Project1/Project1.1.py', wdir='C:/Python_lab1/Project1')

Welcome to Project1.1.py

In [2]:

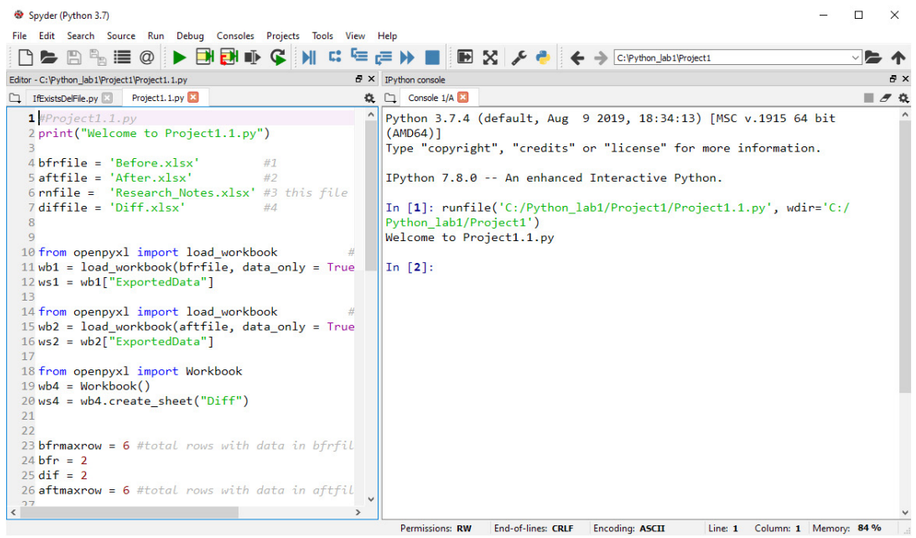


Figure 2.4 The Console

Stop a Program or Restart the Kernel

Click within the Console window and press "Cntrl C" to stop program execution with a keyboard interrupt. You can also select "Restart kernel" from the "**Consoles**" menu.

Adding a "**break**" statement to your code causes program execution within that code block to stop. For example, if you are inside a loop the "break" statement halts the loop, exits the loop, and continues running the next block of code.

2.4 Help

In Spyder, click on the **View** menu and click "Panes" to open the "Help" pane. Now, click on the **Help** menu and click on "Spyder Tutorial." The tutorial opens in the Help pane. The topic "Recommended first steps for Python beginners" is an excellent resource for new programmers.

Chapter 4 demonstrates Debug Mode, Interactive Mode, and Variable Explorer. These tools look at your code while it's running, in effect, "debugging."

To open Help for an object, place the cursor on an object name in the **Editor**, press Ctrl-I, or Command-I on a MAC. Help inspects the object and gathers docstring information.

2.5 What's Next?

Your Lab environment is now setup. Let's move on to Chapter 3 and review a few basic Python language guidelines.

3. Python Basics

In this chapter we discuss

Statements or Expressions

Python Syntax

Objects

Immutable Objects

Variables

Types of Data

Numbers

Strings

Data Structures

Now that your environment is set up, we'll take a brief look at a few basic Python concepts. Syntax and runtime errors often involve incorrect syntax, indentation errors, or a mismatch in object types. This chapter is by no means a complete Python language guide; instead, think of it as an abbreviated part of the Python language documentation. I need this small subset of information to demonstrate how you will refer back to the Python documentation as you debug your program. The "Appendix - References" at the end of this book has links to the Python documentation related to these topics.

3.1 Statements or Expressions

The actions that a program takes are referred to as "statements" or "expressions."

3.2 Python Syntax

The Spyder Integrated Development Environment (IDE) includes an Editor that warns you when you have a syntax error in your script. A yellow triangle on the left side of the Editor pane next to the line number indicates an error. Next, we'll look at a few common causes of syntax errors.

- Valid characters for variable names and identifiers vary between Python 2.x and Python 3.x. Python 3 added support for Unicode characters in PEP 3131 to accomodate programmers who are not familiar with the English language. To avoid errors, I adhere to these guidelines.
  - Identifiers begin with a letter.
  - Numbers are allowed in object names, except as the first character. Object names are also known as identifiers.
  - In Python 2.x, the only special character allowed in an identifier name is an underscore. Instead of spaces in identifier names, try an underscore. Illegal spaces can cause a syntax error.
- The PEP 8 Style Guide suggests lowercase characters for identifier names and functions. Classes begin with an uppercase letter. For example, variables and list names are lowercase.
- Python is case sensitive. There is a difference between "myString" and "mystring." The Python Interpreter displays a NameError when there is a misspelled identifier.
- When defining a function or control statement, the line should always end with a colon.
- Text to the right of the # pound sign is a comment. You can add comments anywhere in the line.
- A data structure name should be plural, and items in the data set should be singular. For example, a **List** named "vacations" with List items: vacation[0], vacation[1], etc.
- Do not use reserved keywords as identifiers. A missing keyword causes a SyntaxError.

- Unpaired parentheses cause a SyntaxError.
- An empty Suite (indented block of code) is illegal. See "Indented Code (Suite)" or Example 7 for more information.

Python has reserved **keywords** like "global" or "try." When you use a keyword as a variable name, it causes a syntax error.

These Chapter 7 examples illustrate a few syntax errors:

Example 7
Example 8
Example 9
Example 10
Example 11
Example 12
Example 16

Referencing Object Values

One of the first questions I had when I started using Python was, "How do I get the value inside of a variable." At any time you can type the object name in the Console, and the Python Interpreter will display the value at that moment. Here my variable "mystr" has a value of "apple."

In [**1**]: mystr

Out[1]: apple

Chapter 6 walks you through examples with syntax for the various objects including items inside of data structures, or objects inside of improted modules.

Variables in Imported Modules

To reference a variable inside another module, use dotted notation. In this script, I import a module "mymodule2" that has the variable "mystr2." The expression **module2.mystr2** returns the value of **mystr2**.

```
import mymodule2

print(mymodule2.mystr2)
```

```
1  import mymodule2
2  print(mymodule2.mystr2)
```

3.3 Objects

An object is a collection of data. Everything in Python is an object. Objects have an identity, type, and value. The "identifier" or "identity" of the object is the name of the object. With the library "**openpyxl**," you assign **objects** to both the workbook and worksheet, and then you use those **objects** with **methods** to read or update values (the data).

```
1  from openpyxl import load_workbook
2  wb2 = load_workbook(aftfile, data_only = True)
3  ws2 = wb2["ExportedData"]
```

3.4 Immutable Objects

Python string and number types are immutable, which means the values can not be changed. You can **not** change an existing string/int/float, but you can create a new string with changed data.

If you're new to programming, this concept may seem strange. Take the case of a Python object of the type "**int**." The code statement **bfr = bfr + 1** seems to change the value of **bfr**. In reality, this statement creates a new object. The new object has a new identifier and a different location in memory. To see this in action, run this code in the **Console** to see the identifiers for the **bfr** objects.

```
In [1]: print(id(bfr))
Out [1]: 12345678
In [2]: bfr = bfr + 1

In [3]: print(id(bfr))
Out [3]: 9876543210
```

The comparison operator "is" returns "True" when two variables point to the same object in memory, as shown later in the topic "Operators."

Immutable objects are quicker to access, and this improves code performance. Another advantage of immutable objects is understandability, and knowing the object will never change.

3.5 Variables

Think of a variable as a container to store values. When a program runs, the value inside the **variable** may change. A letter or number is a value. An **assignment statement** creates a variable and assigns a value, as shown below. The left side of an assignment statement must be a variable.


```
mynumber = 2000000
```

The Python style guide suggests that variable names begin with a letter.

Global Variables

A global variable is available at any part of your program. Normally variables are available within the active or "local" scope. When you create objects inside a function or method, those objects or variables are not available outside of that function, unless you use global variables.

Later in this chapter the topic, "Scope, Namespace, and Memory" looks at scope. Also, in Chapter 4, the topic, "Variables and Objects in Memory" uses Variable Explorer to see the scope of objects.

Take for example a program that has "var1" in the main body of the program (lines 5-7), and "var2" (on line 2) within a function.

Run this code in debug mode and step through the code running one line at a time.

1. The code moves from line 1, the function definition, and skips over to line 5.
2. The Python Interpreter then runs line 6.
3. If you choose to "**step into**" the function on line 6, the code moves up to line 1.
4. Next the code moves inside the function, to lines 2 and 3. At this point you'll see Variable Explorer displays both var1 and var2.

5. The program then skips back to line 7. At this point, var2 disappears from Variable Explorer, and that "scope" is gone.

```
1  def myfunction():
2       var2 = 4
3       print(var2)
4
5  var1 = 7
6  myfunction()
7  print(var1)
```

In the next block of code, let's explore using "var1" inside of "myfunction." This code runs as expected and prints "var1" on line 2, because I'm not trying to change the value of var1.

```
1  def myfunction():
2       print(var1)
3       var2 = 4
4       print(var2)
5
6  var1 = 7
7  myfunction()
   print(var1)
```

However, it's a different story if I attempt to change "var1". This next code causes an error, and the **Console** displays:

"**UnboundLocalError**" local variable 'var1' referenced before assignment.


```
1  def myfunction():
2       print(var1)
3       var2 = 4
4       var1 = 9
5
6  var1 = 7
7  myfunction()
8  print(var1)
```

The solution is to declare "var1" a global variable inside the function on line 2. Notice there is no "**assignment**" value. When line 3 prints "var1" it has a value of "7." Line 5 changes the global var1 to "9." When line 9 prints "var1," the value is now 9.

```
1  def myfunction():
2       global var1
3       print(var1)
4       var2 = 4
5       var1 = 9
6
7  var1 = 7
8  myfunction()
9  print(var1)
```

3.6 Types of Data

Python has several types of data. Numeric values such as "floats" and "ints" are scalar objects, in that there is no internal structure. A string is a non-scalar object, and you use an index to indicate the position within the string. Moving through items using an index is referred to as iteration.

Containers are non-scalar objects with internal structures. Examples of container objects are a list, tuples, dictionary, or range. A range was introduced with Python v3. We'll look at these data structures in depth in later topics. For now, a few of the basic data types. are shown in the next table.

Type	Description	Assignment	Value(s)
int	integer	my_var = 3	3
float	floating-point number	my_var = 3.85	3.85
bool	boolean (true/false)	my_var4 = false	false
NoneType	Function with no return value	myfunction()	None
str	string of characters	my_var2 = 'Hi'	Hi
tuple	any type of data - immutable	mytuple = ('Hi', 4)	Hi, 4
list	any type of data - mutable	mylist = [4, 9, 'hi']	4, 9, hi
range	integers - immutable	range(4, 9)	4, 5, 6, 7, 8

Table 3.1 Data Types

When my_var = 3, the statement **float(my_var+5)** returns **8.0**.

When my_var = 3, the statement **print(34//my_var)** returns **11**.

In the case of the statement **3= 3**, the Python Interpreter returns "True."

What is the Data Type?

If you're unsure of an object's type, the **type()** function displays the type of data. The second statement uses the "isinstance" function that returns "true" when an object is the specified type. In this example, I am testing if the "mystr" is a "str" type. To see this in action, type the code in the Editor and click "run."


```
1  print(type(mystr))
2  print(isistance(mystr, str))
```

Later in this chapter we'll look at identifying types of objects in dictionaries in the topic, "Find the Type of a Dictionary Element."

Converting Data Types

When working with data, you may need to change or convert the data type. For example, during a calculation, you may want to convert between a **float** and an **int** to remove decimal places. When concatenating numeric values and strings, you would convert the integer value to a string with the statement str(**my_int**).

int(my_var)

str(my_var)

float(my_var)

bool(my_var)

The 'NoneType' Value

In Python the absence of a value is called "None" which must be capitalized. In other languages this would be a null value. A function with no return statement also returns the value "None." When working with external data sources you may have to account for this type of value, as shown in Example 17. An "if statement" that tests for a value of "None" is shown below.

```
1  if myvar is not None:
2      pass
```

While I could compare the type of "mystr" on line 2 to **type(None)**, the preferred expression is to use "isinstance", as shown on line 5." The expression on line 2 would return "**True**."

```
1  mystr = None
2  if type(mystr) is type(None):
3      print('the type of mystr is None')
4
5  print(isinstance(mystr, str))
```

3.7 Numbers

Floats and integer types represent numbers in Python, and are scalar objects with no internal data structure. When assigning integer values, do not use commas. Python interprets 2,000,000 as three integers separated by commas.

mynumber = 2000000

In the previous example, I assign 2000000 to the integer variable "**mynumber**." For readability, you can add underscores as a separator.

mynumber = 2_000_000

The function **repr()** displays a printable representation of a float, and is useful in troubleshooting rounding errors.

3.8 Data Structures

Python has several built-in compound data structures or sequence types for non-scalar objects. These data structures have an ordered sequence of elements. That is not to say the items are arranged in a particular order, but rather that Python assigns a sequence of indexes to the items. You use an index to access the value of a particular element.

- Lists
- Tuples
- Strings
- Dictionary
- Range

Python uses the the operations listed below for all of these data structures. Later in this chapter we'll look at these common operations, and you'll see the syntax is the same regardless of whether you're working with a list, tuple, string, or dictionary. So, for example, the function len() tells you how many objects are in the data structure. In the case of a string, len() tells you how many characters are in the string. For a list, len() tells you how many elements are in the list.

- Indexing
- Slicing
- len()
- Comparison operators "in" and "not in"
- Control loops like "for"

With the exception of a range, you can use concatenation and multiplication on data structures. For example, to concatenante two lists using the plus "+"symbol use the expression **mylist1 + mylist2**.

- Concatenation
- Multiplication

Tuples also support concatenation, as shown below.

```
In [1]: mytuple = (1,'two', 3)

In [2]: mytuple
Out [2]: (1, 'two', 3)

In [3]: mytuple + ('H', 2, 'O')
Out [3]: (1,'two', 3, 'H', 2, 'O')

In [4]: mytuple
Out [4]: (1, 'two', 3)
```

3.9 Strings

A **string** is a sequence of characters. These non-scalar objects have an internal data structure accessed through indexes. To assign a value to a string variable, use single or double quotes, as shown below.

b = 'bookstore'

Strings can be concatenated, indexed, and sliced. In the previous example, the index for the letter 's' is b[4], because Pythong starts counting at 0. String indices must be integers. We'll look at string indices and slicing in the next topics.

Strings are immutable and can not be changed. Later in this chapter, the topic "Append to Dictionary" we look at an AttributeError caused by trying to change a string value in a dictionary.

To assign a value to a string, use the same syntax and, in effect, create a new string variable with the same name. The new string has a different "identifier" and location in memory, as discussed earlier in the topic "Immutable."

When comparing values, you may want to insure an object is of type "string", and account for uppercase and lowercase. The string methods upper() and lower() convert a string. The example below

converts a variable to a string with all uppercase letters, and is also shown in Example 7.27.

str(my_var).upper()

Occasionally, you may run across whitespace or a special character like the new line '**\n**' character. The function **repr()** displays a printable representation of a string including whitespace, and is demonstrated in Eample 7.28

> To see the methods available to a string variable, type **dir(my_str_var)** in the Console. Or, type **help(str)** for more detailed information.

String Methods

Let's take a moment to look at some of the common string methods, as well as the "String Module." After creating a string variable "mystr," in the **Console** type "**dir(mystr)**" to see additional information.

Syntax	Comments
`mystr.isalpha()`	
`mystr.capitalize`	Converts to Camel Code
`mystr.`**`count`**`('39')`	How often is '39' in "str"
`mystr.`**`find`**`('39')`	index of first occurrence of '39'
`mystr.index('9')`	Returns the index for '9', or returns error if not found
`mystr.rindex('9')`	Same as index but counting from right
`mystr.`**`lower`**`()`	change to lowercase
`mystr.lstrip`	Remove whitespace on the left
`mystr.rstrip`	Remove whitespace on the right
`mystr.replace`	
`mystr.split`	

Syntax	Comments
`mystr.swapcase`	
`mystr.upper()`	change to uppercase

Table 2.1 String Methods

The String Module

In addition to the built-in string methods we just looked at, there is a "string" module with several invaluable methods. The string module is useful to create strings of ascii characters. The next chart shows a few of the functions in the strings module. In the Console , after importing the string module, you could also type "**help(string)**" to see more information.

Syntax	Comments
`.ascii_letters`	both lower and uppercase letters
`.ascii_lowercase`	abcdefghijklmnopqrstuvwxyz
`.ascii_uppercase`	ABCDEFGHIJKLMNOPQRSTUVWXYZ
`.digits`	'012345689'
`.punctuation`	!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
`.whitespace`	' \t\n\r\x0b\x0c'
`.printable`	all printable characters

Table 2.2 Some String Module Methods

Create a String of Lowercase letters

To create a string of lowercase letters use this syntax shown below. Notice in line 4 I convert the new string into a list.


```
1  import string
2  all_ltrs = string.ascii_lowercase
3  print(all_ltrs)
4  all_ltrs_list = list(all_ltrs)
5  print(all_ltrs_list)
```

And the Console shows:

In [2]:
abcdefghijklmnopqrstuvwxyz

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
'v', 'w', 'x', 'y', 'z']

Whitespace Characters

To see "whitespace" characters use the **repr()** function as shown below. On line 3, I use the repr() function to display the printable representation of the whitespace characters.

```
1  import string
2      all_ltrs = string.whitespace
3  print(repr(all_ltrs))
```

And the Console shows:

In [3]:
\t\n\r\x0b\x0c

Looping Through Strings

String indices must be integers. The next example of a "for" loop is perfectly legal and prints the messagem 'Hello World.'

```
mystr = 'abc'
for i in mystr:

    print('Hello World')
```

What happens if you want to print the string values using the index notation? The "print" statement shown below would cause an **error**, because the values 'abc' are not integers .

```
1  mystr = 'abc'
2  for i in mystr:
3
       print('mystr char is:', mystr[i])
```

The **Console** displays a traceback message with a "**TypeError.**" I've abbreviated the Traceback message below for readability

In [2]:

```
Traceback (most recent call last):
TypeError: string indices must be integers
```

A slight modification in the code would prevent the error. In the example below, I am using the "range()" function combined with the length function "len()" to get the length of the list. We'll look at "range" in detail later in this chapter.

```
1  mystr = 'abc'
2  for i in range(len(mystr)):
3
       print('mystr char is:', mystr[i])
```


3.10 List

A **list** is an ordered collection of values of objects. Lists are usually of the same type, but can be a combination of types. A list is similar to an array in other languages and contains a sequence of elements. A unique index refers to each list item, and the index is an **integer**. When creating Lists use square brackets **[ ]**. The index is used when updating a particular list item.

```
ws1 = wb1["ExportedData"]
```

List values are mutable; the values can change. Because lists are mutable they cannot be used as Dictionary **keys**. Lists can be used as Dictionary "**values**" as demonstrated later in the topic, "Append to a Dictionary." A list can grow or shrink as needed.

> Python starts counting at **0**. The first item in a List has an index of **0**, and the second item has an index of **1**.

When creating a List, use brackets **[]**. A List will usually have homogeneous data; but can mix different data types. Commas separate items.

Description	Syntax	Comments
Create a **List** and assign values	`mylist = ['a', 'b', 'c']`	
Create a **List** and assign number values	`mylist2=[1,2,3 4]`	
Assign a value to the first item in the **List**	`mylist2[0] = 8`	
Access the value of the **List** item	`mylist2[1]`	Returns the value of the first item in the **List**

Description	Syntax	Comments
Access the value of the last **List** item	mylist2[-1]	Use negative index numbers when counting from the right
Return all items in a List	mylist	

Table 2.3 List Objects

Iterate Through Items in a List

A "for loop" is one option to iterate through items in a List, as shown below.

```
1  for j in mylist:
2          print('mylist item is:', mylist[j])
```

These Chapter 7 examples illustrate a few List errors:

Example 7.1
Example 7.2
Example 7.3

List Methods

There are several handy methods that work with lists.

Syntax	Comments
mylist.remove(2)	removes '2' from mylist

Syntax	Comments
mylist.pop(2)	deletes and returns '2'
del(mylist[1])	deletes second item in mylist

Table 2.4 String Functions

Change List Type

In case you have a "string" object, and want to use a "list" method with the string, convert the string to a list using "l**ist(mystring)**." To change back to a string type use "**str(mystring)**."

```
1  mystrvar = 'hello'
2  list(mystrvar)
```

And the Console shows:

In [2]: mystrvar

['h', 'e', 'l', 'l', 'o']

Split List

The split() function is useful for splitting strings. If no argument is give, split() assumes a space.

```
mystr = 'hello world'
mystr.split()
```

And the Console shows:

In [2]: mystr
['hello', 'world']

Join Lists

The join() function is useful for joining strings and create a new string. In this example the "_" underline character is used as an argument. If no argument is given, split() assumes a space.

```
mylist = ['h', 'e', 'l', 'l', 'o']
mylist2 = '_'.join(mylist)
```

And the Console shows:

In [2]: mylist2
h_e_l_l_o

Insert an Item to a List

To add an item "a" at position '2' use the following expression.

```
mylist = ['h', 'e', 'l', 'l', 'o']
mylist.insert(2, 'a')
```

And the Console shows:

In [2]:
['h', 'e', 'a', 'l', 'l', 'o']

Extend a List

The function "extend()" adds exactly one element to the end of your list.

```
mylist = ['h', 'e', 'l', 'l', 'o']
mylist.extend.('u'))
```

And the Console shows:

In [2]:
['h', 'e', 'a', 'l', 'l', 'o', 'u'']

Append an Item to a List

The function "**append()**" adds a single element to the end of your list. Append changes your original list.

```
mylist = ['h', 'e', 'l', 'l', 'o']
mylist.append(' there')
```

And the Console shows:

In [2]:
['h', 'e', 'a', 'l', 'l', 'o', 'there']

Delete an Item from a List

The function "**del(mylist[4])**" deletes the argument 'o' from your original "mylist" object.

```
mylist = ['h', 'e', 'l', 'l', 'o']
del(mylist[4]
print(mylist)
```

And the Console shows:

In [2]:
['h', 'e', 'a', 'l', 'l', 'o', 'there']

Remove an Item from a List

The function "**mylist.remove('o')**" removes that item from the list.

```
mylist = ['h', 'e', 'l', 'l', 'o']
mylist.remove('o')
print(mylist)
```

And the Console shows:

In [2]:
['h', 'e', 'l', 'l']

Pop (Remove) an Item from a List

The function "**mylist.tremove**" removes the last item in the list, or the item where you provide the index. The function "pop()" also returns the item you remove.


```
mylist = ['h', 'e', 'l', 'l', 'o']
pop_item = mylist.pop(4)
print(pop_item)
```

And the Console shows:

In [2]:
o

Find Index Location of an Item in a List

To find the index for the item 'e' use the following expression, which returns "1".

```
mylist = ['h', 'e', 'l', 'l', 'o']
mylist.index[1]
```

And the Console shows:

In [2]:
1

Sort Items in a List

There are two functions to sort list items. The "**sorted**(mylist) function returns a new, sorted version of your list. The mylist.**sort**() method changes the original "mylist" into a sorted list, and returns nothing.

```
1   mylist = [‘h’, ‘e’, ‘l’, ‘l’, ‘o’]
2   print(mylist.count(‘l’))
```

And the Console shows:

In [2]:
2

3.11 Tuple

A **tuple** is similar to a list in that it is an ordered sequence of elements. That is not to say the items are arranged in a particular order, but rather that Python assigns a sequenc of indexes to the items. The first item in a tuple has an index of "0." Tuples use parentheses **( )** and items are separated by commas.

Tuples can also contain tuples. In this next example, when I create "mytuple2," the fourth item is "mytuple1." Notice mytuple2 includes numbers, strings, and a tuple. Tuples can have heterogeneous data, meaning tuples can have different data types.

In [1]: `mytuple1 = (1,'two', 3)`

In [2]: `mytuple2 = ('H', 2, 'O', mytuple1, 4)`

In [3]: `mytuple2`
Out [3]:`('H', 2, 'O', (1,'two', 3), 4)`

Swapping Values with Tuples and Functions

Tuples are useful for swapping values as demonstrated by this function that returns a tuple with two objects. The first few lines create the function. Line 7 invokes the function and assigns the function to "myint1" and "myint2." Because the function returns "var1" and "var2,"

the result is the same as the expressions that assign **myint1** = "var1," and **myint2** = "var2."

```
def myfunction()
          var1 = 3
          var2 = 4
          return var1, var2


myint1, myint2 = myfunction()
print('myint1 is:', str(myint1) + '; myint2 is:', myint2)
```

Slicing a Tuple

Earlier, I create mytuple2, and assigned these values: `('H', 2, 'O', (1,'two', 3) 4)`.When I use "slicing" with my tuple I can retrieve the value of the **fourth** item, as shown below. The index for the fourth item is "**3**," because Python starts counting at zero.

In [4]: `mytuple2[3]`
Out [4]:`('H', 2, 'O', (1,'two', 3), 4)`

Looking at this same example `mytuple2[3]`, how would I use slicing to retrieve the 2nd element "two" inside of mytuple1, which is inside mytuple2? Hint: because Python starts counting at "0" the second index of the second element is [**1**].

`mytuple2[3][1]`

Let's say I wanted to use the last element in mytuple2 for addition. Because "4" is inside of tuple2, I add a **comma**.

10 + mytuple2[**4**]**,**

This expression evaluates to "14."

Tuples are Immutable

If you try to assign a new value to an item in a tuple, a TypeError is raised. Tuples are immutable, and the value can not be changed.

```
In [3]: mytuple[1] = 'three'
Traceback (most recent call last):

  File "<ipython-input-3-db66c3391d15>", line 1, in <module>
    mytuple[1] = 'three'

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

Repetition and Concatenate

Tuples support both concatenation and repetition. In the examples that follow, I am using repetition and adddition for tuples. Because tuples are immutable and can't be changed, addition does not change the value of "mytuple." This behavior is illustrated in the expressions I typed in my **Console**, as shown below.

```
In [1]: mytuple = (1,'two', 3)

In [2]: mytuple
Out [2]: (1, 'two', 3)

In [3]: mytuple + ('H', 2, 'O')
Out [3]: (1,'two', 3, 'H', 2, 'O')

In [4]: mytuple
Out [4]: (1, 'two', 3)
```

	Expression	Returns:
Repetition	2 * ('H', 2, 'O')	('H', 2, 'O', 'H', 2, 'O')

	Expression	Returns:
Addition	mytuple + ('H', 2, 'O')	`(1,'two', 3, 'H', 2, 'O')`

Table 2.5 List Objects

Elements are separated by commas. Dictionary keys are always immutable elements, thus only Tuples can be used as Dictionary keys. Lists are mutable and can not be used as Dictionary keys.

> Tuple indices must be integers or slices, not strings.

Description	Syntax	Comments
Create a Tuple and assign values to 4 items	mytuple = ('a', 'b', 'c', 'd')	
Create an empty Tuple	mytuple = tuple()	
A tuple with one object	mytuple= tuple('fruit',)	Ends in a comma
Create Tuple with one item	mytuple2 = ('Rachel',)	Notice the comma at the end to instruct Python this is a Tuple and not a String.
Assign number values to several Tuple items	mytuple3 = (1, 2, 3)	
Assign a value to the first item in **mytuple4**	**mytuple4**`[0]='Apple'`	The first item in the Tuple has an index value of 0.
View the value of the 2nd Tuple item	**In [1]:** mytuple4[1] **Out [1]**: Orange	The Python Interpreter returns the value Orange.

Table 2.6 Tuple Objects

Iterate Through Items in a Tuple

A "while loop" is one option to iterate through items in a Tuple, as shown below.

```
mytuple4 = ('Apple','Orange', 'Watermelon')
j = 0
while j < 3:
    print('my fruit is:', mytuple4[j])

    j+=1
```

A "for loop" is another option to iterate through items in a Tuple. Indices must be valid integers. The two samples below are valid and do the same thing.

```
mytuple4 = (0, 1, 2)
for k in mytuple4:
    print('my number is:', k)
```

```
mytuple4 = (0, 1, 2)
for k in mytuple4:
    print('my number is:', mytuple4[k])
```

The sample code below is invalid and causes an "**IndexError**," because there are only three objects in the Tuple with values 1, 2, 3. Python starts counting at 0. The print statement is using "**mytuple4[k]**" or mytuple[1], mytuple[2], and mytuple[3]. When you run the program, the Python Interpreter warns that "**3**" is not a valid index because '**3**' is beyond the limits of the tuple.

```
mytuple4 = (1, 2, 3)
for k in mytuple4:
    print('my number is:', mytuple4[k])
```


3.12 Dictionary

A Dictionary is a set of key/value pairs and is a non-scalar object with an internal data structure. The key/value pairs are not in any particular order, and any type of object, including functions, ints, strings, tuples, or bools, may be used as a value.

Dictionaries are often depicted as two columns, with the list of keys in the first column, and values in the second column. Values can be duplicated, but keys must be unique. You can only use immutable elements as Dictionary keys. So, while tuples can be used as Dictionary keys, mutable lists can not be used as Dictionary keys. I would avoid floats as keys, given the way floats are actually stored in memory.

Key	Value
Name	John
Age	32
Height	5.10"

Table 2.7 Sample Dictionary

A Dictionary has unsorted elements that can grow and shrink as needed. When creating Dictionaries, use curly braces **{}**. The key is followed by a colon **:** and a paired value. In the example below, the first key is 'Name,' and the value is '**Zimmerman**.' The second key is 'Grade,' and the value is '**A**.' A comma **,** separates the key pairs.

Create a Dictionary

To create a Dictionary with three key pairs, use the following syntax. For readability, the key pairs are usually written in this format.

To create an empty dictionary type

```
mydictionary = { }
```

To create a dictionary with "string" values use the follow syntax.

```
mydictionary = {'Name': 'Zimmerman'
                'Grade': 'A',
                'Course': 'Python Programming'}
```

In the following example, I am creating a dictionary with "list" values. Because it is a "list" I use brackets.

```
mydictionary2 = {'Name': ['Young']
                 'Grade': ['B'],
                 'Course': ['Excel Fundamentals']}
```

Because a dictionary can have any type of object, pay attention to the object type. Strings are contained in **quotes**, tuples use **parentheses**, and lists use **brackets**.

Find the Type of a Dictionary Element

In case you've run across a dictionary and are wondering about the type of an object, let's look at how to find the object type in a dictionary. In the **Editor** I've created "mydictionary2." Notice the values are in brackets, indicating they are "lists."

```
mydictionary2 = {'Name': ['Young']
                 'Grade': ['B'],
                 'Course': ['Excel Fundamentals']}
```

After running this code in the **Editor,** I want to look at the type of the value in the key/pair. As shown below, in the **Console** I use the type function to determine the type of the value where the key is '**Name**'. The Python Interpreter returns "list."

```
In [2]: type(mydictionary2['Name'])
Out [2]: list
```


Now let's revisit the first dictionary and look at the type of the values stored.

```
mydictionary = {'Name': 'Zimmerman'
                'Grade': 'A',
                'Course': 'Python Programming'}
```

After running this code in the **Editor,** I want to look at the type of the value in the key/pair. As shown below, in the **Console** I use the type function to determine the type of the value where the key is '**Name**'. The Python Interpreter returns "string."

```
In [3]: type(mydictionary['Name'])
Out [3]: str
```

Append to a Dictionary

Why should you care about the object type? Well, you can "append" or add elements to a "list." If you try to add elements to a "string" you'll get an AttributeError, as shown below.

```
In [4]: mydictionary['Name'].append('Smith')
AttributeError: 'str' object has no attribute 'append'
```

The same syntax for "**mydictionary2**" is successful, because the value in the key/pair "Name" is a "list."

```
In [5]: mydictionary2['Name'].append('Smith')
```

To see the new dictionary values, in the Console I type "mydictionary2." Notice the output shows the key "Name" now has two values. The brackets indicate **['Zimmerman', 'Smith']** is a "list."

```
In [6]: mydictionary2
Out [3]: {'Name': ['Zimmerman', 'Smith'],
'Grade': ['A'],
'Course': ['Python Programming']}
```

How Many Elements are in the Dictionary List?

Now I can count the number of values associated with the key "Name" using the len() method to retrieve the length of the list.

```
In [6]: len(mydictionary['Name'])
Out [3]: 2
```

Assign a Dictionary Value Using the Key Name

To update a Dictionary value use the following syntax.

```
mydictionary['Name']: 'Smith'
```

Add a New Key to an Existing Dictionary

To add a new key, 'Credits' to an existing Dictionary, use this syntax.

```
mydictionary['Credits'] = '3'
```

Delete a Key in Existing Dictionary

To remove the key/pair ['Credits']:'3' from the Dictionary, use "del."

```
del (mydictionary['Credits'])
```

Iterate Through Key-Pairs in a Dictionary

This "for loop" returns the key-pairs in the Dictionary. The first line in this example creates two variables "mykey" and "myvalue" for the key-value pairs. The method **items()** returns a list of the key-value pairs.

```
for mykey, myvalue in mydictionary.items():
```



```
    print("\pKey: ", mykey, "\tValue: ", myvalue )
```

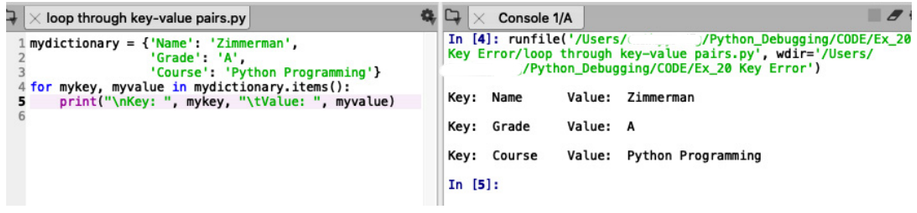


Figure 3.1 Print Key-Value Pairs

Iterate Through Keys in a Dictionary

This "for loop" returns the Keys in the Dictionary. To make your code easier to read, add the keys() method to the same statement.

```
for mykey in mydictionary:
    print("\pKey: ", mykey )
```

The next example is the same statement with the keys() method.

```
for mykey in mydictionary.keys():
    print("\pKey: ", mykey )
```

Find the Value of a Dictionary Item

This example uses the **Console** to display the value of the key "Name." Compared to a list where I need to know the correct index, with a dictionary I simply provide the name of the "key." The Python Interpreter returns the value "Zimmerman" to the Console pane.

```
In [3]: mydictionary['Name']
'Zimmerman'
```

To test if a particular key is in a dictionary, you could use the "in" operator. Continuing with our previous dictionary example, I might look for the keys "DoB" or "Course" with these expressions.

```
In [4]: "DoB" in mydictionary
Out [4]: False


In [5]: "Course" in mydictionary
Out [5]: True
```

Retrieve Keys

To retrieve the dictionary keys use the "**.keys()**" method.

```
In [6]: mydictionary.keys()
Out [6]: dict_keys(['Name','Grade', 'Course'])
```

Title and Value Methods

In this next example, I modified the previous code that returned the keys in the Dictionary. Here I use the **title()** method to access the Dictionary values.

```
for mykey in mydictionary.keys():
    print(mykey, ":", mydictionary[mykey].title())
```

Another way to access Dictionary values is with the **values()** method, as shown below. The first expression returns all values in the dictionary, while the second expression entered in the **Editor** iterates through each item.


```
In [7]: mydictionary.values()
Out [7]: dict_keys(['Name', 'Grade', 'Course'])


for myvalues in mydictionary.values():
    print(myvalues.title())


In [8]:


Zimmerman
A
Python Programming
```

3.13 Range

The range function was introduced with Python 3 and is used to generate a range of **integers**. A range is immutable and can not be changed. When you run an expression with a range, the Python interpreter creates the first integer in the range. The next integer is created when you ask for it, and so on. So you are not hampered waiting on Python to generate a large list of integers, it's more of a just-in-time approach. The format for a range is shown below.

```
for i in range(start: stop: step):
    print('Hello #', i)
```

If only one argument is provided, it is 'stop'. Start defaults to 0, and step defaults to 1. The range(0, 4) starts at index 0, and ends at index 3.

```
for i in range(1,4):
    print(i)
```

When I run this for loop from the **Editor**, it prints **1**, **2**, **3**, as shown below.

```
In [43]:
1
2
3
```

A range uses indexing, slicing, len(), the comparison operators "in" and "not in", and works with the "for" control loop. A range is ideal to iterate over a list. In the next example, the length of the list is the "stop" argument for range.

```
for i in range(len(my_list):

      print('The list item is:', my_list[i])
```

3.14 Set

A Set is unordered object and contains no duplicates. A Set is mutable and grows or shrinks as needed. When working with Sets, there are functions to perform Unions, Intersections, and find Differences. When creating a set, use curly braces **{}** and separate items with a comma. The second row in the table below creates a set with one item. Notice the line ends in a comma, to indicate to Python this is a Set and not a String.

Description	Syntax	Comments
Create a Set and assign values	myset = {'a', 'b', 'c'}	
Create a Set with one item	myset2 = {'John',}	
Create an empty Set	myset3 = set()	
Create a Set and assign values	myset3 = set('abc')	With set() function

Table 2.8 Creating Sets

3.15 Indexes

Strings, tuples, ranges, and lists are non-scalar objects with internal data structures. These objects use indexing to locate a particular element in the sequence. The format is the object name with brackets around the index. For example, **b[4]** evaluates to "s" in the string "b."

In Python the index starts at position 0. In the previous example, '**bookstore**' is assigned to **b**, and there are 9 characters. The start index is 0 and the end index is 8. If you go beyond the end of the index, it causes an "IndexError," as shown in Example 7.1.

b = "**bookstore**"
0 start index
8 end index
9 length of string

To find the length of string "**b**" use the len() function.
len(**b**) returns **9**

b	**o**	**o**	**k**	**s**	**t**	**o**	**r**	**e**
0	1	2	3	4	5	6	7	8

Table 3.2 String Index Example

len() is used with strings, tuples, ranges, and lists.

3.16 Slicing

Slicing is used with strings, ranges, tuples, and lists. "**Slicing**. breaks a sequence into a substring of elements. Notice in the example below, slicing uses brackets and takes 3 arguments separated by colons. The first argument "start" tells the function where to start splicing the string. Start defaults to 0, and "step" defaults to 1. If only one argument is given it is used as the "stop" argument.

stringvariable**[**start**:**stop**:**step**]**

The default for the second argument "stop" evaluates to len(stringvariable) + 1. Using the "bookstore" string follow along with this slicing example. Notice the stop index is **9**. Earlier we saw the function **len(b)** returns **9**, so the length of this string is **9** characters.

b = "**bookstore**"

b**[**4**:9:**1**]**

b**[**4**:9:**1**]** returns: **store**

The "stop" value "**9**" would evaluate to "**9**-1." Recall that Python starts counting at 0, so this slice **b[**4**:9:**1**]** stops at "8" and returns characters 0-8.

Slicing is often combined with the **len()** function, as shown below.

b**[**4**:len(b):**1**]**

The third argument "step" tells the function which characters to return. For example, step 2 would skip every other character. "Step" can be omitted. In that case you would type **b[**4**:len(b)]**.

The default Slicing values are:

Argument	Description	Default Value
start	start is the index to begin slicing	0
stop	(the stop index where you want to stop) evaluates to **stop value - 1**	len(b)
step	return every "**1**" character step "**2**" skips every other character	**1**

Table 3.3 Default Slicing Values

This example b**[**0:4**]** evaluates to **book**.

b**[**0:4**]**								
b	**o**	**o**	**k**	s	t	o	r	e
0	1	2	3	4	5	6	7	8

The next example b**[**4**:]**evaluates to **store**, because only the start argument is given. If you don't provide a "stop" argument, the default of len(stringvariable) + 1 is used.

b**[**4**:]**								
b	o	o	k	**s**	**t**	**o**	**r**	**e**
0	1	2	3	4	5	6	7	8

Notice the example below b**[**4**]** looks similar to the previous example, but omits the colon. Now Python returns only the character "**s**" at index **4**.

b**[**4**]**								
b	o	o	k	**s**	t	o	r	e
0	1	2	3	4	5	6	7	8

Negative values tell Python to start counting from the right. The example below **b[::-1]** evaluates to:

erotskoob

b[:: -1]								
b	**o**	**o**	**k**	**s**	**t**	**o**	**r**	**e**
0	1	2	3	4	5	6	7	8

This example **b[5:2:-1]** the "-1" step argument tells the Python interpreter to move right to left. This example evaluates to:

tsk

b[5:2: -1]								
b	o	o	**k**	**s**	**t**	o	r	e
0	1	2	3	4	5	6	7	8

In this example b[**-8**:**-3**:] a negative start argument "**-8**" counts from right to left, beginning with "o". There is no step argument, so the default of "**1**" is used; meaning you move left to right, from **-8** to **-7** to **-6**, and so on. The stop argument is "**-3**" telling Python to stop before it reaches **-3**. This slice evaluates to:

ookst

b[-8:-3:]								
b	**o**	**o**	**k**	**s**	**t**	o	r	e
-9	**-8**	**-7**	**-6**	**-5**	**-4**	-3	-2	-1

This example **b[-7:2:-1]** below evaluates to an empty string, because it goes beyond the end of the string. It starts at -7, moves to -8, and then -9. At that point it has moved 3 indices from right to left and can't move any more.

b[-7:-5: -1]								
b	o	o	k	s	t	o	r	e
-9	-8	-7	-6	-5	-4	-3	-2	-1

3.17 Operators

Now let's take a look at operators for numerical operations and comparisons.

Numerical Operators

Arithmetic operators work pretty much the way you would expect in Python.

Operator	Example	Description
+	x + y	the sum
<	x - y	the difference
*	x * y	the product
/	x / y	division
//	x // y	floor division: 5 /2 returns 2
%	x%y	modulo: the remainder when x is divided by y
**	i**j	i to the power of j

Table 2.9 Numerical Operators

You'll often see the remainder "%" operator used to identify odd or even numbers.

```
for i in range(0, 20):
```

```
    if i % 2 != 0:
print("i is an odd number", i)
```

One way to identify odd numbers is to use "**+=.**" Given that the first element in a tuple is mytuple1[0], I can increment a counter += 2 to iterate through all the odd elements in the tuple. The following example creates a new tuple with the first element, third element, and fifth elements from "**mytuple1**."

```
myTuple1 = (1, 2, 3, 4, 5)
myTuple2 = ()
i = 0
while i < len(myTuple1):
     myTuple2 += (myTuple1[i],)
     i += 2
print(myTuple2)
```

Comparison Operators

Use Comparison Operators to compare two values. The "in" and "not in" operators are handy for searching or finding elements in a data structure.

Operator	Description	Type
>	Greater than	Values
<	Less than	Values
>=	Greater than or equal to	Values
<=	Less than or equal to	Values
==	Equal (values)	Values
!=	Not equal	Values
is	Equal (boolean)	Boolean
is not	Not Equal (boolean)	Boolean
in	Test for membership	
not in	Test for membership	

Table 2.10 Comparison Operators

To test whether a character is in a string, use the "in" comparison operator as shown below.

```
mystr = 'apple'
if 'a' in mystr:
     print('a is in', mystr)
```

> The "in" comparison operator is used with strings, tuples, ranges, and lists.

The "is" comparison operator is also a useful way to see if two objects ultimately point to the same object in memory. So for example, let's say you create a variable "myvar1" and assign the value "hello." You then create a new variable "myvar2" to assign it to "myvar1" with the statement myvar2 = myvar1. In effect, you create an "alias" from "myvar2" to "myvar1". This statement returns "True" because the objects are the same.

```
myvar1 == myvar2
```

You could also use the id() function to verify these two variables point to the same object. In this example the identifier is the same.

```
In [14]: id(myvar1)
Out[14]: 140498313577904

In [15]: id(myvar2)
Out[15]: 140498313577904
```

Comparisons that Return True or False

When you compare two objects, the Python Interpreter returns "True" if the comparison is true, or "False" if the comparison is not true. In the Console type the following statement. Python returns "True"

In [1]: ‘apple’ == ‘apple’
Out[1]: True

The next two functions both return “True”. Because the second function “**myfunction2**” is simpler, it is considered more “Pythonic.”.

```
def myfunction():
    if 2 == 2:
        return True

def myfunction2():
    return 2 == 2

print(myfunction())
print(myfunction2())
```

3.18 Control Statements

Python control statements control the flow of the program. Examples of control statements are **For**, **While**, **If**, and **Else**. When the control statement is true, the indented lines that follow run.

- for
- while
- if
- else

The control statement always ends with a colon, and you indent the next line of code to the right. If you want to run several lines of code as part of the control statement, the lines are all indented to the same level.

The first line of the control statement, and all the indented lines that follow, is called a “Suite” in Python. Other programming languages often refer to this structure as a block of code.

> The “for” control statement is used with strings, tuples, ranges, and lists.

A control statement moves through items in a data structure. When this program runs, each time the program loops through the code, the next item in the list is displayed.

For Loop

To see a **for** loop in action, type this code in your Editor window, then click run.

```
fruits = [‘Apple’,‘Orange’, ‘Watermelon’]
for fruit in fruits:

    print(‘my fruit is:’, fruit)
```

The output of this code is shown below.

my fruit is: Apple
my fruit is: Orange
my fruit is: Watermelon

Iteration Variable

An iteration variable can also be used to iterate through list items. For example, when the variable “i” is a number with a value of “0.” The first time the loop runs list[i] refers to list[0]. The next time the list runs, list[i] refers to list[1].

```
fruits = [‘Apple’,‘Orange’, ‘Watermelon’]
```

```
for i in range(3):
    print('my fruit is:', fruits[i])
```

The output of this code is shown below.

my fruit is: Apple
my fruit is: Orange
my fruit is: Watermelon

In Chapter 7, Example 20 demonstrates a "for" control statement. Example 6 demonstrates a "while" control statement.

3.19 Indented Code (a Suite)

In the next figure, there is a red box around the code from line 28 to 48. I added a red vertical dotted line to highlight where the code is indented. This is a Suite of code.

Let's look at the code on lines 29, 30, 31, 32, and 48. These lines are all indented to the same vertical level. This code Suite, or block of code, begins on line 28. The last line in this code Suite is line 48.

Indentation in Python scripts defines a "Suite" or code block.

In this example, the shaded Suite (block of code) is a second "while loop" (lines 34 to 46.) This second Suite is "nested" because it is inside the first Suite. Within the nested Suite, line 38 only runs when

the **if statement** on line 37 evaluates to "**true**." A nested if statement means there is a second "if statement" within the first if statement.

In this example, the "**bfr**" counter on line 48 is the last line in this Suite and, in effect, moves forward in the loop to the next item.

Project1.1.py*

```
22
23 bfrmaxrow = 6 #total rows with data in bfrfile
24 bfr = 2
25 dif = 2
26 aftmaxrow = 6 #total rows with data in aftfile
27
28 while bfr <= bfrmaxrow:
29     bfritem = ws1.cell(row = bfr, column = 2)
30     aft = 2
31     itemretired = 1
32     while aft <= aftmaxrow:
33         #%%
34         aftitem = ws2.cell(row = aft, column = 2)
35         if bfritem.value == aftitem.value:
36             itemretired = 0
37             if ws1.cell(bfr, 3).value == ws2.cell(aft,  3 ).value: #cells match
38                 aft = aftmaxrow + 1
39             else:
40                 ws4.cell(row = dif, column = 1).value = bfritem.value
41                 ws4.cell(row = dif, column = 2).value = ws1.cell(row = bfr, column = 3).value
42                 ws4.cell(row = dif, column = 3).value = ws2.cell(row = aft, column = 3).value
43                 dif = dif + 1
44                 aft = aftmaxrow + 1
45         else:
46             aft = aft + 1 #move to next row in aftfile
47             #%%
48     bfr = bfr + 1    #move to next row in bfrfile
49
```

Figure 3.2 An Indented "Suite" or Block of Code

In Python, an empty Suite (indented block of code) is illegal. For example, an "if statement" that does nothing is illegal. Instead, use the "pass" function when your code should take no action, as shown in Chapter 7, Example 7. These Chapter 7 examples illustrate a few List errors:

Example 1
Example 2

The Outline pane is a great way to see nested control statements. In Spyder, select "Outline" from the View, Panes menu.

3.20 Functions and Methods

Functions are a sequence of statements. When you define a function within a class, it is called a **method**. First, you define a function. Once a function is defined, you can use it as many times as you like by calling the function.

```
1  def menu(meal, special=False):
2
```

Depending on whether you are defining or calling a function, you call the items in parentheses either "parameters" or "arguments." When defining a function, the items in parenthesis are **parameters**. When calling a function, the items are **arguments**.

Defining a Function

When defining a function in the **Editor**, the parameters in parentheses specify what types of arguments (values) the function can accept. In my earlier example, the parameter "special" assigned a default boolean value of "False." The "special" parameter has a type of "boolean."


```
1  def menu(meal, special=False):
2       msg = ""
3       if special is True:
4            msg = 'The specials today are Mimosas. '
5       if meal == 'breakfast':
6            msg = 'Breakfast is eggs and toast. '
7       else:
8            msg = 'Sorry, we ran out of food. '
9       return msg
```

The parameter "meal" has no default value. Because of Python's dynamic typing, when calling the function, I can pass any type of data for "meal."

The Python style guide recommends function names begin with a lowercase letter. Class names should begin with an uppercase letter.

The Chapter 6 topic, "The Function Call Signature," explains how to use the **signature()**function to see what parameters another function expects, what that function does, and the function's return object. In Chapter 7, Example 20 uses the **signature()** function to retrieve parameter information.

Parameters

In this example, there are two parameters in the function definition, "meal" and "special."

	Parameter Name	Default Value	Required/Optional
Positional	meal		Required
Keyword	special	`False`	Optional

Table 2.11 Parameters

Arguments

Arguments are the values you pass to a function when calling the function. Not all functions have arguments. In this example, the function has two arguments.

```
menu('breakfast', special=True):
```

Order	Parameter Name	Argument Value
1	meal	breakfast
2	special	True

When I call the this function, the parameters are now referred to as "arguments" because I am calling the function.

Keyword Arguments

The second parameter in my function definition includes the **keyword** "special" with a default value of "False." When calling a function, a name precedes the keyword argument. List keyword arguments at the end of the parenthesized list, after positional arguments. In this example, the keyword name is "special."

Positional Arguments

The "meal" argument is a positional argument because it does not have a keyword. List positional arguments before any keyword arguments.

The example below is **invalid** because a positional argument is after a keyword argument.

```
def menu(special=False, meal):
```


Optional Arguments

Because "special" has a default value, when calling the function, "special" is an **optional** argument. If you don't provide the argument when calling the function, Python uses the default value "False."

An Arbitrary Number of Arguments

Occasionally, you may need to set an arbitrary number of arguments for a function. Let's say you have a function to print personalized movie tickets for each patron. The patron names vary from day-to-day. In the next code example, on line 1, there are two **parameters** enclosed in parenthesis:

```
1  def print_tickets(number_of_tickets, *name):
2      i = 0
3      while i < number_of_tickets:
4          print(name[i])
5          i += 1
6  print_tickets(3, 'Carter', 'Rachel', 'Michael')
```

Line 1 includes an asterisk * to indicate there are an arbitrary number of **name** arguments (values) passed to the function.

number_of_tickets

***name**

When I call the function on line 8, I pass it four values or arguments. Three of the arguments are names.

```
print_tickets(3, 'Carter', 'Rachel', 'Michael')
```

How to View the Function Argument Definition

To view arguments accepted by a function or method, you can use the Help() function, or inspect the function's **call signature**. For

example, to see the arguments of the meal() function, run the program to create the function. In the **Console** import the inspect library, and type the print statement shown below. The Python Interpreter returns the parameters for the menu function.

```
In [3]: from inspect import signature
In [4]: print(str(signature(menu)))
(meal, special=False)

In [5]:
```

Calling a Function or Method

When calling or invoking a function, you pass arguments with values to the function. Continuing with my example, the last line calls the function "**menu**."

```
print(menu('breakfast'))
```

```
1  def menu(meal, special=False):
2      msg = ""
3      if special is True:
4          msg = 'The specials today are Mimosas. '
5      if meal == 'breakfast':
6          msg = 'Breakfast is eggs and toast. '
7      else:
8          msg = 'Sorry, we ran out of food. '
9      return msg
10 print(menu('breakfast'))
```

The two statements below call the "menu" function and produce the same result. In the second example, I omit the optional argument. When calling the function, the parameter "meal" is referrred to as an argument that has a value of "**breakfast**."

```
menu('breakfast', special=False)
menu('breakfast')
```


In Chapter 7, Example 6 illustrates an AttributeError caused by missing keyword names when calling a method.

Function Return Object

A function returns one object, however that object might be a container like a tuple with several items. We looked at a return tuple object earlier in the topic "tuple." When a function doesn't specify a return value, it returns the special value "NoneType" discussed earlier. For debugging purposes, let's look at the function return object in terms of:

- What **type** of return object does the function return?
- Does the function return a value of "**None**?"
- Does the function return a tuple with several items?

This function returns a tuple with 3 strings and an int. When I call the function, I pass the return tuple elements to my variables "mystr1", "mystr2", "myint", and "mystr3".

```
def myfunction():
    print('hi')
    return 'str1', 'str2', 5, 'str3'


mystr1, mystr2, myint, mystr3 = myfunction()
```

Once I run the code and the tuple is created, I can use "type" to see what type of object the tuple returns.

```
In [3]: type(myfunction())

Out [3]: tuple
```

Boolean Return Object

The next two functions both return "True". Because the second function "**myfunction2**" is simpler, it is considered more "Pythonic.".

```
def myfunction():
    if 2 == 2:
        return True

def myfunction2():
    return 2 == 2

print(myfunction())
print(myfunction2())
```

All Paths Do Not Have a Return Value

Previously, the "menu" function returned a value on line 9. A return value **must exist for all paths** through the function. In the next example, I modified the program to have different return values for several paths. The "if" suites of code on lines 5 and 8 both have return values.

```
1  def menu(meal, special=False):
2      msg = ""
3      if special is True:
4          msg = 'The specials today are Mimosas. '
5          return msg
6      if meal == 'breakfast':
7          msg = 'Breakfast is eggs and toast. '
8          return msg
9      else:
10         msg = 'Sorry, we ran out of food. '
11 print(menu('lunch'))
```

Do you see the problem with my code? The "else" suite of code beginning on line 9 does not have a return value. When there is no return value, the Python Interpreter returns the value "**None**," which may not be what you wanted. The topic "Does the Object have a Value of None"

in Chapter 6 explains the pitfalls of the value "**None**." Example 17 in Chapter 7 illustrates how to identify a

More than One Return Element

Occasionally, functions return more than one value. For example, a function may return a tuple or list. In the next example, I pass the function's return values to **mytxt**. Because the function returns two values on line 10 in a tuple, the Console displays an error.

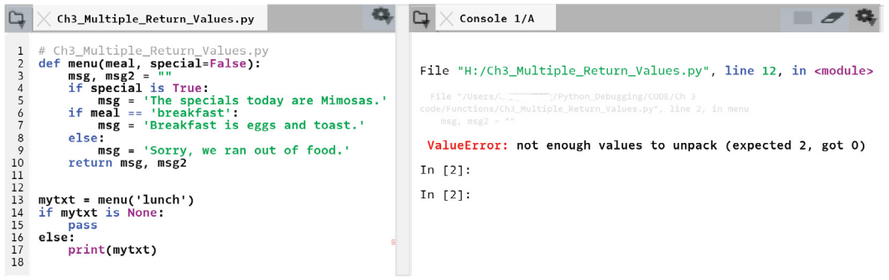


Figure 3.3 Multiple Return Values

To fix my program, I need to change line 17 to print each element in the "**mytxt**" tuple, as shown below.

```
print(mytxt[0], mytxt[1])
```

The Type of Return Value

The type of return value is especially important if you're using it as the argument for another function. For example, the "print" function expects a string, and my "**menu**" function returns a tuple. In this case, the Python Interpreter displays a TypeError. Using the previous function as an example, in the **Console** I could use the function "type" to identify the type of object the "**menu**" function returns.

```
In [1]: type(menu('lunch'))
Out [1]: tuple
```

To resolve this error, I use index notation to reference a particular item in the "**mytxt**" tuple returned by the "menu" function.

```
print(mytxt[0], mytxt[1])
```

3.21 Classes

This topic provides a brief overview of classes. The **docs.ptyhon.org** website has a tutorial on classes and explains the concept of "self" in great detail.

- Create a Class
- The DocString
- Variables - Attributes
- Create an Instance of the Class
- Methods
- Dotted Notation for Attributes
- Calling a Method

```
1  class Car():
2      """This class represents a car."""
3      yr = 2020
4  def __init__(self, model, make, year):
5      """Initialize model, make, and year variables."""
6      self.model = model
7      self.make = make
8      self.year = year
9  def drive(self):
10     """Move the car."""
11     print(self.model.title() + " is now moving.")
12 def parallelpark(self):
13     """Parallel park the car."""
14     print(self.model.title() + " is now parking.")
15 my_car = Car('Subaru', 'Crosstrek', 2019)
16 print(my_car.model, my_car.make, my_car.year)
17 my_car.parallelpark()
```


Create a Class

In this class example, the first line creates a class named "Car." Class names begin with a capital letter, to differentiate them from function names which should be lowercase.

The DocString

Lines 2, 5, 10, and 13 look like comments, but are actually examples of a "DocString." The Chapter 6 topic, "The Function Call Signature," explains how to work with the "inspect" module and view a DocString signature.

The function help() reads the docstring when gathering information about an object.

Variables - Attributes

Beginning with the function definition on line 4, you can see the four parameters in the class.

```
def __init__(self, model, make, year):
```

self
model
make
year

When working with the "my_car" instance of the Car class, use dotted notation to reference the variables.

my_car.model

my_car.make

my_car.year

When referring to the state of an object, you are referring to variables or **data attributes**. The variables **model**, **make**, and **year** on lines 6, 7, and 8, respectively, are accessible through instances.

```
self.model = model
self.make = make
self.year = year
```

The statement below is invalid because there is no attribute named "**color.**" When I run this program, the Python Interpreter displays an **AttributeError** in the **Console**.

my_car.color

In Chapter 7, Example 6 illustrates an error with an incorrect call for a class method, which causes an AttributeError.

Instance Variables and Class Variables

Instance variables are unique to each instance of the class. For example, **my_car.model** is different than **my_car2.model**. All instances of a Class share class variables and methods. For example, all instances of the **Car** class share the class variable "**yr**" I created on line 3.

Create an Instance of the Class

Instantiation is when you create an instance of an object from a class. On line 15, I create an instance of the Car class named "**my_car**."

```
my_car = Car('Subaru', 'Crosstrek', 2019)
```

Instance objects have attribute references. Valid attribute names include "**data attributes**" and "**methods**."

Methods

A function that is part of a class is called a "**method**." When referring to the behavior of an object, you are discussing the function or

method. The "Car" class has two methods, defined in lines 11 and 15. The "drive" method is shown below.

```
def drive(self):
    """Move the car."""
    print(self.model.title() + " is now moving.")
```

Dotted Notation for Attributes

The normal dotted notation "**object.variable**" is used to access the instance of the class (the object) and the attribute. In this example, the syntax is "**my_car.model**." The primary object instance is "**my_car**," and the attribute identifier name is "**model**." To refer to the model, make, or year attributes follow the syntax on line 21, as shown below.

```
print(my_car.model, my_car.make, my_car.year)
```

Variables in Imported Modules

Often modules are broken into separate files, and you add these to your code with the import statement.To reference a variable inside another module, use dotted notation. In this example, I import a module "mymodule2" that has the variable "mystr2." The expression **module2.mystr2** returns the value of **mystr2**.

import mymodule2

print(mymodule2.mystr2)

Calling a Method

To call a method in a class instance, use the syntax shown in line 17.

```
my_car.parallelpark()
```

At the end of this book, the **Appendix - Reference** has links for more information on Classes, Functions, Methods, Attributes, and Instances.

These two statements call a method. The statement syntax varies, but the statements do the same thing.

```
Car.drive(my_car)

my_car.drive()
```

3.22 Attributes

The Chapter 4 topic, "Variables and Objects in Memory," discusses the current namespace and the concept of attributes. The Python glossary entry for "attributes" is "a value associated with an object which is referenced by name using dotted expressions."

In the Chapter 2 example, we looked at a line of code with a number variable.

```
myint = 57
print(myint.upper)
```

When the program runs, it causes an unhandled exception, and the **Console** Traceback message is "**AttributeError**," because there is no attribute "upper" for a variable of type "int."

When looking at Classes in this chapter, we saw that attributes could be variables or methods within a class instance. In our earlier Class example, we saw that instance objects have attribute references. Valid attribute names include "**data attributes**" and "**methods**." In this example of attributes, "**yr**" is a class variable, and "**drive**" is a method in the **my_car** instance of the "**Car"** Class.


```
my_car.yr

my_car.drive
```

3.23 Scope, Namespace & Memory

When you step through your code, the "local scope" reflects the objects in memory at that point in time. Scope changes when your code moves into a method or function, and a new "scope" is created while you're inside that function. I

It's possible to have two variables with the same name and different values, because there are two different "scopes." You'll notice this behavior as you step though code and watch the list of variables in Variable Explorer. When you step into a function, the variable names will probably change. I say probably, because you may be using "global variables" that are available to all "scopes."

Earlier we looked at "global variables" with an example of how objects change as "scope" changes. In Chapter 4, we'll also look at "Variables and Objects in Memory."

4. Debugging Tools

In this chapter we discuss

This chapter outlines a few ways to use the Spyder IDE to debug your program. With a few simple commands, there is a wealth of information available about your variables, functions, data structures, and more. We'll look at:

- Adding Print Statements to code in the Editor, and viewing the results in the Console (the Python Shell.)
- Using Debug Mode in Spyder.
- Using Interactive Mode in the **Console**.

4.1 Debugging Overview

When debugging code, I inspect values, types, function arguments, and function return objects. "Introspection" functions like help() and dir() also provide information on methods, functions, and objects. There are also libraries for "logging" and functions to identify bottlenecks and timing issues. Finally, I'll demonstrate how to focus on a specific area of code for testing and how to create test data.

To begin, I'll look at several ways to debug my code.

- Inspect Objects and Variables
- Add Print Statements to a Script
- Debug Mode
- Variable Explorer
- Interactive Mode

When you're working in the Editor, if you place your cursor over an object name and pause a few seconds, the Editor highlights all instances of that variable or object name. This is a quick way to spot inconsistencies with variable names.

In the Editor, add **print statements** to your script, and run the program. The **Console**, also known as the Python Shell, displays the results of print statements.

> In the Run menu select  "Run selection or current line" to run only the selected lines of code.

Run your program in Debug mode, stepping through the lines of code. The **Console** prompt changes to ipdb in Debug mode. In Debug Mode, you step through the lines of code, pausing to look at the **Variable Explorer** pane or type commands at the **Console** prompt, to inspect object values. The Outline pane is a great way to see nested control statements. In Spyder, select "Outline" from the View, Panes menu.

The Editor pane highlights the current line. This line executes when you click "run current line" on the debug toolbar. The Variable Explorer displays the values of each variable in the current context.

Interactive Mode in the Console allows you to type commands that display object values. In the Console type the name, or identifier, of the object. The Python Interpreter returns the value of the identifier. The object can be an integer variable, a list item, tuple, or another type of object.

In the **Console**, you can also run individual lines of code while developing or testing your code. Interactive Mode is a great way to test code before adding it to your script. You can also set a "breakpoint" to move to a particular location in your program.

> The topic "Variables and Objects in Memory" outlines how Python creates variables and objects when you run a program or script. If you type your object name in the Console and the Traceback says "NameError," insure that you ran the line of code that creates the object.

4.2 Add Print Statements to Your Script

A popular debugging choice is to add print statements to your code. A print statement is a quick and easy way to inspect an object's type, value, and length, while your code is running. Add a print statement to your script in the **Editor** window, and on the **Run** menu execute your program. The **Console** displays output from the print statement. The Console is the Python Shell.

A quick and easy way to debug a program is to add print statements to your code. In the next example, I added two print statements to help me follow my running code.

```
1  meals = ['breakfast', 'lunch', 'snack', 'dinner']
2  fruits = ['apple', 'orange', 'grape']
3  i = 0
4
5  while i < 4:
6        j = 0
7        print("my meal is: ",   meals[i])
8        while j < 4:
9            print("My choice of fruit is: ", fruits[j])
10           print ("j is:", j)
11           j = j + 1
12       i = i + 1
```

```
In [1]: my meal is: breakfast
my meal is: breakfast
my meal is: breakfast
my meal is: breakfast
my meal is: breakfast
my meal is: breakfast
my meal is: breakfast
```

Try This 6.1 Print Statements

While marginally helpful, the print statements in the Console window quickly scroll by because this program is in an infinite loop. Scrolling output is where Debug Mode or Logging comes into play, and we'll look at both in the next sections.

In Chapter 7, Example 15 uses print statements with exception handling logic. Chapter 6 explores the syntax to view an object's type, length, and value.

Indenting Loop Print Statements

Another print option is "indenting" the print statements each time the program loops through a Suite of code. In Python, a "Suite" of code is a block of indented code, as discussed in Chapter 3. These print statements provide a visual representation of how many times the loop has run.

```
1    meals = ['breakfast', 'lunch', 'snack', 'dinner']
2    fruits = ['apple', 'orange', 'grape']
3    i = 0
4    level = ""
5    while i < 4:
6        level = level + "...."
7        print("my meal is: ", fmeals[i])
8        i = i + 1
```

In this example, the **Console** shows the print output with a series of dots representing the depth of the loops. I use the "level" string variable to create the effect.

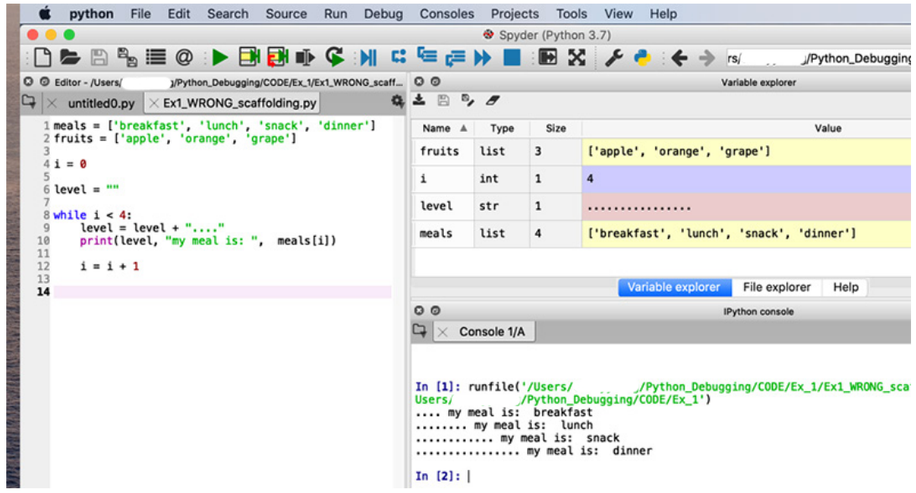


Figure 4.1 Indenting Loop Print Statements

4.3 Overview of the Editor

The Editor automatically suggests relevant code completion, based on the particular object you are working with. For example, if you create and assign a date to the variable "thedate," the Editor will display a pop-up window after you type "thedate" followed by a dot. In the next figure, I am scrolling through the pop-up items to select "year." You can turn off code completion in Preferences, Editor, Code Introspection/ Analysis.

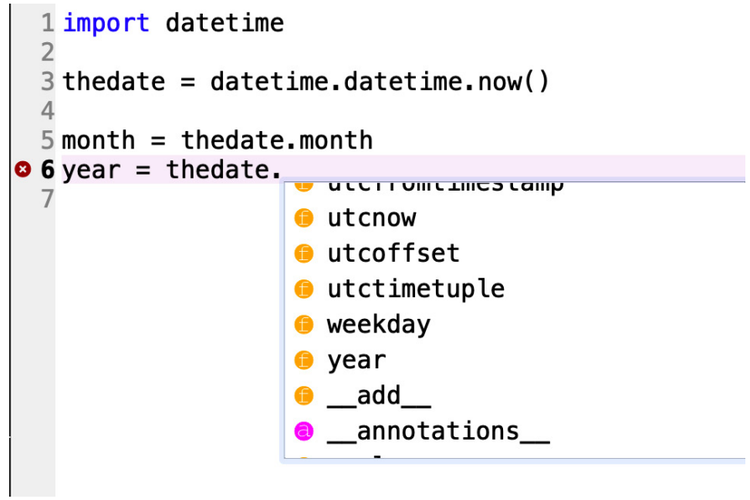


Figure 4.2 Code Completion

In this case, an orange icon indicates "year" is a function or descriptor of the datetime.datetime object "thedate", and is referenced using dotted notation, as shown below.

print(thedate.year)

If you want more information on what "year" does, you can use introspection functions, which we'll look at in the topic "Introspection" that follows. In the **Console** pane, type **dir(thedate)** to see attributes for "thedate" object.

dir(thedate)

In the Console pane, when you type **help(thedate)** the object attributes are displayed, as shown below.

help(thedate)

```
1 import datetime
2
3 date1 = datetime.datetime.now()
4 date2 = datetime.datetime(1976, 3, 25)
5 date3 = datetime.datetime(1989, 6, 30)
6
7 thedates = [date1, date2, date3]
8
9
10 maxdate = max(thedates)
11 print('the oldest date in the list is', maxdate)
12
13
14 datetime.datetime.
15
```

```
fromtimestamp
hour
isocalendar
isoformat
isoweekday
max
microsecond
min
```

Figure 4.3 The max() Function

The purple icon in the pop-up window indicates attributes, while the orange icon is a function (or method.) The **max(thedates)** function in the example above displays the oldest date in the list "thedates."


```
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  fold
|
|  hour
|
|  microsecond
|
|  minute
|
|  second
|
|  tzinfo
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  max = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)
|
|  min = datetime.datetime(1, 1, 1, 0, 0)
|
|  resolution = datetime.timedelta(microseconds=1)
|
```

Figure 4.4 Console Display of Object Attributes

If you type "datetime**.**" in the Editor, the code completion pop-up includes "date" with a blue icon indicating this is a class constructor. This datetime.date() method takes 3 integer arguments: year, month, day.

dob = datetime.date(1972, 12, 3)

> A function created in a class is called a method.

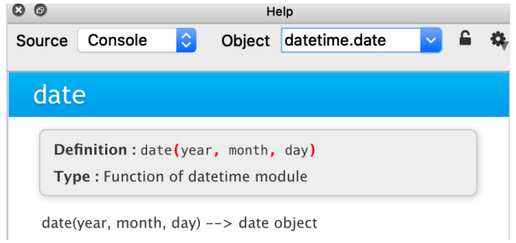


Figure 4.5 Help for datetime.date Function

4.4 Debug Mode

Use the **Debug** menu commands to step through the lines of code, or press Cntrl + F12 on a Windows computer to move to the next breakpoint. The Variable Explorer displays object values, changing over time as you step through the program code.

In the **Editor**, double click on a line of code to set a **breakpoint** or press F12 on a Windows computer. When running a program in Debug Mode, a breakpoint pauses the program at that point, so that you can inspect the variable and object values. A red dot appears to the left of the line number with the breakpoint.

On the **Debug** menu, select 'debug' to launch the iPython debugger, or press Cntrl + F5 on a Windows computer. The prompt in the **Console** changes to **ipdb>**, indicating the iPython debugger is active.

> If a program halts and displays a Traceback error, you can type **%debug** to start "Debug Mode." Functions that begin with the percent symbol are "magic functions."

In the next example, there is a breakpoint • on line 3. The figure shows the **Editor** pane, as well as the **Console** pane after I pressed Cntrl + F5 on a Windows computer to start debugging. Notice the **Console** prompt changed to **ipdb>**.

```
1 mystring = "purple peanuts"
2
3 print (mystring)
```

```
In [1]: debugfile('C:/SampleScript.py',
wdir='C:')
>C:\SampleScript.py(1)<module>()
---->1 mystring = "purple peanuts"
      2
      3 print (mystring)

ipdb>
ipdb>
```

Table 4.1 Setting a Breakpoint

In the **Console** pane, an arrow indicates the current line number, in this case, line 1. If Variable Explorer is not already open, on the **View** menu select "Panes," and then click on Variable Explorer. As I "**step-through**" the code, I want to watch the "mystring" object in Variable Explorer. At this point, the Variable Explorer is empty because we have yet to run the first line of code.

As you step through the code, the Editor highlights the current line.

Click the icon to **Run the current line of code** or press **Cntrl + F10** on a Windows computer. The Python interpreter creates the object "mystring" and assigns the value "purple peanuts." This example of dynamic typing is one of the reasons I love Python. With one line of code, Python figures out the type of object to create and assigns a value.

In Chapter 7, Example 2 illustrates Debug Mode.

End Debug Mode

To exit the debugger, type **q** or **quit** at the **Console** prompt and press enter, as shown below. If you are in an iPython Session in the **Console**, you may have to press enter several times to quit Debug Mode, or type **Esc + Enter**. You can also restart the kernel when you select "Restart Kernel" from the **Consoles** menu.

```
ipdb>C:\SampleScript.py(1)<module>()
     1 mystring = "purple peanuts"
     2
---->3 print (mystring)

ipdb>
ipdb>quit

In [2]:
```

Table 4.2 Quit Debug Mode

4.5 Variable Explorer

Now, **Variable Explorer** displays a row with the type of the "mystring" object, and the value I assigned in line 1.

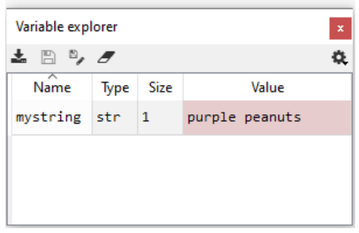


Figure 4.6 The Variable Explorer Pane

Variable Explorer shows variables and objects in memory. If you don't see your object displayed in Variable Explorer, you need to execute that part of the program. To start fresh, in the Console use the magic command **%Reset**. See the earlier topic "Variables and Objects in Memory." In Chapter 7, Examples 1-3 demonstrate using the Variable Explorer.

4.6 Example: My Program Loops & Never Ends

Let's briefly look at an example of **Debug Mode** in action. When I run this test program, it never ends. In other words, it loops continuously. My hypothesis is the while loop that begins on **line 8** needs adjusted. First, I'll stop the running program. Next, I'll use Debug Mode to step through the code and identify what is happening.

1. On the **Consoles** menu, select "Restart Kernel" to interrupt the running program.

2. Double click to the left of line 9 to add a breakpoint. A **red** dot appears to the left of the line number.

3. On the **Debug** menu, select "Debug" or click on the Debug control.

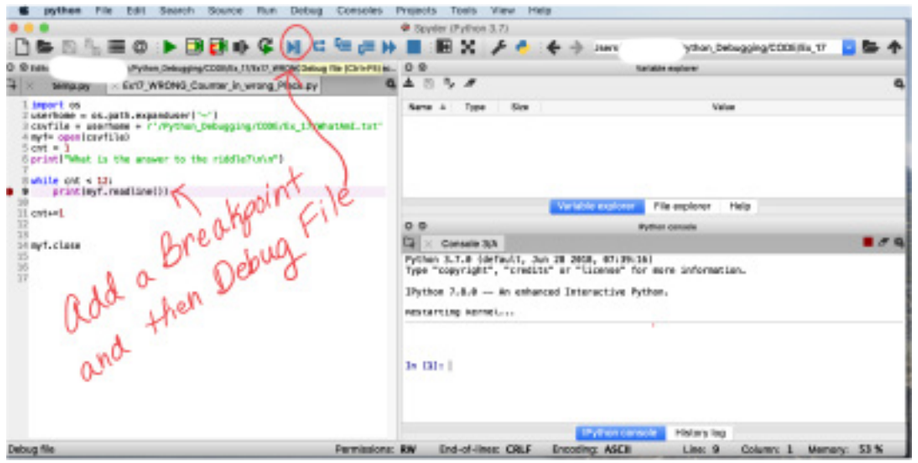


Figure 4.7 Add a Breakpoint

4. The **Console** prompt changes to **ipdb>** to indicate the iPython Debugger is active, and the **Variable explorer** displays values for the current active variables. Right now, the variable **cnt** has a value of **1**.

The Console displays a few lines of the code, with an arrow indicating the current line 9. Line 9 runs when I click on "Continue Execution."

> The Debug Mode commands "u" or "up" move backward in your program. These commands are useful to find where the value was assigned to a variable. The Console prompt changes to ipdb> when in Debug Mode.

5. When I click on "Continue Execution" again, line 9 runs and loops back to line 8. The variable **cnt** still has a value of **1**. At this point in the program, I wanted the value of **cnt** to be 2.

   If I use the command "Run Current Line," in the Console pane, I can see the program moving continuously from line 8 to 9 and then looping back to 8. This program is in an infinite loop.

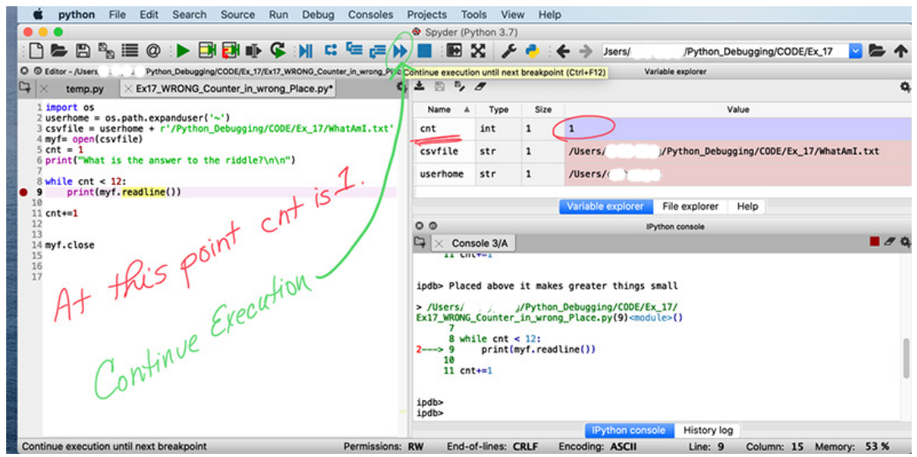


Figure 4.8 Continue Execution

6. To resolve the issue, I need to indent line 11, so this statement that increments the **cnt** variable is part of the while loop. The concept of a "Suite" of Indented Code was discussed in Chapter 3.

In the next figure, the output is correct in the **Console**. In the Editor pane, you can see line 11 is now indented.

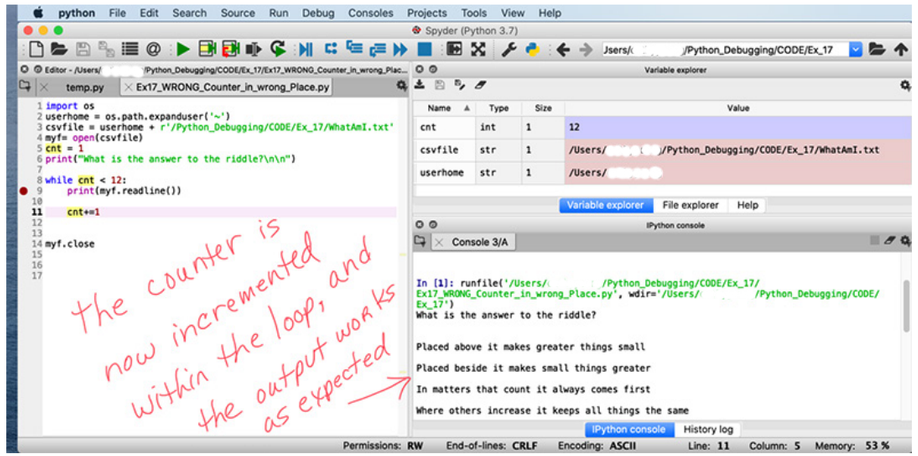


Figure 4.9 The Finished Program

To complete the program, I could add a print statement with the answer to the riddle - the number one.

In Chapter 7, Example 1 demonstrates an infinite loop.

4.7 Debug Commands

In Debug Mode, type ? in the **Console** prompt **ipdb>** and press enter to see a list of Debug Commands. A brief list of popular commands is shown below.

ipdb> ?

For specific details on a particular command, type help, and the command name. For example, type "**help next**."

?	Help with Debug commands
b or **break**	Add a break
c	Continue
cl or **clear**	Clear breaks
d or **down**	Move down in the stack trace
exit	Exit Debug Mode
h or **help**	Help on Debug Mode
j or **jump**	Jumps to line number with a block of code
n or **next**	Move to next line
u or **up**	Move up in the stack trace
q or **quit**	Exit Debug Mode

4.8 Console Interactive Mode

Another option to view object values involves typing in the **Console** in **Interactive Mode**, which is similar to typing in the **Console** while in Debug Mode. In the **Console**, type the identifier (the name) of the object. The Python interpreter displays the value in the **Console**, as shown below.

```
In [1]: mystring

Out[1]: 'purple peanuts'
```

You must run the program statement that creates or sets the object value in the current namespace or local scope before the Python Interpreter, or Variable Explorer, can display a value.

At any time, you can type the name of an object in the **Console**, and the Python Interpreter displays the value. This "Interactive Mode" also allows you to perform calculations or use functions and methods, as shown in the next example.

```
ipdb> mystring
'purple peanuts'

ipdb> 2+3
5

ipdb> import math

ipdb> math.sqrt(16)
4.0

ipdb>
```

Table 4.3 Type in Console

The iPython kernel also has several "magic commands" that begin with the % character.

%debug

%reset

> Click in the Console window and press **Ctrl C** to cancel program execution. **Ctrl L** clears the namespace memory.

In Chapter 7, Example 1, Example 2, and Example 6 demonstrate Interactive Mode.

Increment Counters in the Console

One of my **favorite debugging shortcuts** is to change a counter so that I can move forward when I'm looping through the code. For example, I can change a "while loop" to move from the 2nd iteration to the 1200th iteration. Let's say I'm in debug mode, and my program pauses at a breakpoint where my "bfr" counter = 2. In the Console, I would increment the "bfr" counter by typing **bfr =1200**.

Watch Out for Changing Values

While most functions or methods provide useful results when typed in the Console, you can get unexpected results. In Chapter 7, Example 17 reads a TXT file with the OS library. **Readline()** moves to the next line of the TXT file on its own every time you type it in the Console, which may not be what you were expecting.

iPython Session

The **Console** prompt changes to three dots and a colon **...:** to indicate you are in an **iPython Session**. Press enter twice, or press **Esc + Enter**, to exit the iPython session.

```
In [1]: def myfunction(str)
   ...: print(str)
```

> When the newline prompt ...: is displayed, press Shift Enter to execute the commands.

4.9 Introspection

Introspection is the ability to determine the type of the object at runtime. Several functions help with introspection, as well as the "**inspect**" library.

objectname?
dir()
help()
id()
repr()
type()

To inspect objects, we'll execute statements in the **Editor** and **Console**, including statements with "Instrospection" functions that provide details about objects. The syntax varies depending on whether you are typing in the **Editor** or **Console** pane. The syntax is also specific to the type of object. We'll look at those differences in depth in Chapter 6. In the case of data structures like Lists, Tuples, or Dictionaries, you may want to see values for the entire List or the value of only a particular List item.


Editor


Console (Python Shell)

4.10 Variables and Objects in Memory

The Python Interpreter creates variables and objects when you run a program or script. The collection of these objects in the **Console** is the "**Namespace**." If you're debugging a line of code and a function uses an object, you want to ensure the object exists in memory before trying to view the object in **Variable Explorer**. The Variable Explorer shows active variables in memory. Take, for example, this line of code that uses a variable "**myint**" of type "int."

```
myint = 57
print(myint.upper)
```

If the program ran and created "myint" already, the **Console** Traceback message is "**AttributeError**," because there is no attribute "upper" for a variable of type "int." **If the program hasn't run and created the variable "myint,"** the **Console** Traceback message is "**NameError.**" In this example, a misleading Traceback message "NameError" is hiding the Traceback message you want to see, "AttributeError."

In Chapter 3, we looked at an example of a "global variable," and how object values change as scope changes.

When you change a function definition and want to use the new version of the function, you can run just that part of your code. In the Run menu select "Run selection or current line" to run only the selected lines of code.

> Use the %reset magic command in the **Console** to reset the namespace.

Using ? in the Console

For details on any object, in the **Console** type the object name followed by a question mark. For details on the object "**myfunction**,"

in the **Console** type the function name followed by a question mark, as shown below. The output includes the Signature, DocString, and Type of object.

```
In [2]: myfunction?
Signature: myfunction(str)
Docstring: <no docstring>
File:      ~/Python_Debugging/CODE/Ch 3 code/Functions/<ipython-input-8-df3069fd62ae>
Type:      function
```

dir()

The function **dir()** displays all objects in the current local namespace, as shown in the next figure. After running the sample "*Project1.1.py*" script, the local scope changes. This script uses the "openpyxl" library to create the **ws4** object. For example, after running the "*Project1.1.py* script, the `dir()` function displays relevant information about the program objects in the **Console**. Type the `dir()` command in the **Console** window.

```
In [2]: dir()
```


Figure 2.5 Objects in Current Local Scope

dir(object)

While dir() looks at all objects, the statement **dir(ws4)** takes the argument "ws4" and retrieves information on that particular object. In the **Console** window attributes specific to the **ws4** object are displayed, as shown below. The `dir(object)` function displays different attributes depending on the type of object you use for the argument.

There is quite a long list of valid attributes for the **ws4** object, and the next example only shows a few of the attributes. In particular, I'm interested in what functions I can use with the **ws4** object, and I've highlighted the "**delete_rows**" method.

Note, if you're using an older version of Python, "**delete_rows**" might not be available. The **dir()** function is an easy way to check if a particular function or method should work with your code.

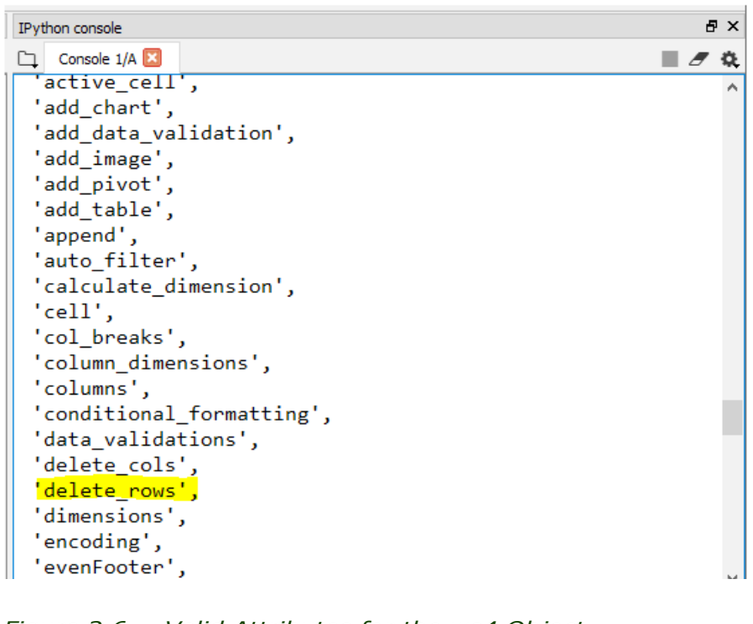


Figure 2.6 Valid Attributes for the ws4 Object

help()

The Help() function invokes the help system for help with a module, function, class, method, keyword,for objects in the current namespace. For example, when I type help(load_workbook) in the **Console** window, Python displays information specific to the method "load_workbook" from the "openpyxl" library.

```
In [2]: help(load_workbook)
```

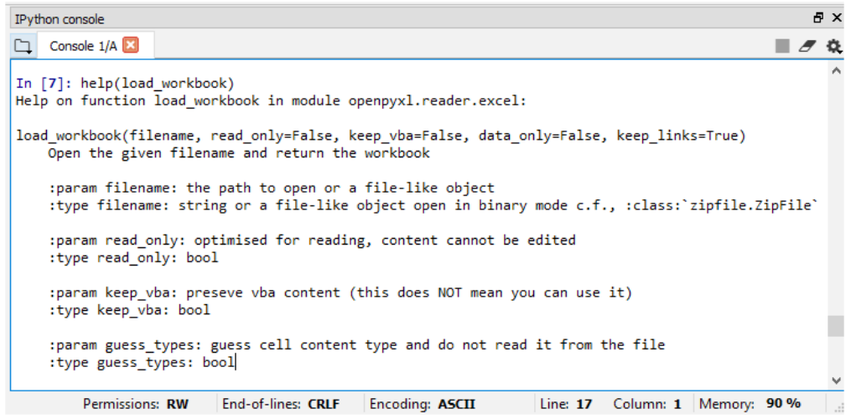



Figure 2.7 Help for load_workbook Method

The help() function reads the **docstring** and inspects objects to gather the output.

The Inspect Library

Use the "Inspect" library for additional information on an object, including the Docstring or call signature of a function or method. There are many functions available in the library. Details are available on the docs.python.org website.

```
In [3]: from inspect import signature
In [4]: print(str(signature(load_workbook)))
(filename, read_only=False, keep_vba=False, data_only=False, keep_
links=True)
```

The type() Function

In the Chapter 3 topic, "What is the Data Type?," we used the **type()** function to examine the type of an object. Chapter 6 also includes numerous examples using the type() function.

```
print(type(my_var))
```

The id() Function

When dealing with immutable objects or scope issues, the **id()** function is useful in isolating which object you are referencing. The id() function displays the identity of the object. Scope has to do with global vs. local variables.

```
print(id(bfr))
bfr = bfr + 1
print(id(bfr))
```

The repr() Function

The **repr()** function returns a string representation of an object, and is useful in finding "whitespace," special characters like new line **\n**, or float rounding errors.

The len() Function

The len() function shows the length of the string or the number of items in a data structure. For example, len(mydictionary) would return the number of Dictionary key pairs. len(mylist) would return the number of items in the List.

4.11 Logging

Logging is a simple way to capture debugging data. Use logging when the output in the **Console** pane scrolls and is lost because there is too much data. Logging is also useful when you're working out code logic, or have a live program with user reports of erratic behavior.

This script has a logging level set to ERROR, which means it logs errors and critical events. For a thorough look at logging, please refer to the docs.python.org.

```
logging.basicConfig(format='%(asctime)s - %(message)s',
         datefmt='%d-%b-%y %H:%M:%S',
         filename='test.log',
         level=logging.ERROR)
```

When there is an exception on line 9 in the statement **10/my_int**, the Python Interpreter logs a critical error to the **test.log** file.

```
1  import logging
2  logging.basicConfig(format='%(asctime)s - %(message)s',
3           datefmt='%d-%b-%y %H:%M:%S',
4           filename='test.log',
5           level=logging.ERROR)
6  logging.error('The logging level is ERROR and above.')
7  my_int = 0
8  try:
9       10/my_int
10 except Exception:
11      logging.critical("my_int is %s", my_int, exc_info=True)
```

The first time you run the program the logfile **test.log** is created. The default mode is "append." If the log file is not created, try restarting Spyder. This is the output in the log file.

```
29-Jan-19 10:29:33 - Logging level is ERROR and above.
29-Jan-19 10:29:33 - my_int value is 0
Traceback (most recent call last):
     File "/Ch 4 code/Logging/logging.py", line 13, in <module>
     10/my_int
ZeroDivisionError: division by zero
```

To disable logging, use the "disable" method with the appropriate argument, as shown below.

```
logging.disable(logging.CRITICAL)
```

To enable logging again, use the "disable" method with the "NOTSET" argument.

```
logging.disable(logging.NOTSET)
```

4.12 The timeit() Function

The timeit() function can identify bottlenecks in your code. Let's say we want to time this block of code.

```
colors = ('blue', 'red', 'green')
for color in colors:
    print(color)
```

Import the **timeit** module. Create a string "**mycode**" that encloses the code statements in triple quotes. In the example below, the last line invokes the **timeit** method to run the code 100 times.

```
1  from timeit import timeit
2  mycode = '''
3  colors = ('blue', 'red', 'green')
4  for color in colors:
5      print(color)
6  '''
7  print(timeit(stmt=mycode, number=100))
```

4.13 Logging Time and Loop Counters

When writing new code I often add print statements to display the time, as well as loop counters. I have to admit, watching a Python

program take 30 seconds to complete a quarter of a million comparisons makes me very happy I use Pythong!

This is an example showing process time, the current local time, and a counter.

```
import time
i = 1
j = 50
start = time.process_time()
start1 = time.localtime()
thestarttime = time.asctime(start1)
print('Start time is: ', thestarttime,
      'and start process time is', start, '\n')
while i < 10000:
    if i == j:
        end = time.process_time()
        print('Time so far is:', start - end)
        print("i is", i)
        j += 1000
    i += 1
end = time.process_time()
print('Time to complete is: ', start - end)
start2 = time.localtime()
theendtime = time.asctime(start2)
print("\nStarted at:", thestarttime)
print("Ended at:", theendtime
```

The Console output is shown below:

```
Start time is:  Sun Jun 14 19:25:05 2020 and start process time is 11.184358

Time so far is:  -0.0003019999999995804
i is 50
Time so far is:  -0.001435999999999928
i is 2050
Time so far is:  -0.001829999999999983
i is 3050
Time so far is:  -0.002218000000009414
i is 4050
Time so far is:  -0.00279200000001238
i is 5050
Time so far is:  -0.003218000000003872
i is 6050
Time so far is:  -0.003602000000007713
```

```
i is 7050
Time so far is:  -0.003984000000000876
i is 8050
Time so far is:  -0.00436800000000126
i is 9050
Time to complete is:  -0.005131999999999692

Started at: Sun Jun 14 19:25:05 2020
Ended at: Sun Jun 14 19:25:06 2020
```

4.14 Focused Testing

Sometimes I need to focus on one part of my code, to the exclusion of other areas. By providing test data for the steps I'm excluding, I can focus on the defect. Let's look at my program that has five tasks.

1. Get KDP royalites.
2. Get the GBP exchange rate.
3. Calculate total sales for the month.
4. Calculate the average daily sales for the month.
5. Calculate the expected monthly sales.

Actual Result

When I run the code, the program halts. The Traceback shows a ZeroDivisionError on line 65. I haven't changed anything in the program in several weeks. Until today the program ran successfully. This is the code on line 65.

```
dailysales = (total/myday)
```


Incorrect Code

The value of "**myday**" is set on line 64, as shown below. As it happens, on the first day of the month, the statement on line 64 evaluates to zero.

```
myday = (datetime.datetime.today().day-1)
dailysales = (total/myday)
```

While this particular example is easy to troubleshoot, when you have a program with external connections, it can be a challenge to isolate the defect. With a slight modification, I can use a variable "**testmode**" as a switch to use test values. When I want to test 'datecalculations,' I set all the other conditional statements to use test data. In effect, I remove all the other code from the equation and only run lines 62-68.

For example, the "else" statement on line 54 sets **gbp** to a value of 999.99 when **testmode** is "**tst_datecalculations**."

test_mode.py

```
44
45 ## Get exchange rate
46 if testmode == '' or testmode == 'tst_exchangerate':
47     url2 = 'https://usd.fxexchangerate.com'
48     html2 = urlopen(url2)
49     soup2 = BeautifulSoup(html2, 'lxml')
50     tables2 = soup2.findChildren('td')
51     gbpex = float(tables2[3].string[:6])
52     gbp = gbp/gbpex
53     print("gbp converted to USD is:", gbp)
54 else:
55     gbp = 999.99
56
57 # Calculate total sales for the month
58 total = usd + gbp
59 print("total sales this month  ", round(total, 2))
60
61
62 if testmode == '' or testmode == 'tst_datecalculations':
63     # Calculate average daily sales
64     myday = (datetime.datetime.today().day-1)
65     dailysales = (total/myday)
66     print("average daily sales ", dailysales)
67 else:
68     dailysales = 2.22
```

Figure 4.10 test_mode.py

4.15 Create Test Data

Use the smallest subset of data possible for testing. After removing a chunk of data, ensure you still have enough data for your program to function. If you're debugging an error, be careful to keep the data that recreates the error.

In this example, I changed the code on line 13 and created an HTML data file. Rather than connecting to a live website, I copied the "HTML" data to a file. I also removed unnecessary headings and tables from the HTML file.

test_data.py

```
12 # get kdp royalties
13 html_file = open('kdproyalties.html', 'r')
14 source_code = html_file.read()
15 soup = BeautifulSoup(source_code, 'html.parser')
16 tables = soup.findChildren('table')
17 mytable = tables[0]
18 rows = mytable.findChildren(['tr']
19 for row in rows:
20     currency = row.findChildren()[4].string
21     if currency == 'USD':
22         mymoney = (row.findChildren()[13].string)
23         usd = usd + float(mymoney.replace(',', ''))
24     if currency == 'GBP':
25         mymoney = (row.findChildren()[13].string)
26         gbp = gbp + float(mymoney.replace(',', ''))
27
28
29 # Get exchange rate
30 html2 = urlopen('https://usd.fxexchangerate.com')
31 soup2 = BeautifulSoup(html2, 'lxml')
32 tables2 = soup2.findChildren('td')
33 gbpex = float(tables2[3].string[:6])
34 gbp = gbp/gbpex
35
36
```

Figure 4.11 Test Data.py File

Below is a small excerpt of the html data, with the data I need for my program.

try and except.py | royalties_divide_and_conquer.py | kdproyalties.html*

```
56      </tr>
57      <tr>
58        <td valign="middle" class="td1">
59          <p class="p2"><span class="s1">Amazon.co.uk</span></p>
60        </td>
61        <td valign="middle" class="td2">
62          <p class="p2"><span class="s1">GBP</span></p>
63        </td>
64        <td valign="middle" class="td3">
65          <p class="p2"><span class="s1">0.00</span></p>
66        </td>
67        <td valign="middle" class="td4">
68          <p class="p2"><span class="s1">0.00</span></p>
69        </td>
70        <td valign="middle" class="td5">
71          <p class="p2"><span class="s1">2.43</span></p>
72        </td>
73      </tr>
74      <tr>
75        <td valign="middle" class="td1">
76          <p class="p2"><span class="s1">Amazon.de</span></p>
77        </td>
78        <td valign="middle" class="td2">
79          <p class="p2"><span class="s1">EUR</span></p>
80        </td>
81        <td valign="middle" class="td3">
82          <p class="p2"><span class="s1">0.00</span></p>
83        </td>
84        <td valign="middle" class="td4">
85          <p class="p2"><span class="s1">0.00</span></p>
86        </td>
87        <td valign="middle" class="td5">
88          <p class="p2"><span class="s1">0.00</span></p>
89        </td>
90      </tr>
91      <tr>
92        <td valign="middle" class="td1">
```

Figure 4.12 Test HTML Data File

5. Exceptions

In this chapter we discuss

Kinds of Errors

The Stack Trace or Traceback Message

Try and Except

Raise

Assert

Built-in Error Types

To begin our discussion of exceptions, we'll look at the basic kinds of programming errors. When I say "kind" of error, this is just a general classification to characterize programming errors. When an event happens that the Python Interpreter can't process successfully, it stops the program with an exception.

After an unhandled exception occurs, the Python Interpreter displays the stack trace in a "**Traceback**" message is the **Console** pane with details about the exception. In the topic, "Traceback Message," you'll see there is a wealth of information in a Traceback message. Often the Traceback message immediately points to the cause of the error.

To handle exceptions, you can add "**try and except**" statements to deal with exceptions gracefully. For critical events, we'll look at the "**raise**" command where you raise your exception. When an object must be a certain value, adding an "**assert**" statement to trigger an exception is a great time saver.

Finally, we'll briefly look at the Python built-in exceptions you're likely to encounter when programming in Python.

5.1 Kinds of Errors

Generally, when things go wrong in a program, they fall into one of three categories.

- Syntax Errors
- Logic or Semantic Errors
- Runtime Errors

Syntax Errors are usually obvious, and the Spyder Editor points out Syntax errors with a yellow triangle. Runtime errors occur when the Python Interpreter halts and displays an exception. I find my "Logic" errors the most difficult to identify because the program does what I told it to, but my initial design or logic is flawed.

Syntax

A syntax error is raised by the parser when the parser encounters a syntax error. The Spyder Editor makes it virtually impossible to have Syntax errors. A yellow triangle appears to the left of the line number if there is a Syntax error in the code.

Chapter 7 demonstrates syntax errors in Example 5, and Examples 7-12.

Logic or Semantic

With a logic error, the flaw is with me. I told the program to do something, but it's not the outcome I want. To identify logic errors, I find it helpful to go back to the drawing board and look at my initial "Intended Outcome" or pseudocode. Pseudocode is an outline of your program design in simple terms, often written in plain English.

Runtime

The challenge with debugging runtime errors is a line (or suite) of code runs as expected several times, and then suddenly halts with an error. As a program runs, variable values change. Another example of a RunTime error is when a program takes too long to run.

While general RunTime errors are flagged as a "RunTimeError" by the Python Interpreter, what I am referring to as "Runtime" is the overall kind of error. The actual Traceback message displayed in the Console may vary, as shown in Chapter 7 in Examples 5 and Examples 7-12.

To research a runtime error, we need to look at the values at the moment the error occurred When looking at values, you may have a critical variable that must be a certain value for your code to function. Assert statements halt the program and warn you when values are outside the parameters you require.

Another example of a Runtime error is a chunk of code that takes too long to run. In this case, the function "timeit" times program execution.

- Debug Mode
- Variable Explorer
- Interactive Mode
- Print Statements

Chapter 7 looks at these kinds of errors in Examples 1, 4, 6, 14, 15, and 17.

5.2 The Stack Trace or Traceback

The Traceback includes this basic information.

- File
- Line Number
- Module
- Exception
- Exception Description

The next lines are a sample **Console** Traceback. I abbreviated the file path for readability.

```
File "workbook.py", line 289, in __getitem__
    raise KeyError("Worksheet {0} does not exist.".format(key))
KeyError: 'Worksheet Sheet1 does not exist.'
```

The Traceback details are as follows:

File: "workbook.py"
Line Number: 289
Module: __getitem__
Exception: **KeyError**
Exception Description: Worksheet Sheet1 does not exist.

This Traceback information provides the clues needed to research many issues. In Chapters 6 and Chapter 7, I'll often refer back to the Exception in the Traceback.

Chapter 7, Examples 1, 2, 3, and 6 demonstrate traceback screens.

Don't Be Fooled

A misleading Traceback message "NameError" could be hiding the Traceback message you want to see. In the Chapter 4 topic, "Variables and Objects in Memory," we looked at how the Python Interpreter creates variables and objects when you run a program or script. In this example, there is a variable "myint" of type "int."


```
myint = 57
print(myint.upper)
```

If the program ran and created "myint" already, the **Console** Traceback message is "AttributeError" because there is no attribute "upper" for a variable "myint." If the program hasn't run and created the variable "myint," the **Console** Traceback message is "NameError."

When the program encounters an Out of Memory error, the Traceback exception is rarely the actual cause of the defect.

5.3 Try and Except

Unhandled exceptions halt the program and display a Traceback with an exception error. When you add "try" and "except" statements to your code, you can control how exceptions are handled, and prevent your program from unexpectedly halting. In this next example with "except," you can see where I added custom messages.

```
try:
    gbpex = float(tables2[3].string[:6])
    gbp = gbp/gbpex
    print("gbp converted to USD is:", gbp)
except TypeError:
    print("Type error when converting exchange rate")
except ZeroDivisionError:
    print('ZeroDivisionError where gbpex is:', gbpex)
except Exception as exceptdetails:
    print(exceptdetails, 'gbpex is:', gbpex)
finally:
    print("Done calculating the gbp exchange rate.")
```

The **finally** clause at the end runs whether the **try** clause has an exception or not.

5.4 Raise

At any point, you can add your own "raise" statements in your program to raise an exception, as shown below.

```
raise Exception("I broke my program.")
```

5.5 Assert

When your program depends on a statement to be true, consider adding an "assert" statement to alert you if the statement does not evaluate to "true." Some interesting assertions are:

- A number is > 0
- A variable is a particular type (datetime)
- A string is not None

When I was calculating KDP royalties earlier, I found the GBP exchange rate on a web site. In order for my program to calculate GBP royalties in USD currency, "**gbpex**" must be greater than zero.

```
assert gbpex > 0, 'gbpex must be > ' + str(gbpex)
```

Now when my program runs and "**gbpex**" is not greater than zero, an exception is raised. The **Console** displays a Traceback message, as shown below.

```
AssertionError: gbpex must be > 0
```


In this example, I check that the variable "thedate" is a datetime type. This time thedate is valid.

```
import datetime

thedate = datetime.datetime.now()

assert type(thedate) == datetime.datetime, "this isn't a date"

if thedate is not None and type(thedate) == datetime.datetime

    print('everything is ok')
```

With a small modificatione "thedate" becomes an "int." Now "thedate" is not a datetime type, and the assertion fails.

```
import datetime

thedate = (datetime.datetime.today().day-1)

assert type(thedate) == datetime.datetime, "this isn't a date"

if thedate is not None and type(thedate) == datetime.datetime:

    print('everything is ok')
```

5.6 Built-in Error Types

The following list of built-in exceptions is a reference for the examples that follow. For a complete list of exceptions, visit https://docs.python.org/3/library/exceptions.html.

ArithmeticError

ArithmeticError is the base class for built-in exceptions for various arithmetic errors.

AssertionError

An **AssertionError** is raised when the assert statement fails.

AttributeError

The **AttributeError** is raised on attribute assignment or when the reference fails. When an object does not support attribute references or attribute assignments at all, a **TypeError** is raised. For example, an "int" object has no attribute "upper." The code below would cause an **AttributeError**:

```
myint = 57
print(myint.upper)
```

Because a "string" object does have an attribute "upper," this code for a string is valid.

```
mystr = 'age'
print(mystr.upper())
```

To view attributes of an object named "**mystr**" run the program, then type "**dir(mystr)**" in the Python **Console**. You must run the program for the Python Interpreter to create the variable "mystr." **If you haven't run the program**, the Python Interpreter displays a **NameError** exception. See Chapter 7, Example 6, for a description of debugging an **AttributeError**.

Earlier in Chapter 3 we looked at strings in dictionaries.

In Chapter 7, Example 6 demonstrates an AttributeError.

In Python, Objects have attributes. So, for example, the object "**ws1**" has an attribute named ".**cell**." In this example, the dotted notation would be **ws1.cell()**.

EOFError

Raised when the input() function hits the end-of-file condition without reading any data. In Chapter 7, Example 15 demonstrates an EOFError.

FloatingPointError

Raised when a floating-point operation fails.

ImportError

When Python Interpreter has trouble loading a module, the Interpreter raises an ImportError.

Indentation Error

The IndentationError is raised when there is an incorrect indentation. In Chapter 7, Example 5 demonstrates an IndentationError.

IndexError

An IndexError is raised when the index of a sequence is out of range as discussed in Chapter 3. If the index is not an integer, TypeError

is raised. The base class of an IndexError is a LookupError. See Examples 1-2 in Chapter 7.

IOError

Starting from Python 3.3, an IOError is an OSError.

KeyError

A KeyError is raised when a key is not found in a dictionary and is a subclass of LookupError. See Example 20.

KeyboardInterrupt

A KeyboardInterrupt is raised when the user hits the interrupt key (Ctrl+c or delete).

LookupError

A LookupError is the base class for KeyError and IndexError.

MemoryError

Raised when an operation runs out of memory.

ModuleNotFoundError

The ModuleNotFound error indicates a module could not be located and is a subclass of ImportError.

NameError

Raised when a variable is not found in the local or global scope. This exception occurs when an identifier is invalid or an unknown name. For example, a misspelled identifier can cause a NameError. The Spyder IDE would highlight a NameError. In Chapter 7, Examples 3, 10, 13, and 22 demonstrate NameErrors. We also looked at the related **UnboundLocalError** in the Chapter 3 topic, "Global Variables."

> The Chapter 4 topic, "Variables and Objects in Memory," outlined how the Python Interpreter creates variables and objects when you run a program or script. If you type your object name in the Console and the Traceback says "NameError," ensure that you ran the line of code that creates the object.

OSError

Raised when a system operation causes a system-related error, such as failing to find a local file on disk.

OverflowError

Raised when the result of an arithmetic operation is too large to be represented. The OverflowError is a subclass of ArithmeticError.

RecursionError

A RecursionError is derived from the base class RuntimeError. A RecursionError is raised when the maximum recursion depth is exceeded.

RuntimeError

Raised when an error is detected that does not fall under any other category.

StopIteration

Raised by the next() function to indicate that there is no further item to be returned by the iterator.

SyntaxError

Raised by the parser when a syntax error is encountered. The Spyder Editor makes it virtually impossible to have Syntax errors. A yellow triangle appears to the left of the line number if there is a Syntax error in the code.

TabError

Raised when indentation contains an inconsistent use of tabs and spaces. The TabError is a subclass of IndentationError.

SystemError

Raised when the interpreter detects an internal error.

SystemExit

Raised by the sys.exit() function.

TypeError

Raised when a function or operation is applied to an object of an incorrect type. Attempting to perform an operation on an incorrect object type. Examples of TypeErrors you might see are:

- tuple object does not support item assignment
- string indices must be integers

Tuple Object does not Support Item Assignment

For example, if you try to assign a new value to an item in a **tuple**, a TypeError is raised.

```
In [3]: mytuple[1] = 'three'
Traceback (most recent call last):

    File "<ipython-input-3-db66c3391d15>", line 1, in <module>
      mytuple[1] = 'three'

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

String Indices Must be Integers

In Chapter 3, we saw a **TypeError** when iterating over a **string**. The "print" statement shown below would cause an error. because the values 'abc' are not integers

```
mystr = 'abc'
for i in mystr:

    print('mystr char is:', mystr[i])
```

The **Console** would display a traceback message with a "**TypeError.**" I've abbreviated the Traceback message below for readability

In [2]:

```
Traceback (most recent call last):
TypeError: string indices must be integers
```

A slight modification in the code would prevent the error. In the example below I am using the "range()" function combined with the length function "len()" to get the length of the list. We'll look at "range" in detail later in this chapter.

```
mystr = 'abc'
for i in range(len(mystr)):
    print('mystr char is:', mystr[i])
```

ValueError

A ValueError is raised when a function gets an argument of correct type but improper value. For example, a datetime object considers a time value for seconds < 60 or a month between 1 and 12 to be valid. A datetime month value of 13 creates an exception, and the Python Interpreter displays a ValueError. The syntax below is invalid for a datetime object:

```
d1 = datetime( 1999, 13, 31)
```

In Chapter 7, Examples 21, 23, and 31 demonstrate a ValueError.

ZeroDivisionError

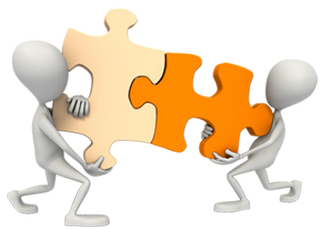
Raised when the second argument of a division or modulo operation is zero. The ZeroDivisionError is a subclass of ArithmeticError.

6. Try This

In this chapter we discuss

What is the Object Value?
String and Number Variable Values
Tuple Objects and Values
List Objects and Values
Dictionary Objects and Values
Does the Object Have a Value of None?
What is the Object Type?
What is the Length of the Object?
What are the Function/Method Arguments?
What Type of Object Does a Function Return?

The examples in the next chapter do have several suggestions on how to identify a particular bug and are great if you're experiencing the same problem. But this is the real world, and we both know that's not likely to happen. My concern with only using "Examples," is you'll rarely encounter the same issue when working with your code. Your situation is unique, and may not fit one of the examples.



To provide the missing piece of the debugging puzzle, I'm going to take some time in this Chapter to break down the debugging process into a reusable format. I'll cover some common issues. Unfortunately, my issue list isn't going to be all-inclusive, but I hope it kickstarts your debugging experience.

While an odd chapter title, "Try This," is a fitting name. When I was learning to program, I would share my dilemmas with a good friend. He would say, "Try this..." and offer a few suggestions. That little nudge in the right direction was a godsend that helped me find my way. I'm not sure if I can create that experience for you, but I'm going to try.

As we work through the next sections, you'll notice a common theme, where we look at these topics in different contexts.

- Object Values
- Types of Objects
- Length of Objects
- Passing Arguments to Functions or Methods.
- The Return Object of a Function

We'll look at the common objects outlined below. This list of objects isn't every possible Python object, but I think these are enough to get you started.

- Strings and Numbers
- Tuples

- Lists
- Dictionaries

It can be exasperating when you have a runtime or logic error and have no idea where to start debugging. The suggestions in this chapter may help you get started debugging your program.

6.1 What is the Object Value?

When I am debugging, often, the first thing I check is the object value. If you decide to take the hands-on approach and add print statements to your code or use Interactive Mode in the Console, the next few topics show you examples for strings, tuples, lists, and dictionaries. In the case of data structures like lists, tuples, and dictionaries, I'll also include the syntax to inspect all items or a single item. This content may be a bit repetitive, but on the plus side, this is a handy reference.

6.2 String and Number Variable Values

In this topic, I'll look at several ways to find the value of String and Number Variables.

Print the Value of a String Variable

Add a print statement to your script in the **Editor** window, and run your program.

mystring = "Purple Peanuts"

In the example the follows, I add a print statement in the Editor window. When I run the program, the output of the print statement is shown in the **Console** pane. The Console is the Python Shell.

A String Identifier: mystring

Value: purple peanuts
Reference: Chapter 4 - Add Print Statements

```
1  mystring = “purple peanuts”
2  print(mystring)
3
```

```
In [1]: runfile(‘C:/SampleScript.py’, wdir=’C:’)
purple peanuts
```

Variables in Imported Modules

To reference a variable inside another module, use dotted notation. In this example, I import a module “mymodule2” that has the variable “mystr2.” The expression **module2.mystr2** returns the value of **mystr2**.

```
import mymodule2

print(mymodule2.mystr2)
```

Inspect a Number Variable in Debug Mode

Debug Mode with Variable Explorer is a simple way to see object values as you step through your code. In this example, the Editor has one line to create a string variable named “mynumber.”

```
mynumber = 57
```

1. First, I run the program in Debug Mode.
2. Next, I type ”**mynumber**” in the **Console**. Because I am in Debug Mode, the Console prompt is **ipdb>**.

The **Variable Explorer** also shows the value of the “**mynumber”** variable.

Inspect a String Value with Interactive Mode

To see the value of the string in Interactive Mode, type the string name “mystring” in the **Console**.

```
mystring = “purple peanuts”
```

A String Identifier: mystring
Value: purple peanuts
Reference: Chapter 4 - Interactive Mode

1. Run the program to create the variables in memory.
2. Type “mystring” in the Console.
3. The Python Interpreter displays the value of “mystring” on the next line in the **Console**.

```
In [2]: mystring
Out[2]: ‘purple peanuts’
```

6.3 Tuple Objects and Values

In this topic, I’ll look at several ways to inspect Tuple Objects and Tuple Item values. To create a Tuple, use this syntax in the **Editor**:

```
mytuple = (‘Apple’, ‘Orange’, ‘Watermelon’)
```

Print All Tuple Item Values

Add a print statement to your script in the **Editor** window and run your program. The output of the print statement is shown in the **Console** pane.

All Items in the Tuple
Identifier: mytuple
Value: Name: Apple, Orange, Watermelon
Reference: Chapter 4 - Add Print Statements

```
2  print(mytuple)
3
4
```

```
In [1]: runfile('C:/SampleScript.py', wdir='C:')
('Apple', 'Orange', 'Watermelon')
```

Print a Tuple Item Value

To see the value of a Tuple item, type "mytuple[0] in the Console. The first item in the Tuple has an index value of **0**.

A Tuple Item
Identifier: mytuple[**0**]
Value: Name: Apple
Reference: Chapter 4 - Add Print Statements

1. Add a print statement to your script in the **Editor** window.
2. Run your program. The output of the print statement is shown in the **Console** pane.


```
1  print(mytuple[0])
2
3
```

```
In [1]: runfile('C:/SampleScript.py', wdir='C:')
Apple
```

Inspect All Tuple Items in Interactive Mode

To see all the values of a tuple named "mytuple," type "mytuple" in the **Console**.

All Items of the Tuple
Identifier: mytuple
Value: Name: Apple, Orange, Watermelon
Reference: Chapter 4 - Interactive Mode

1. Run the program to create the variables in memory.
2. Type "mytuple" in the Console in **Interactive Mode**. The Python Interpreter displays the value of "mytuple" on the next line in the **Console**.

Object Type	Identifier	Value
A Tuple	mytuple	purple peanuts

```
In [2]: mytuple
Out[2]: ('Apple', 'Orange', 'Watermelon')
```

Inspect A Tuple Item in Interactive Mode

To see the value of a tuple item "mytuple," type "mytuple" in the **Console**. The first item in the Ruple has an index value of **0**.

A Tuple Item
Identifier: mytuple[**0**]
Value: Name: Apple
Reference: Chapter 4 - Interactive Mode

1. Run the program to create the variables in memory.
2. Type "mytuple[0]" in the Console in **Interactive Mode**. The Python Interpreter displays the value of the "mytuple[**0**]" object on the next line in the **Console**.

```
In [2]: mytuple[0]
Out[2]: 'Apple'
```

6.4 List Objects and Values

In this topic, I'll look at several ways to inspect List Objects and List Item values. To create a List, use this syntax:

```
mylist = ['Soda', 'Water', 'Coffee']
```

Print All List Item Values

In this example, I print out all List item values to the **Console** pane.

All Items of the List
Identifier: mylist
Value: Name: Soda, Water, Coffee

Reference: Chapter 4 - Add Print Statements

1. Add a print statement to your script in the **Editor** window.
2. Run your program. The output of the print statement is shown in the **Console** pane.

```
1
2
3
```

```
mylist = ['soda', 'water', 'coffee']
print(mylist)
```

```
In [1]: runfile('C:/SampleScript.py', dir='C:')
('Soda', 'Water', 'Coffee')
```

Print the Value of a List Item

To print the value of a List item, add a print statement to your script in the **Editor** window, and run your program. The second item in the List has an index value of **1**.

A List Item
Identifier: mylist[**1**]
Value: Soda, Water, Coffee
Reference: Chapter 4 - Add Print Statements

1. Add a print statement to your script in the **Editor** window.
2. Run your program. The output of the print statement is shown in the Console pane. In the example below, the output is "Water."

```
1  mylist = [‘soda’, ‘water’, ‘coffee’]
2  print(mylist[1])
3
```

```
In [1]: runfile(‘C:/SampleScript.py’, wdir=’C:’)
soda
```

Inspect a List Item in Debug Mode

Debug Mode with Variable Explorer is a simple way to see object values as you step through your code. In this example, I run the program in **Debug Mode**. Because I am in Debug Mode, the prompt is **ipdb>**.

Notice the **Variable Explorer** shows the values in the “**drinks”** list. The third item is “coffee” and has an index value of **2**.

A List Item
Identifier: drinks[**2**]
Value: coffee
Reference: Chapter 4 - Debug Mode

1. Run the program in Debug Mode.
2. Type ”**drinks[2]** in the **Console**.

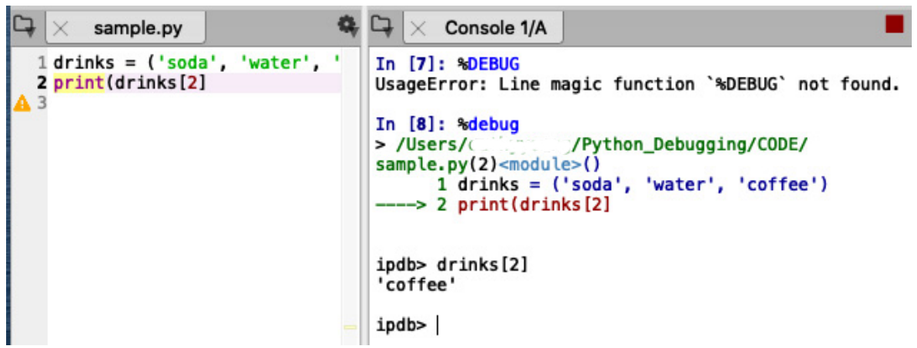


Figure 6.1

Inspect All Items of a List in the Console

To see all values of a List, type the list name in the **Console**.

All Items in the List
Identifier: mylist
Value: Name: Soda, Water, Coffee
Reference: Chapter 4 - Interactive Mode

1. Run the program.
2. Type “mylist” in the Console. The Python Interpreter displays the value of “mylist” on the next line in the **Console**.

```
In [2]: mylist
Out[2]: [‘Soda’, ‘Water’, ‘Coffee’]
```

Inspect a List Item in the Console

In this example, I run the program and type "mylist[1]" in the Console. The second item in the List has an index value of **1**.

A List Item
Identifier: mylist[**1**]
Value: Name: Water
Reference: Chapter 4 - Interactive Mode

1. Run the program.

2. Type "mylist[**1**]" in the **Console**. The Python Interpreter displays the value of mylist[**1**] in the **Console**.

```
In [2]: mylist[1]
Out[2]: Water
```

6.5 Dictionary Objects and Values

In this topic, I'll look at several ways to inspect Dictionary Objects and Dictionary Key-Pair values. To create a Dictionary, use this syntax:

```
mydictionary = {'Name': 'Zimmerman',
                'Grade': 'A',
                'Course': 'Python Programming'}
```

Print the Value of a Dictionary Key-Pair

To print the value of a Dictionary item, add a print statement to your script in the **Editor** window, and run your program.

A Dictionary Key-Pair

Identifier: mylist[**1**]
Value: Soda, Water, Coffee
Reference: Chapter 4 - Add Print Statements

1. Add a print statement to your script in the **Editor** window.

2. Run your program. The output of the print statement is shown in the Console pane. In the example below, the output is "Water."

```
1 mydictionary = {'Name': 'Zimmerman',
2                 'Grade': 'A',
3                 'Course': 'Python Programming'}
4 print(dictionary['Name'])
5
```

```
In [1]: runfile('C:/SampleScript.py', wdir='C:')
Water
```

Inspect All Dictionary Items in the Console

To see all the key-pairs of a Dictionary named "mydictionary," type "mydictionary" in the **Console**. In this example, I run the program, and the Python Interpreter displays the value of "mydictionary" on the next line in the **Console**.

All Dictionary Items
Identifier: mydictionary
Key-Pair Values: Name: Zimmerman, Grade: A
Reference: Chapter 4 - Interactive Mode

1. Run the program.

2. Type "mydictionary" in the Console. The Python Interpreter displays the value of "mylist" on the next line in the **Console**.

```
In [2]: mydictionary
Out[2]: 'Name': 'Zimmerman', 'Grade': 'A'
```

Inspect a Dictionary Item Value in the Console

Once you know a key name in a Dictionary, you can find the value of the Dictionary key-pair item. In this example, I run the program and type mydictionary[‘grade’] in the Console. The key name is “grade.”

A Dictionary Item
Identifier: mydictionary[‘Grade’]
Value: A
Reference: Chapter 4 - Interactive Mode

1. Run the program.
2. Type “mydictionary[‘grade’]” in the **Console**. The Python Interpreter displays the value of “mydictionary[‘Grade’]” on the next line in the **Console**.

```
In [2]: mydictionary['Grade']
Out[2]: 'A'
```

Inspect a Dictionary Item in Variable Explorer

Debug Mode with Variable Explorer is a simple way to see object values as you step through your code. In this example, I create a variable “schoolinfo” in a “for statement” on line 5. When I step through the “for loop” twice in Debug Mode, the “schoolinfo” variable has the value of the second key-par in “mydictionary.”

A Dictionary Item
Identifier: mydictionary[‘Grade’]
Variable: schoolinfo
Value: A
Reference: Chapter 4 - Debug Mode Variable Explorer

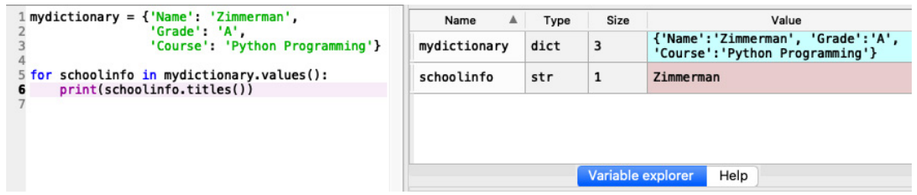


Figure 6.2 Variable Explorer

When you double click on the name of a Dictionary in Variable Explorer, a pop-up window opens

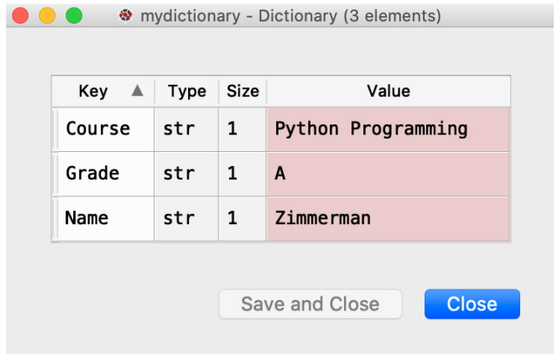


Figure 6.3 Pop-up in Variable Explorer

6.6 Does the Object have a Value of NoneType or Whitespace?

When importing external data into Python data structures, it is not uncommon to have items with a value of “None” or unexpected whitespace. For example, if you import an Excel worksheet with empty cells, those List items have a value of “None.” Also, sometimes strings are null or equal to “”.

Functions with return values can also sometimes return “None.” The value “None” is returned when you don’t have a return value for all paths in the function, as explained in the Chapter 3 topic, “Function Return Values.” Example 17 in Chapter 7 also illustrates a function that returns “None.”

The “None” value causes problems when working with functions that expect a particular type or value for an object. For example, the DateTime library function “.strftime” expects an object of type “datetime”, “time”, or “date.” If the object has a value of “None,” the Python interpreter displays an error, as shown in Chapter 7, **Example 20**.

Sadly, this is also one way Divide by Zero errors happen in a program. When you convert an item in a data structure from a string to an “int,” an item with a value of “None” becomes a zero. The Divide by Zero error is shown in the Examples Chapter, **Example 24 and 17**, and includes a sample “if statement” to test for a “None” value.

Whitepsace

Another cause of unexpected consequences is whitespace. There may be a tab, line return, or some other character that impacts a search or comparison. These functions are useful for removing those unseen characters.

lstrip() Remove left whitepspace characters.

rstrip() Remove right whitespace characters.

strip() Remove whitespace characters from both sides.

6.7 What is the Object Type?

In Debug Mode, the Variable Explorer shows the type of the Object. Another option is to add a print statement or type in the Console, as outlined earlier.

print(type(mystring))

print(type(myfunction())

Object	Syntax
number, string or data structure	type(mystring) type(mytuple) type(mylist) type(mydictionary)
a Tuple item	type(mytuple[0])
a List item	type(mylist[0])
a Dictionary item	type(mydictionary(‘Name’]

6.8 What is the Length of the Object?

The Len() function shows the length of the string or the number of items in a data structure. For example, len(mydictionary) would return the number of Dictionary key pairs. len(mylist) would return the number of items in the List.

Object	Syntax
number, string or data structure	len(mystring) len(mytuple) len(mylist) len(mydictionary)
a Tuple item	len(mytuple[0])
a List item	len(mylist[0])
a Dictionary item	len(mydictionary(‘Name’]

6.9 What are the Function Arguments?

First, identify what type of arguments a Function expects. Then, check your arguments to be sure they are the correct type and value, as outlined previously. The Help pane and searching the Internet are a great way to review the definition of the function to ensure both the arguments and return objects are what you expect. In the Console, you can also use

these introspection functions for more information about your object, as outlined in Chapter 4 in the topic "Interactive Mode."

```
In [1]: myfunction?
In [2]: dir(myfunction)
In [3]: help(myfunction)
```

The reference for built-in functions can be found on **docs.python.org**. Example 26 in Chapter 7 looks at these options.

The Function Call Signature

Ideally every function should include a "function call signature." The "inspect" module includes the **signature(myfunction)** function that returns a function's docstring. The **getfullargspec()** function displays the names and default values of a function. Chapter 7, Example 20 uses the **signature()** function to retrieve parameter information.

For additional information on the "**openpyxl**" method "**load_workbook**," I can look at the "calThe Console is tl signature" for a callable object. In Python v3.x, PEP 362 specifies the function signature object and lists each parameter accepted by a function. In the **Console**, import the module and print the signature for the object, as shown below. Chapter 7, Example 20, also illustrates this syntax from PEP 362. The inspect library can also display the DocString mentioned in the topic "Classes" in **Chapter 3**.

```
1  from inspect import signature
2  print(str(signature(load_workbook)))
```


Inspect the Docstring

Hopefully, functions include a "docstring" explaining the purpose of the function and what the function returns, if anything. In this next example, I'm inspecting the arguments, signature, and docstring of "myfunction." The last line retrieves the "docstring" using the dotted notation "**myfunction.__doc__**."

```
1   from inspect import *
2
3
4   def myfunction(a: int, b: float):
5           '''
6           this function takes an int and a float
7           and doesn't do anything
8           '''
9           pass
10
11  from inspect import *
12  print(getfullargspec(myfunction))
13  print(signature(myfunction))
14  print(myfunction.__doc__)
```

6.10 What Type of Object Does a Function Return?

The Interactive Mode of the Console is a great way to check what a function returns. You may want to check the value, type, or length.

print(myfunction())

type(myfunction())

len(myfunction())

The Help pane is another way to review the definition of the function and docstring, to ensure both the arguments and return object is what you expect. A function returns one object, but that object might

be a tuple with several items as outlined in Chapter 3. Example 26 in Chapter 7 looks at these options.

7. Examples

In this Chapter, we take “debugging” for a spin. These examples build on everything we’ve looked at so far. It doesn’t matter if you landed here first, or read everything to this point. Either way, I provide references to those previous topics, in case you want to take a brief sojourn to review them.

Ex	Description	Built-in Error Type	Kind
1	List index out of range	IndexError	Runtime
2	List index out of range (Example 1 continued)	IndexError	Runtime
3	Wrong Variable Name	NameError	Logic
4	Invalid Assignment	Runtime	Runtime
5	While statement not indented	IndentationError	Syntax
6	Method arguments incorrect	AttributeError	Runtime
7	Empty Block is illegal	IndentationError	Syntax
8	Parentheses not matched	SyntaxError	Syntax
9	Colon misssing	SyntaxError	Syntax
10	Case sensitive	NameError	Runtime
11	Keyword missing	SyntaxError	Syntax
12	Illegal characters or keyword	SyntaxError	Syntax
13	Misspelled identifier	NameError	Runtime
14	File doesn't exist	FileNotFoundError	Runtime
15	Adding incorrect types	TypeError	Runtime
16	Misspelled keyword	SyntaxError	Syntax
17	Value is none	TypeError	Runtime
18	Module not found	AttributeError	Runtime
19	Module not found	ModuleNotFoundError	Runtime
20	Key not in Dictionary	KeyError	Runtime
21	Arugment is incorrect type	ValueError	Runtime
22	Object not found- NameError	NameError	Runtime
23	Invalid data passed to method	ValueError	Runtime
24	Calculation causes a ZeroDivision Error	ZeroDivision Error	Runtime
25	There is a mistake in a math calculation		Logic Error
26	Assigning datetime value causes ValueError	ValueError	Runtime

Ex	Description	Built-in Error Type	Kind

For each example that follows, I use a systematic approach to examine the program.

Intended Outcome: What I want the program to do is the Intended Outcome.

Actual Result: What the program did is the Actual Result.

Incorrect Code: The Incorrect Code is the actual code that is not working properly.

Debugging Experiment: The steps I use to "debug" what the program is actually doing comprise the Debugging Experiment.

How to Resolve the Issue: This section is a brief description of the change to the code to resolve the issue.

Correct Code: The Correct Code works as I intended.

References: The References list previous topics related to this example.

Ex 7.1 List Index Out of Range

Description: The list index is out of range.

Intended Outcome

There are two Lists in this program, "meals" and "fruits." I want the program to loop through each list and print the items in order.

Actual Result

The **Console** output shows the print statement on line 6 repeats with the first item in the List.

Incorrect Code

This is the Example 1 Code before any changes. Can you spot the three areas we need to fix? We'll look at each error in Examples 1-3.

```
meals = ['breakfast', 'lunch', 'snack', 'dinner']
fruits = ['apple', 'orange', 'grape']
i = 0
while i < 4:
    j = 0
    print("my meal is: ",  meals[i])
    while j < 4:
        print("My choice of fruit is: ", fruits[i])
        j = j + 1
i = i + 1
```

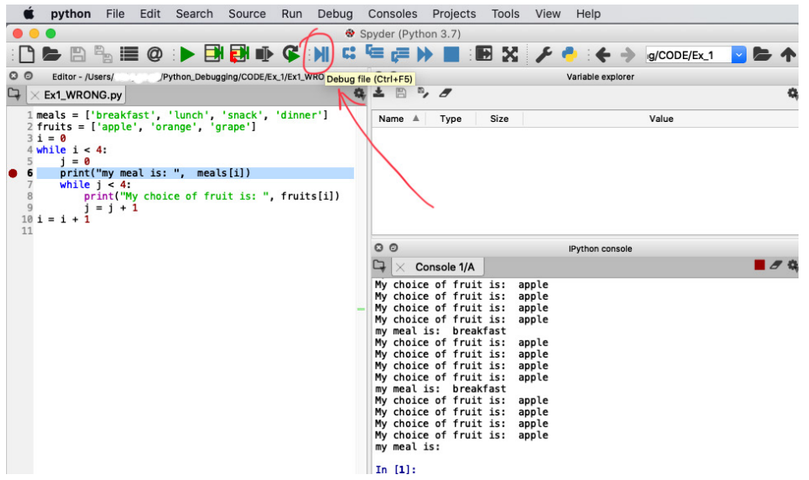

Debugging Experiment

In this example, when I run the program, it loops continuously. Next, I use Debug Mode to research what is happening.

1. Run the program. The program runs endlessly. The program is in an infinite loop. In the **Console**, the Python Interpreter repeatedly outputs the print statement from line 6.

2. To stop the program, in the **Consoles** menu, I select "Restart kernel."

   Double click twice on line 6 to add a breakpoint. Select "Debug File" on the menu, as shown below.



3. The **Console** prompt changes to **ipdb>** to indicate you are in **Debug Mode.** In the **Console**, type "**s**" to step through the program. The Python Interpreter moves to the next line.

   Type "**s**" a few times, and you'll notice the program loops back to line 4. Variable Explorer shows the value of "**i**" is 0 and is not changing as the program runs.

4. In the earlier figure, the print statement on line 6 was repeatedly output to the **Console**, indicating that the "while loop" on line 4 is looping continuously. I suspect that my counter "i" is not incremented properly on line 10.

   In the **Console** pane, type "**q**" to quit **Debug Mode**. The next topic outlines the change to resolve this issue. Press Control + C to stop a program, or choose "restart kernel" from the Console menu.

How to Resolve the Issue

In the **Editor**, I indent line 10. Now line 10 is part of the while loop that begins on line 4. One error is fixed, but there is another error we'll look at in Example 2.

Good Code

```
meals = ['breakfast', 'lunch', 'snack', 'dinner']
fruits = ['apple', 'orange', 'grape']
i = 0
while i < 4:
    j = 0
    print("my meal is: ",  meals[i])
    while j < 4:
        print("My choice of fruit is: ", fruits[i])
        j = j + 1
    i = i + 1
```


Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Indexing
Chapter 4 - Add Print Statements
Chapter 4 - Interactive Mode
Chapter 4 - Debug Mode Variable Explorer
Chapter 4 - Infinite Loop
Chapter 5 - Traceback
Chapter 5 - IndentationError
Chapter 5 - IndexError
Chapter 6 - Check Object Type

Ex 7.2 Index Error

Description: The list index is out of range. This kind of runtime error appears when you run the program.

Intended Outcome

There are two Lists in this program, "meals" and "fruits." I want the program to loop through each list and print the items.

Actual Result

The print statement on line 8 **fruits[i]** causes an IndexError because the index is out of range..

Incorrect Code

Example 2 code before any changes follows.

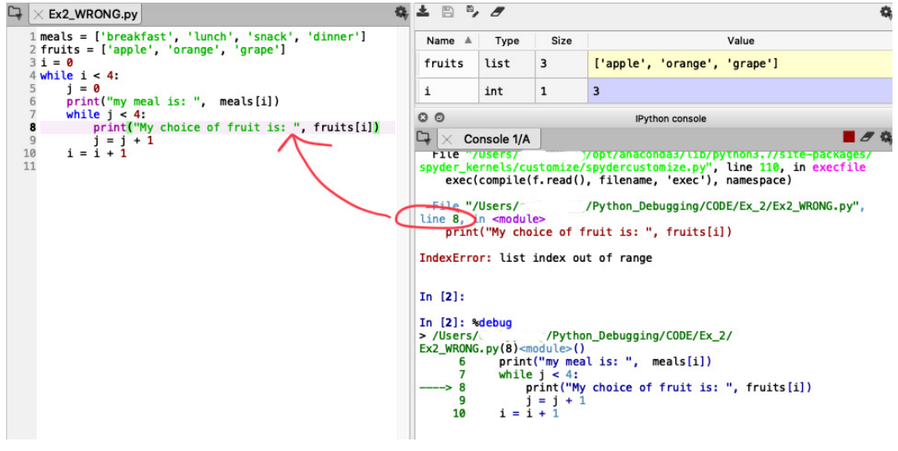
```
1     meals = [‘breakfast’, ‘lunch’, ‘snack’, ‘dinner’]
2     fruits = [‘apple’, ‘orange’, ‘grape’]
3     i = 0
4     while i < 4:
5          j = 0
6          print(“my meal is: “,  meals[i])
7          while j < 4:
8               print(“My choice of fruit is: “, fruits[i])
9               j = j + 1
10         i = i + 1
```

Debugging Steps

In this Example the program halts. I use Debug Mode to research what is happening.

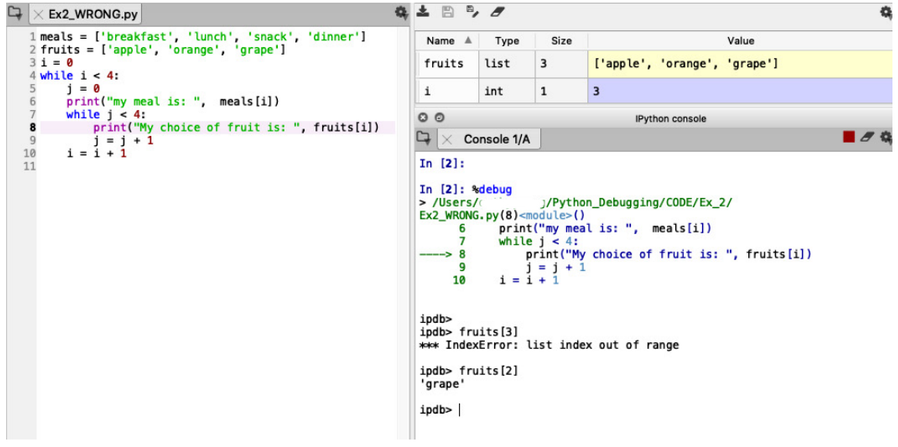
1. Run the program. The Python Interpreter halts because of an exception and displays an **IndexError** in the **Console**.

   In the **Console**, the Python Interpreter displays a Traceback message telling me the error is in Line 8. If I click on “line 8” in the **Console**, it is a hyperlink to that location in my code in the Editor pane.



2. Type **%debug** In the **Console,** pane to start Debug Mode. The **Console** prompt changes to **ipdb>**.

3. Type **fruits[3]** In the **Console,** pane. The Variable Explorer shows j has a value of 3, so fruits[i] evaluates to fruits[3]. The message “IndexError: list index out of range” is displayed.



4. Now type **fruits[2]** In the **Console,** pane. The value 'grape' is displayed. Grape is the last item in the "fruits" list. The range of the fruits list is 0 to 2.

   If this had been a long list, I could have typed **len(fruits)** In the **Console**, to see how many items were in the list.

   In the Console, pane, type "**q**" to quit Debug Mode. The next topic outlines the change to resolve this issue.

How to Resolve the Issue

In the **Editor**, I update line 7.

```
while j < 3:
```

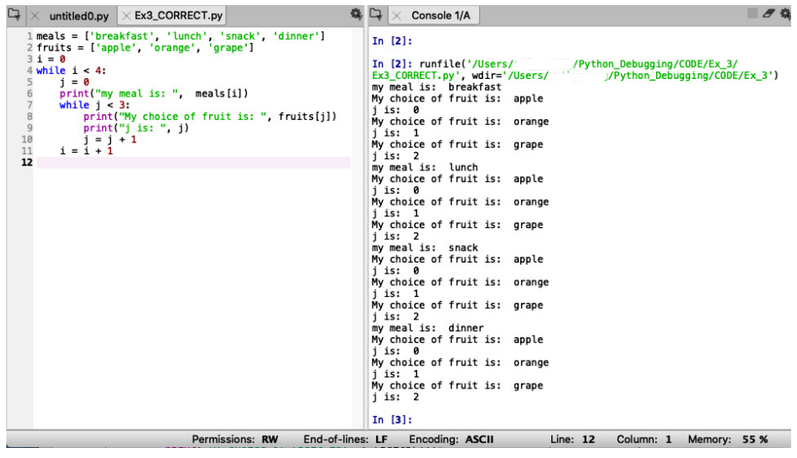


Figure 7.1 Corrected Code

Good Code

```
1  meals = ['breakfast', 'lunch', 'snack', 'dinner']
2  fruits = ['apple', 'orange', 'grape']
3  i = 0
4  while i < 4:
5      j = 0
6      print("my meal is: ",  meals[i])
7      while j < 3:
8          print("My choice of fruit is: ", fruits[j])
9          j = j + 1
10     i = i + 1
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Indexing
Chapter 4 - Debug Mode
Chapter 4 - Interactive Mode
Chapter 4 - Debug Mode Variable Explorer
Chapter 5 - Traceback
Chapter 5 - IndexError
Chapter 6 - Check Object Type
Chapter 6 - Check Length of Object

Ex 7.3 Wrong Variable

Description: The code references the wrong variable name.

In this example, there is a flaw in the overall design or logic of my program. The program does what I coded, but the outcome is not what I intended. In the Console, my print statement does not iterate through my list of fruits.

Intended Outcome

The program should print a list of fruits for each meal.

Actual Result

The program halts with an IndexError exception.

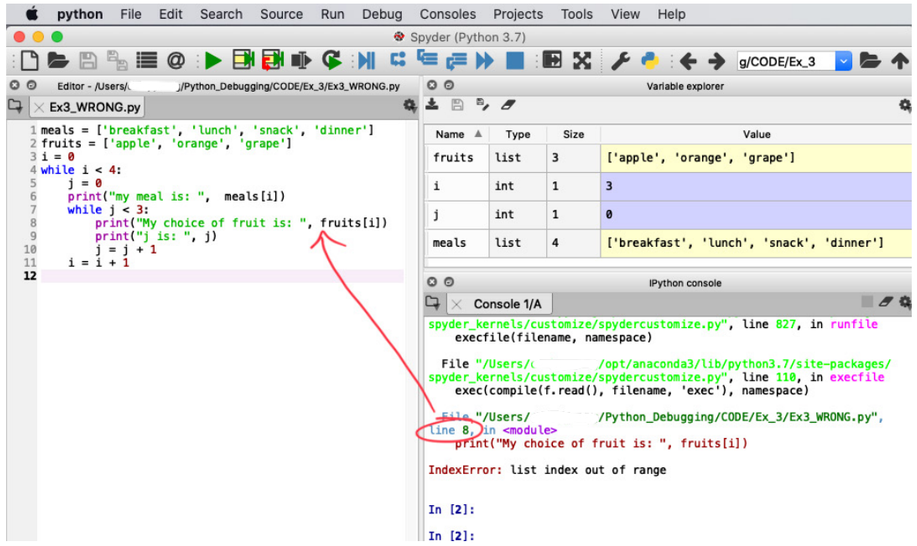
Incorrect Code

Example 3 code before any changes follows.

```
meals = ['breakfast', 'lunch', 'snack', 'dinner']
fruits = ['apple', 'orange', 'grape']
i = 0
while i < 4:
    j = 0
    print("my meal is: ",  meals[i])
    while j < 3:
        print("My choice of fruit is: ", fruits[i])
        print("j is: ", j)
        j = j + 1
    i = i + 1
```

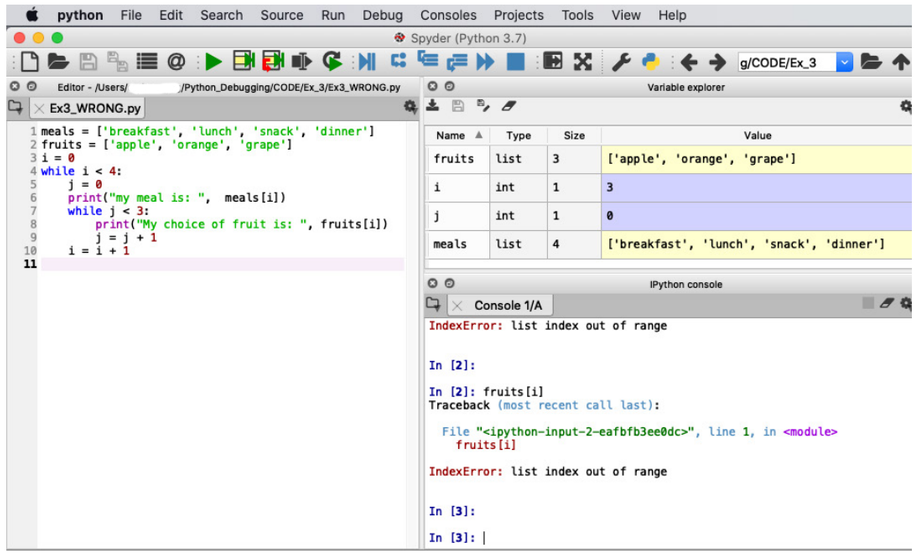
Debugging Steps

1. When I run the program, the Python Interpreter halts with an error. The Traceback shows the IndexError was caused by line 8.



2. In the **Console**, I type **fruits[i]**, which returns the same IndexError.

   Variable Explorer shows "**i**" has a value of 3. The **fruits** list has three items, and the indices are 0-2. Now I realize I should have used the "**j**" variable as a counter for the **fruits** list.



At this point, I just typed a statement in the **Console**, so I don't need to exit **Debug Mode**. Instead, I'll update my script in the **Editor** pane to resolve the issue.

How to Resolve the Issue

In the print statement I change the variable to "j."

```
print("My choice of fruit is:", fruits[j])
```


Good Code

```
meals = ['breakfast', 'lunch', 'snack', 'dinner']
fruits = ['apple', 'orange', 'grape']
i = 0
while i < 4:
    j = 0
    print("my meal is: ",  meals[i])
    while j < 3:
        print("My choice of fruit is: ", fruits[j])
        print("j is: ", j)
        j = j + 1
    i = i + 1
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - IndentationError
Chapter 4 - Debug Mode Variable Explorer
Chapter 6 - Check Object Type

Ex 7.4 Invalid Assignment

Description: The assignment statement is invalid. This kind of runtime error is uncovered when you run the program.

Intended Outcome

Print "mylist" items to the Console.

Actual Result

The **Console** output shows the print statement on line 4 repeats with the first item in the List.

Incorrect Code

This is the Example 4 code before any changes.

```
mylist = [‘soda’, ‘water’, ‘coffee’]
i = 0
while i < 3:
     print(mylist[i])
     i =+ 1
```

Debugging Steps

1. Run the program. The program runs endlessly. The program is caught in an an infinite loop. In the **Console**, the Python Interpreter repeatedly outputs the print statement from line 4.

2. To stop the program, in the **Consoles** menu, I select “Restart kernel.”

   Double click twice on line 4 to add a breakpoint. Select “Debug File” on the menu, as shown below.

3. The **Console** prompt changes to **ipdb>** to indicate you are in **Debug Mode.** In the **Console**, type “**s**” to step through the program. The Python Interpreter moves to the next line.

   Type “**s**” a few times, and you’ll notice the program loops back to line 3. Variable Explorer shows the value of “**i**” is 0 and is not changing as the program runs.

How to Resolve the Issue

The counter is not incremented on line 5 because I reversed the syntax. The statements “i = i + 1” and “i **+=** 1” both increment the “i” counter.

Good Code

```
mylist = [‘soda’, ‘water’, ‘coffee’]
i = 0
while i < 3:
     print(mylist[i])
     i += 1
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Debug Mode
Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Infinite Loopt

Ex 7.5 While Indentation Error

Description: The “while” statement is not indented properly. The **Console** displays an **IndentationError**.

Intended Outcome

Print a list of numbers.

Actual Result

When I run the program, it halts with an IndentationError.

Incorrect Code

This is the Example 5 code before any changes.

```
wadofcash = [111, 222, 333, 444]
i = 0
x = 3
while i <= x:
print(wadofcash[i])
     i = i + 1
```

Debugging Steps

Spyder displays a yellow warning triangle next to the print statement on line 6. When I hover my mouse over the triangle, a pop-up message is displayed, as shown below.

If I run the program, the **Console** displays an **IndentationError**.

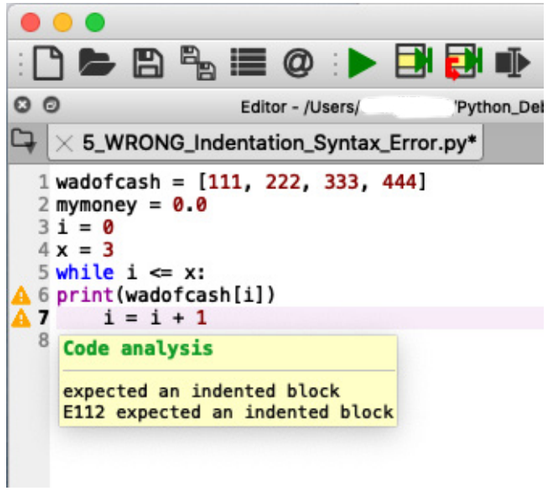


Figure 7.2 Indentation Warning

How to Resolve the Issue

Indent the print statement on line 6.

Good Code

```
wadofcash = [111, 222, 333, 444]
i = 0
x = 3
while i <= x:
     print(wadofcash[i])
     i = i + 1
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 -Help()
Chapter 5 - Traceback
Chapter 5 - IndentationError

Ex 7.6 Incorrect Method Arguments

Description: The openpyxl "cell" method has incorrect attributes. The **Console** displays an **AttributeError**.

Intended Outcome

My intention was for the code to open an Excel file and print each column 2 value as the program iterates through the rows.

Actual Result

The program halted with an "AttributeError" exception when run with the Python 2.7 Interpreter. The program runs fine on my Python 3 environment. This **AttributeError** indicates the Python Interpreter doesn't recognize the line 8 syntax.

Line 8 is calling a method in a class. In the discussion of Classes in Chapter 3, we created an instance of a class. Valid attribute names of a class include both "**data attributes**" and "**methods**." Python objects have attributes that are referenced with the dot notation.

Incorrect Code

This is the Example 6 code before any changes.

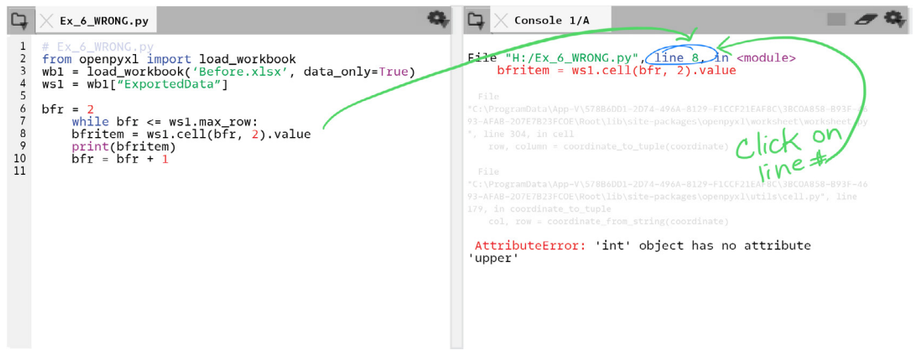
```
from openpyxl import load_workbook
wb1 = load_workbook('Before.xlsx', data_only=True)
ws1 = wb1["ExportedData"]
bfr = 2
while bfr <= ws1.max_row:
    bfritem = ws1.cell(bfr, 2).value
    print(bfritem)
    bfr = bfr + 1
```

Debugging Experiment

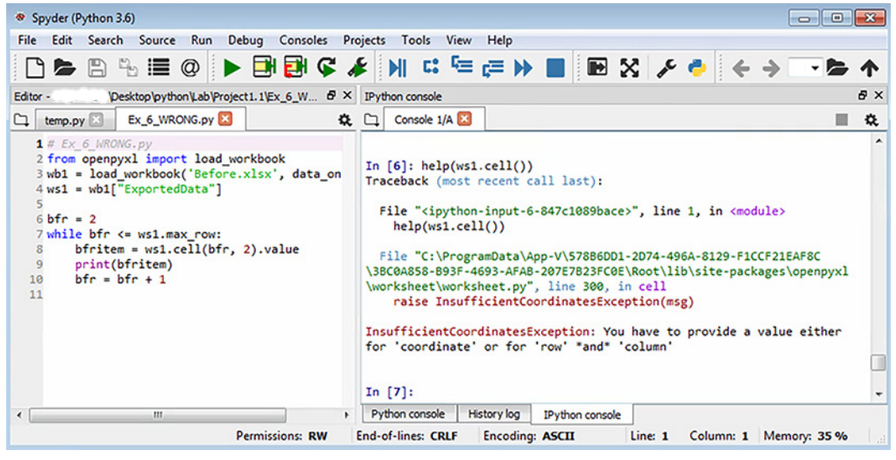
In this example, when I run the program, the Python Interpreter prints an "AttributeError" to the Console. I use Help to research what is happening.

1. Run the program. The Python Interpreter halts because of an exception and displays an **AttrbuteError** in the **Console**.

In the **Console**, the Python Interpreter displays a Traceback message telling me the error is in Line 8. If I click on "line 8" in the **Console**, it is a hyperlink to that location in my code **Ex_6_WRONG.py** in the Editor pane.



2. The issue seems related to the object on line 8, and I suspect there is something wrong with the syntax for the object value. I'm curious about what syntax I should use with the cell method. In the **Console,** type **help(ws1.cell())** or **dir(ws1.cell())** for more information on the object.



3. Help indicates the Python Interpreter could not use the values for row and column when calling the function "cell." In line 8, I need to add the argument keywords (or names).

In the Console pane, type "**q**" to quit Debug Mode. The next topic outlines the change to resolve this issue.

How to Resolve the Issue

In the Editor pane, update line 8 of **Ex_6_WRONG.py** to use the keywords, as shown below. This change ensures the program runs with Python 2.7 or 3.7.

```
bfritem = ws1.cell(row=bfr, column=2).value
```

Good Code

```
from openpyxl import load_workbook
bfritem = ws1.cell(row=bfr, column=2).value
wb1 = load_workbook('Before.xlsx', data_only=True)
ws1 = wb1["ExportedData"]
bfr = 2
while bfr <= ws1.max_row:
    bfritem = ws1.cell(row=bfr, column=2).value
    print(bfritem)
    bfr = bfr + 1
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Attributes
Chapter 3 - Methods
Chapter 4 - Help()
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 5 - AttributeError
Chapter 6 - Check Arguments

Ex 7.7 Empty Block of Code

Description: An empty block of code is illegal. The **Console** displays an **IndentationError**.

Intended Outcome

While writing a program, I want to have a block of code that does nothing. At some point, I intend to add logic.

Actual Result

The Python Interpreter has an IndentationError exception.

Incorrect Code

This is the Example 7 code before any changes.

```
for mynum in [157, 19, 56]:
    if mynum == 157:
    else:
        print('Happy birthday, you are', mynum)
```

Debugging Steps

The **Console** shows an **IndentationError** on line 3. As is often the case, the actual error is the line above.

How to Resolve the Issue

In keeping with my design goal, I want the code to do nothing, so I add a pass() statement.

Good Code

```
for mynum in [157, 19, 56]:
     if mynum == 157:
          pass
     else:
          print('Happy birthday, you are', mynum)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - Traceback
Chapter 5 - RuntimeError
Chapter 5 - SyntaxError

Ex 7.8 Parentheses Not Matched

Description: Parentheses not matched. The Co**nsole** displays a **SyntaxError**.

Intended Outcome

On line 3, I want to calculate projected sales.

Actual Result

When I run the program, the **Console** displays a Syntaxerror in line 3.

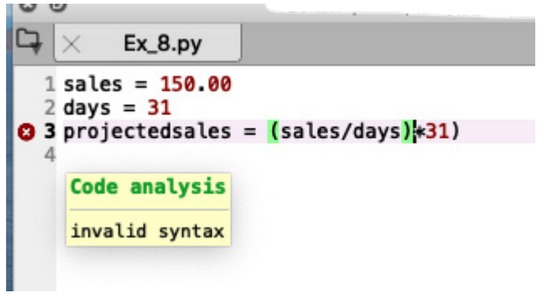
Incorrect Code

This is the Example 8 code before any changes.

```
sales = 150.00
days = 31
projectedsales = (sales/days)*31)
```

Debugging Steps

The **Editor** displays a red x to the right of line 3, indicating the parser identified invalid syntax. When I hover my mouse over the parentheses on that line, the paired parentheses are highlighted in green.



When I move my mouse to the end of the line, the last parenthesis is highlighted in orange, indicating there is no corresponding parenthesis.

How to Resolve the Issue

In line 3, I added an open parenthesis in front of "sales" as shown below.

Good Code

```
sales = 150.00
days = 31
projectedsales = ((sales/days)*31)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - SyntaxError

Ex 7.9 Missing Colon

Description: The colon is misssing. The **Console** displays a **SyntaxError**.

Intended Outcome

Print mylist items to the Console.

Actual Result

When I run the program, there is a SyntaxError.

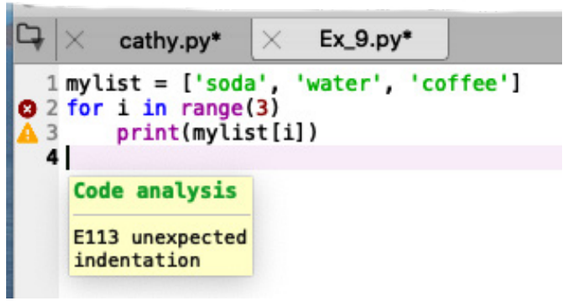
Incorrect Code

This is the Example 9 code before any changes.

```
mylist = ['soda', 'water', 'coffee']
for i in range(3)
    print(mylist[i])
```


Debugging Steps

The Editor has a **red x** by line 32 and a **yellow triangle** to the left of line 3. When I hover over the yellow triangle, a pop-up message is displayed, "unexpected indentation."



How to Resolve the Issue

Line 2 is a "for" statement. I added a colon at the end of the line.

Good Code

```
mylist = ['soda', 'water', 'coffee']
for i in range(3):
    print(mylist[i])
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - SyntaxError

Ex 7.10 Case Sensitive

Description: Python is case sensitive. Variables with the wrong case are interpreted as misspelled by the Python Interpreter and cause a NameError.

Intended Outcome

Print mylist items to the Console.

Actual Result

When I run the program, the Console displays a NameError.

Incorrect Code

In this example, I changed the case on the variable "myList" with the intention of causing a NameError.

> If you previously ran Example 9, "mylist" is still in memory, and you won't see a NameError. I wanted to demonstrate that sometimes you need to reset the "namespace" to be certain you're looking at values from this program.

This is the Example 10 code before any changes.

```
myList = ['soda', 'water', 'coffee']
for i in range(3):
    print(mylist[i])
```


Debugging Steps

1. To clear memory (the namespace), in the **Consoles** menu, select "Restart kernel." You could also type %reset in the Console.
2. When I run the program, the **Console** displays a **NameError** on line 3.

How to Resolve the Issue

On the first line, I change "mylist" to all lowercase.

Good Code

```
mylist = ['soda', 'water', 'coffee']
for i in range(3):
    print(mylist[i])
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - NameError

Ex 7.11 Missing Keyword

Description: A keyword is missing when defining a function causing a SyntaxError.

Intended Outcome

This code creates a new function that adds two numbers together.

Actual Result

The Editor displays a red "x" to the left of line 1. When I run the program, the Console Traceback says there is a SyntaxError on line 1.

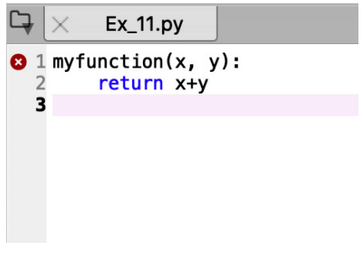


Figure 7.3 Function with Error

Incorrect Code

This is the Example 11 code before any changes.

```
myfunction(x, y):
    return x+y
```

Debugging Steps

When looking at line 1, I see that I left off the keyword "def." I could also search online for the Python documentation on defining a function.

How to Resolve the Issue

Add "**def**" to the beginning of line 1.

Good Code

```
def myfunction(x, y):
    return x+y
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 5 - SyntaxError

Ex 7.12 Illegal Character

Description: Illegal character in identifier name causes a SyntaxError.

Intended Outcome

This code creates a string variable.

Actual Result

The program halts with a SyntaxError.

Incorrect Code

This is the Example 12 code before any changes.

```
my$str = ‘hello’
print(my$str)
```

Debugging Steps

The Editor has a red “x” to the left of line 1, indicating there is a syntax error in my assignment statement. Normally, the Editor highlights functions and methods in purple, and variables are black. The formatting is different because the Editor is unable to interpret the code.

When I run the program, the Console displays an arrow pointing to the invalid character in my string name.

```
  File "/Users/          /Python_Debugging
Ex_12.py", line 1
    my$str = 'hello'
      ^
SyntaxError: invalid syntax


In [2]:

In [2]:
```

Figure 7.4 Invalid Character in the Console

How to Resolve the Issue

Special characters like $, #, and @ are not allowed for variable names. I rename my variable to “mystr” and the SyntaxError is resolved.

Good Code

```
mystr = ‘hello’
print(mystr)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 4 - Debug Mode
Chapter 4 - Interactive Mode
Chapter 5 - NameError
Chapter 5 - SyntaxError

Ex 7.13 Undefined Name

Description: Undefined name when variable is misspelled. Traceback shows a NameError.

Intended Outcome

Line two calculates profit using the “royalties” variable.

Actual Result

The Editor has a yellow triangle to the left of line 2. When I run the program the Traceback shows a NameError.

Incorrect Code

This is the Example 13 code before any changes.

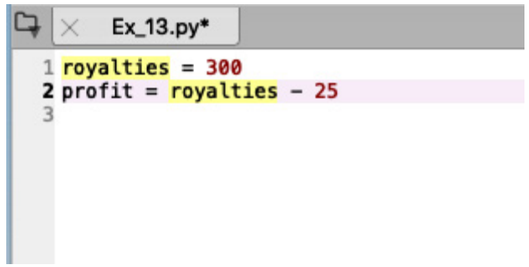
```
royalties = 30
profit = royaltie - 2
```

Debugging Steps

In the Editor, if I click on the variable name, the Editor highlights all instances of the variable throughout the code. I assign 300 to the "royalties" variable in line 1. The variable name is misspelled on line 2.

How to Resolve the Issue

On line 2 I correct the variable name. The Editor pane highlights all instances of the variable name in yellow, so I know I am using the variable I intended on line 2.



Good Code

```
royalties = 30
profit = royalties - 2
```


Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 5 - NameError

Ex 7.14 FileNotFound

Description: While reading a file, the Console halted with an error "FileNotFoundError." In this example, factors outside your control impact your program.

Intended Outcome

This program opens a text file and prints the contents in the Console.

Actual Result

The Console displays a "FileNotFoundError."

Incorrect Code

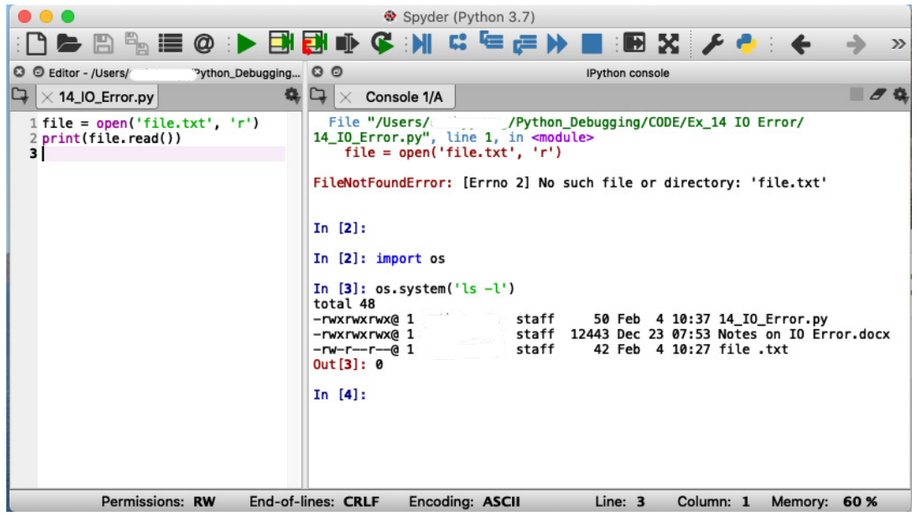
This is the Example 14 code before any changes.

```
file = open('file.txt', 'r')
print(file.read())
```

Debugging Steps

The "FileNotFoundError" has a base class of "IOError." I want to check the filename in my OS directory.

1. Import the “os” library to work with the OS commands. In the Console, type “import os” and then type the list command, as shown below.



2. Type os.system(‘ls -l’) to see a list of files in the current directory. The filename has a space in the name. I can rename the file or updated my program.

How to Resolve the Issue

I decided to rename the file. I did not need to change my code. I can also add “try” and “except” logic to handle this type of error.

Good Code

```
file = open(‘file.txt’, ‘r’)
```



```
print(file.read())
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 5 - RuntimeError
Chapter 5 - OSError (IOError)

Ex 7.15 Error Adding Numbers

Description: TypeError when adding numbers.

Intended Outcome

Print the total when adding two numbers.

Actual Result

When I run the program, the Console Traceback message is a TypeError. The code uses “try” and “except” to provide exception details in the Traceback message. In this example, the code prints a custom message when there is a TypeError. The last line prints a custom message if there is another type of Exception.

Incorrect Code

This is the Example 15 code before any changes.

```
eur = ‘euro’
gbp = 8
```

```
usd = 3.45
try:
     mymoney = eur + usd
     print(mymoney)
except TypeError:
     print('Type error when adding')
except Exception as strmsg:
     print(strmsg)
```

Debugging Steps

Add the variable information to my print statement and run the program again.

```
print('Type error when adding; eur', eur, '; usd', usd)
```

The Console Traceback shows the variable values. After reviewing I realize I used "eur" which is a string, and I meant to use "gbp."

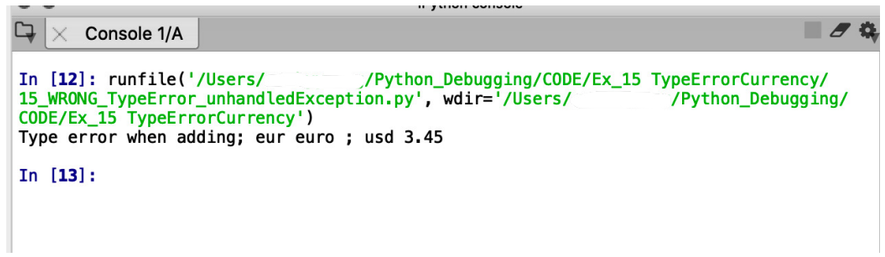


Figure 7.5 Console Exception Message

Note: I could also have used the "**type()**" function to identify the type of variables.

How to Resolve the Issue

The line assigning a value to "mymoney" is updated to use **gbp + usd**.

Good Code

```
eur = 'euro'
gbp = 8
usd = 3.45
try:
     mymoney = gbp + usd
     print(mymoney)
except TypeError:
     print('Type error when adding; gbp', gbp, '; usd', usd)
except Exception as strmsg:
     print(strmsg)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Print Statements
Chapter 4 - Type()
Chapter 5 - Traceback
Chapter 5 - TypeError

Ex 7.16 Misspelled Keyword

Description: A misspelled keyword causes a SyntaxError. This is similar to Example 13 where a variable name was misspelled causing a NameError.

Intended Outcome

An if-else statement prints a message to the Console.

Actual Result

The Editor has a red “x” next to line 4. When I run the program it halts with a “SyntaxError.”

Incorrect Code

This is the Example 16 code before any changes.

```
if 4 < 5:
     pass
esle:
     print(“Python rocks”)
```

Debugging Steps

There is a typographical error on line 3.

How to Resolve the Issue

Update line 3 with the correct spelling of the keyword.

Good Code

```
if 4 < 5:
     pass
else:
     print(“Python rocks”)
```


Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - Traceback
Chapter 5 - NameError
Chapter 5 - SyntaxError

Ex 7.17 Value is None

Description: The function return value is “None”.

Intended Outcome

My calculation using the “mymath” function **return value** prints the result to the Console.

Actual Result

When “i” is 3, the Traceback In the **Console**, displays a **TypeError**.

Incorrect Code

This is the Example 17 code before any changes.

```
def mymath(i=5, j=200):
     if i > 3:
          return i*j
     if i < 3:
          return j/i


result = mymath(3, 100)
```

```
print(result/10)
```

Debugging Steps

This program works as expected when "i" is any number except 3.

1. To identify what type the function returns, I use the type() function in the Console.

```
In[2]: type(mymath(3,100))
Out[2]: NoneType
```

The function returns "None" when "i" is 3.

2. In the Console, I call the "mymath" function again where "i" is 4. Now the type is "int." This means that there is a path through the "mymath" function without a return value.

   In Chapter 3, we looked at a function that did not have a return value for all paths. For this example, I'm not going to change the "mymath" function. Instead, I add logic to my program to handle a "None" value.

How to Resolve the Issue

Add an "if" statement to test for a "None" value.

Good Code

```
def mymath(i=5, j=200):
    if i > 3:
        return i*j
    if i < 3:
        return j/i


result = mymath(3, 100)
if result is not None:
    print(result/10)
```



```
else:
    print("result is None")
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Function Returns None
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type
Chapter 6 - Value is None

Ex 7.18 Method Not Found

Description: AttributeError when the method is not found.

Intended Outcome

Using the math library, I would like to use a cube method.

Actual Result

When I run the code the Console displays the message "AttributeError :module 'math' has no attribute 'cube'."

Incorrect Code

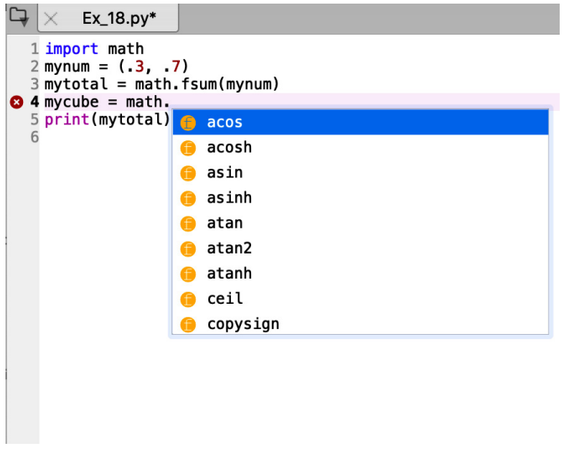
This is the Example 18 code before any changes.

```
import math
mynum = (.3, .7)
mytotal = math.fsum(mynum)
```

```
mycube = math.cube(3)
print(mytotal)
```

Debugging Steps

I need to see what functions or methods are available for the math library. In the Editor, after I type "math." I pause for a moment. A pop-up opens with available methods, as shown below.



In the Console, I could also type dir(math) to see a list of functions/methods in the math library.

How to Resolve the Issue

After scanning the list I see there is no "cube" method available. I update the code to manually calculate the cube of "3."

Good Code

```
import math
```



```
mynum = (.3, .7)
mytotal = math.fsum(mynum)
mycube = 3*3*3
print(mytotal)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 3 - Tuple
Chapter 4 - Debug Mode
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 5 - NameError

Ex 7.19 Module Not Found

Description: ModuleNotFoundError when when the method is not found.

Intended Outcome

Using the "matplotlib" library I want to plot a chart.

Actual Result

When I run the code, the Console displays ModuleNotFoundError.

Incorrect Code

This is the Example 19 code before any changes.

```
import matplotlibpyplot as plot
plt.plot([1, 2, 3, 4], [25, 30, 29, 31])
```

```
plt.ylabel('age')
plt.xlabel('participants')
plt.show()
```

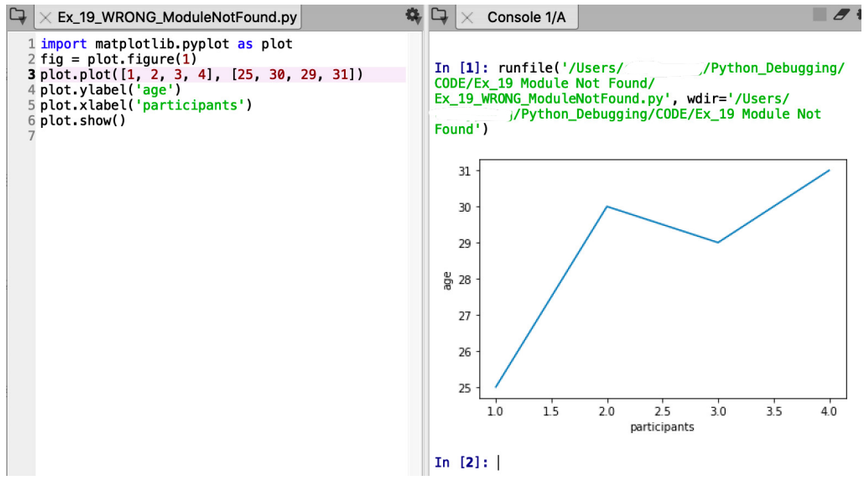
Debugging Steps

I am trying to use the library "matplotlib" plot function, but am unsure how to import the library. When I search the Internet for "import matplotlib" I find the correct syntax. In the help pane, I could search for "mathploglib.pyplot" for additional details.

How to Resolve the Issue

The first line needs updated to import the correct module. There is a missing period. After I run the updated program, In the Console, I can type **help(plot.figure(1))** to see additional information on my new object.

```
import matplotlib.pyplot as plot
```

Good Code

```
import matplotlib.pyplot as plot
plot.plot([1, 2, 3, 4], [25, 30, 29, 31])
plot.ylabel('age')
plot.xlabel('participants')
plot.show()
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - Traceback
Chapter 5 - ModuleNotFound

Ex 7.20 Key Not in Dictionary

Description: Keyerror. The Key is not in the "**wb**" Dictionary.

Intended Outcome

This program works with an Excel file and formats cells. Line 11 loops through the rows, and line 12 loops through the cells of each row. In line 13 I want to set "wrap text" for the cells.

temp.py | 20_KeyError.py*

```
1  # 20_KeyError.py
2  from openpyxl import load_workbook, styles
3  wb = load_workbook('before.xlsx', data_only=True)
4  ws = wb["Sheet1"]
5
6  ft = styles.Fontt(color='4F81BD', bold=True)
7  ws['A1'].font = ft
8  ws.cell(row=1, column=1).value = 'Heading 1'
9  ws.column_dimensions['A'].width = 12
10
11
12 for row in ws.iter_rows():
13     for cell in row:
14         cell.alignment = cell.alignment.copy(wrapText=True)
15
```

Figure 7.6 20_KeyError.py Script

Actual Result

The program halts and the **Traceback** In the **Console,** displays this error:

KeyError: 'Worksheet Sheet1 does not exist.':

Incorrect Code

This is the Example 20 code before any changes.

```
# 20_KeyError.py
from openpyxl import load_workbook, styles
wb = load_workbook('before.xlsx', data_only=True)
ws = wb["Sheet1"]

ft = styles.Fontt(color='4F81BD', bold=True)
ws['A1'].font = ft
ws.cell(row=1, column=1).value = 'Heading 1'
ws.column_dimensions['A'].width = 12

for row in ws.iter_rows():
      for cell in row:
          print("Looping through data")
```

Debugging Experiment

In this Example, when I run the program the program halts. I use Debug Mode to research what is happening.

1. Run the program. The Python Interpreter halts because of an exception.
2. In the **Console**, pane the **Traceback** displays an error, as shown below.

   KeyError: 'Worksheet Sheet1 does not exist.':
3. Type **%debug** In the **Console,** pane to start Debug Mode. The **Console** prompt changes to **ipdb>**.
4. While in Debug Mode, In the **Console**, type "**u**" to step backward through the program. The Python Interpreter moves back to the previous call. Now an arrow indicates the last line of code was **line 4** in my 20_KeyError.py script.

```
ipdb> u
> /20_KeyError.py(4)<module>()
      2 from openpyxl import load_workbook, styles
      3 wb = load_workbook('before.xlsx', data_only=True)
----> 4 ws = wb["Sheet1"]
      5
      6 ft = styles.Fontt(color='4F81BD', bold=True)
```

Line 4 is trying to reference "Sheet1." Although it's not readily apparent, this **KeyError** indicates this is a Dictionary key. In the Console, pane, type "**q**" to quit Debug Mode.

5. Now I need a method to find the **sheetnames** for the "**wb**" object. In the **Console**, I use the help command, as shown below.

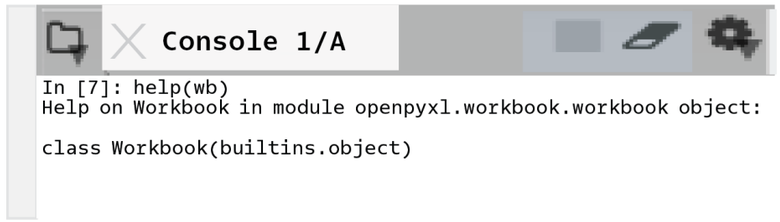


Figure 7.7 Help In the Console, Pane

When I scroll down In the **Console**, pane I see the methods for the "**wb**" object. A small sample of the output is shown below. I am interested in the "sheetnames" method.

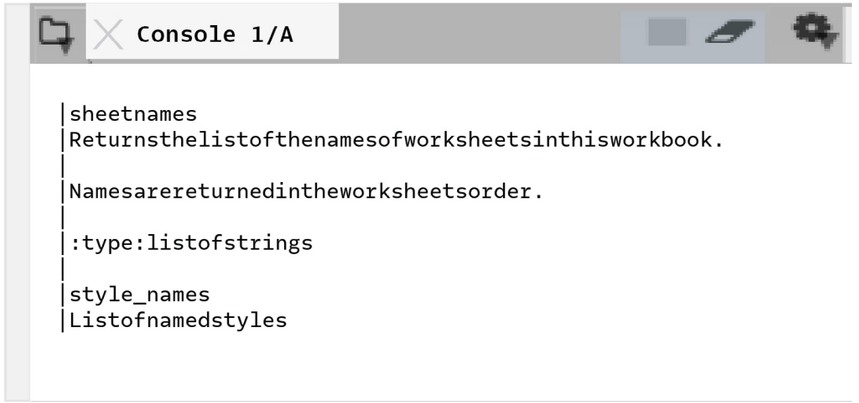


Figure 7.8 Details on "wb" Object Methods and Functions

To see the sheetnames for the "**wb**" object type "**ws.sheetnames**" In the **Console,** pane, as shown below.

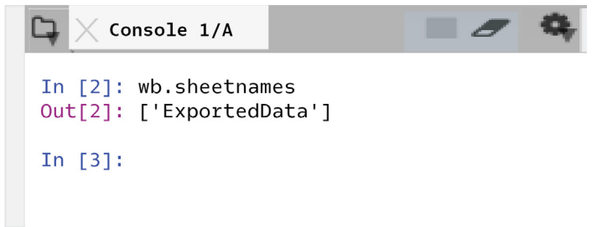


Figure 7.9 The sheetnames Method

The output indicates there is only one worksheet named "**ExportedData**".

How to Resolve the Issue

When you create a Dictionary named "mydictionary," it's easier to recognize a Dictionary KeyError. Because the openpyxl created the "**ws**" object it's not as obvious that this was a Dictionary KeyError. Tuples are immutable, and any immutable object can be used as a Dictionary key. For additional information, search the Intranet for help on openpyxl worksheet objects.

To resolve the error, I update my code to the correct worksheet name, as shown below.

Good Code

In the Editor, I updated line 4 with the sheetname "ExportedData," as shown below.

```
# 20_CORRECT_KeyError.py
from openpyxl import load_workbook, styles
wb = load_workbook('before.xlsx', data_only=True)
ws = wb["ExportedData"]

ft = styles.Fontt(color='4F81BD', bold=True)
ws['A1'].font = ft
ws.cell(row=1, column=1).value = 'Heading 1'
ws.column_dimensions['A'].width = 12

for row in ws.iter_rows():
      for cell in row:
          print("Looping through data")
```

Additional Troubleshooting

In step 4, I wanted more information on "**load_workbook**." The signature function from the Chapter 6 topic, "The Function Call Signature" would be perfect for this purpose. With Python v3.x the "signature" lists each parameter accepted by a function. In the Console, import the module and print the signature for the object, as shown below.


```
In [3]: from inspect import signature
In [4]: print(str(signature(load_workbook)))
(filename, read_only=False, keep_vba=False, data_only=False, keep_
links=True)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Dictionary
Chapter 3 - Tuple
Chapter 4 - Debug Mode
Chapter 4 - Help()
Chapter 5 - Traceback
Chapter 5 - KeyError

Ex 7.21 Incorrect Argument Type

Description: A function argument is an incorrect type causing a ValueError.

Intended Outcome

The program asks for a month as input. The int() function converts the data to an integer so I can use it in a calculation. The print() function outputs the value to the Console.

Actual Result

When the program runs, if I enter a string for input, a ValueError is displayed in the Console. A ValueError is raised when a function gets an argument of correct type but improper value.

Incorrect Code

This is the Example 21 code before any changes.

```
birthmo = int(input('what month were you born?'))
monthstogo = 12 - birthmo
print(monthstogo, "months until your birthday")
```

Debugging Steps

The Traceback shows the program fails on line 1, which means the variable "birthmo" is not created in memory, and is not shown in Variable Explorer.

The Traceback message shows ValueError: invalid literal for int() with base 10: 'June.' The string input 'June' causes a ValueError when the int() function tries to convert the string to an integer.

How to Resolve the Issue

I am going to add "try" and "except" logic to handle the ValueError exception.

Good Code

```
try:
    birthmo = int(input('what month were you born?'))
    monthstogo = 12 - birthmo
    print(monthstogo, "months until your birthday")
except ValueError:
    print('enter a number, no letters')
```


Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 5 - Try and Except
Chapter 5 - ValueError
Chapter 6 - Check Object Type

Ex 7.22 Name Error

Description: When I try to use "plot" there is a NameError.

Intended Outcome

Using the "matplotlib" library I want to plot a chart.

Actual Result

When I run the code, the Console displays a NameError and highlights line 2.

Incorrect Code

This is the Example 22 code before any changes.

```
import matplotlib.pyplot as plot
plt.plot([1, 2, 3, 4], [25, 30, 29, 31])
plt.ylabel('age')
plt.xlabel('participants')
plt.show()
```

Debugging Steps

The NameError indicates the object can't be found. When I imported the library on line 1, I used "plot," and on the other lines I used "plt."

How to Resolve the Issue

Update line 1 to use "plt."

Good Code

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [25, 30, 29, 31])
plt.ylabel('age')
plt.xlabel('participants')
plt.show()
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - NameError

Ex 7.23 Value Error

Description: Invalid data passed to method causing ValueError.

Intended Outcome

I want to remove one item from my list.

Actual Result

When the program runs it halts with a ValueError on line 3. A ValueError is raised when a function or method gets an argument of correct type but improper value.

ValueError: list.remove(x): x not in list

Incorrect Code

This is the Example 23 code before any changes.

```
fruits = ['apple', 'orange', 'grape']
myfruit = 2
fruits.remove(myfruit)
```

Debugging Steps

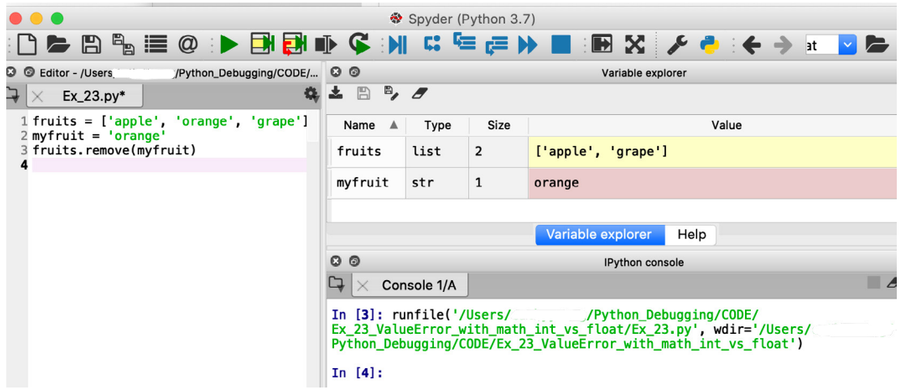
Line 3 uses the "remove" method. I would like to inspect the list object to see what methods are available.

1. In the Console, type **help(fruits)**. The interpreter returns a list of methods available, showing the "remove" uses the value of a list item.

2. In the Editor, I could position my cursor in front of "remove" and press Cntrl + I. The Help pane displays the same information on the remove method.

How to Resolve the Issue

Instead of using the value "2", I update the value of "myfruit" to "orange." When I rerun the program there is no error, and Variable Explorer shows the value "orange" was removed from my list.



Good Code

```
fruits = [‘apple’, ‘orange’, ‘grape’]
myfruit = ‘orange’
fruits.remove(myfruit)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 4 - Help()
Chapter 5 - ValueError
Chapter 5 - RuntimeError

Ex 7.24 Divide by Zero Error

Description: Calculation causes a ZeroDivisionError.

Intended Outcome

The program retrieves the current GBP exchange rate, and then converts “gbp” to the equivalent USD value.

Actual Result

When the program runs, a ZeroDivisionError from line 7 is displayed in the Console

Incorrect Code

This is the Example 24 code before any changes.

```
from bs4 import BeautifulSoup
from urllib.request import urlopen
usd, gbp, gbpex = 10.0, 20.0, 0.00
html2 = urlopen(‘https://usd.fxexchangerate.com’)
soup2 = BeautifulSoup(html2, ‘lxml’)
tables2 = soup2.findChildren(‘td’)
gbp = gbp/gbpex
print(“gbp converted to USD is:”, gbp)
```

Debugging Steps

1. Variable Explorer shows the value of “gbpex” is zero. In this example, I omitted the line to retrieve the GBP exchange rate.

How to Resolve the Issue

In addition to adding the line of code to retrieve the GBP exchange rate from a web page, I added code to handle when “gbpex” causes an exception. In Chapter 4 I also added logging to handle this type of error.

Good Code

```
from bs4 import BeautifulSoup
from urllib.request import urlopen
usd, gbp, gbpex = 10.0, 20.0, 0.00
html2 = urlopen('https://usd.fxexchangerate.com')
soup2 = BeautifulSoup(html2, 'lxml')
tables2 = soup2.findChildren('td')
try:
     gbpex = float(tables2[3].string[:6])
     gbp = gbp/gbpex
     print("gbp converted to USD is:", gbp)
except ZeroDivisionError:
     print('ZeroDivisionError where gbpex is:', gbpex)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Logging
Chapter 5 - Traceback
Chapter 5 - ZeroDivisionError

Ex 7.25 Math Logic Error

Description: There is a logic error in the math calculation.

Intended Outcome

The math calculation should return 10.

Actual Result

The calculation returns 40 instead of 10.

Incorrect Code

This is the Example 25 code before any changes.

```
myval = 60.0/3.0 * 2
print("myval is:", myval)
```

Debugging Steps

To ensure multiplication occurs before the division, I add parentheses to my code.

How to Resolve the Issue

Add parentheses to change the operator precedence.

Good Code

```
myval = 60.0/(3.0 * 2)
print("myval is:", myval)
```

Ex 7.26 ValueError Assigning Date

Description: Operation on incompatible types. ValueError when assigning a datetime object.

Intended Outcome

After creating a datetime object with a date of 12/31/1999, I want to print the value to the Console.

Actual Result

The program halts with a **ValueError** exception. The Traceback indicates the error is on line 3.

Incorrect Code

This is the Example 26 code before any changes.

```
from datetime import datetime
d1 = datetime.strftime(datetime(1999, 13, 31), '%Y-%m-%d')
```

Debugging Steps

At a glance I can see that while my intentions were good, I made a mistake on line 3. It's obvious there is no month "13" and the statement on line 3 is invalid. The ValueError In the **Console**, also states the "month must be in 1..12."

When the cause of the ValueError is not obvious, you could use the Help pane, or search the Internet, to find correct arguments for a function or method.

How to Resolve the Issue

Line 3 needs updated to use "12" for the month instead of "13."

Good Code

```
from datetime import datetime
d1 = datetime.strftime(datetime(1999, 12, 31), '%Y-%m-%d')
```


Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 5 - ValueError
Chapter 6 - Check Arguments
Chapter 6 - Check Function Return Objects

Appendix - URLs

Arguments

The Python design and glossary entries for arguments are shown in this section. Also see Calls, Functions, and Parameters.

The design of keyword only arguments is covered in PEP 3102:

https://www.python.org/dev/peps/pep-3102/

The Python glossary entry for arguments:

https://docs.python.org/3/glossary.html#term-argument

Python terminology on arguments and parameters:

https://docs.python.org/3.3/library/inspect.html#inspect.Parameter

The difference between Arguments and Parameters:

https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter

Assert

The Python reference for assert statements is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

Attributes

The Python glossary entry for "attributes" is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-attribute

Built-in Functions

The Python reference for built-in functions is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/library/functions.html

Calls

The Python reference documentation explains "calling" functions and methods, and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/expressions.html#calls

Classes

Information on Python Classes is available on the **docs.ptyhon.org** website.

9.3.2 Instantiation and Attribute References

9.3.3 Instance Objects, Attributes and Methods

9.3.4 Method Objects

9.9 Container Objects, Elements, and Iterators.

https://docs.python.org/2/tutorial/classes.html

Comparisons

The following link is the Python reference on comparisons.

https://docs.python.org/3/reference/expressions.html#comparisons

Containers

The **docs.ptyhon.org** website explains "containers" in the 9.9 section "Container Objects and Iterators." The section 3.1 topic "Objects, values and types" also explains that objects that contain references to other objects are containers. Examples of containers are tuples, lists and dictionaries.

https://docs.python.org/3.8/reference/datamodel.html#index-3

The 3.1 topic "objects, values, and types" explains that "container" objects contain references to other objects. This "Data Model" reference is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/datamodel.html#index-3

doctest

The **docs.ptyhon.org** website has details on using the doctest module to search and validate examples in docstrings.

https://docs.python.org/3/library/doctest.html?highlight=doctest

Interactive Python examples are also available. These examples include reading in a text file

Functions

The Python reference documentation explains “functions,” and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/compound_stmts.html#function

The Python tutorial for “Defining Functions” is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/tutorial/controlflow.html#defining-functions

https://docs.python.org/3/reference/compound_stmts.html#function-definitions

The difference between function parameters and arguments is explained in the FAQs available on the **docs.ptyhon.org** website.

https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter

The Python glossary entry for “functions” is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-function

Glossary

The official Python glossary is available on the **docs.ptyhon.org** website

https://docs.python.org/3/glossary.html

The if Statement

Information on the **if statement** is available on the **docs.ptyhon.org** website

https://docs.python.org/3/reference/compound_stmts.html#the-if-statement

Immutable

The Python glossary explains the concept of “immutable” objects, and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-immutable

Inspect

The Python reference for the “inspect” library is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/library/inspect.html

Interactive Mode

Interactive Mode in the **Console** is explained on the ipython.readthedocs website.

https://ipython.readthedocs.io/en/stable/interactive/reference.html.

Iterable and Iterations

The Python glossary explains the "iterable" concept, and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-iterable

The **docs.ptyhon.org** website explains Classes in the 9.9 section "Container Objects and Iterators."

https://docs.python.org/2/tutorial/classes.html

Logging

The Python docs for the "logging library" are available on the **docs.ptyhon.org** website.

https://docs.python.org/3/library/logging.html#logging.basicConfig

https://docs.python.org/3.8/howto/logging.html

https://docs.python.org/3/library/logging.html

Magic Functions

Functions that begin with the percent symbol are magic functions or magic commands and are sometimes implemented in a iPython kernel. Read more about magic functions at https://ipython.readthedocs.io/en/stable/interactive/reference.html.

Methods

The Python glossary explains "methods," and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-method

Objects

The Python glossary explains "objects," and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-object

Objects like data attributes have value or "state," and objects like methods have "defined behavior."

The 3.1 topic "objects, values, and types" in the "Data Model" reference is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/datamodel.html#index-3

Parameters

The Python glossary explains "parameters," and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-parameter

There is information on arguments and parameters in the "inspect library."

https://docs.python.org/3.3/library/inspect.html#inspect.Parameter

The difference between Arguments and Parameters is explained in the FAQs.

https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter

The pass Statement

Information on the **pass statement** is available on the **docs.ptyhon.org** website

https://docs.python.org/3/reference/simple_stmts.html#the-pass-statement

The return Statement

Information on the **return statement** is available on the **docs.ptyhon.org** website

https://docs.python.org/3/reference/simple_stmts.html#the-return-statement

State

The Python glossary explains the "state" of data attributes, or objects with value.

https://docs.python.org/3/glossary.html#term-object

Statements

The Python glossary explains "statements," and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-statement

https://docs.python.org/3/reference/simple_stmts.html

timeit

Information on the timeit() function is available on the **docs.ptyhon.org** website

https://docs.python.org/3/library/timeit.html

The try Statement

Information on the **try statement** is available on the **docs.ptyhon.org** website

https://docs.python.org/3/reference/compound_stmts.html#the-try-statement

Types

The Python glossary explains "types," and is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/glossary.html#term-type

The 3.1 topic "objects, values, and types" in the "Data Model" reference is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/datamodel.html#index-3

Values

The 3.1 topic "objects, values, and types" in the "Data Model" reference is available on the **docs.ptyhon.org** website.

https://docs.python.org/3/reference/datamodel.html#index-3

Conclusion

Einstein said, "If you can't explain it simply, you don't understand it well enough." Learning new things is a passion of mine, and I've found the process of organizing notes, creating illustrations, and pondering how to craft clear examples helps me grasp concepts. Then too, it's nice to go back in a year when I've forgotten something and refer to a solid example.

Thank you for reading along with me through the interesting topics and less than thrilling subjects. If the result is you have mastered new features, it was worth it! I'd love to hear the cool things you're doing with Python, so please don't hesitate to leave comments in a review.

Index

K

L

M

N

O

Index

Index