Python Coding

One Year Later

Visual Programming Series

RL Zimmerman

Python Coding -

One Year Later

A treasure trove of practical and simple examples.

RL Zimmerman

Python Coding -

One Year Later

Copyright © January, 2021 by H. Zimmerman

Educators at primary, secondary, and post-secondary schools are encouraged to contact hzimmerman4741@att.net for written permission to freely use this publication for educational purposes and receive original diagrams and PDFs.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any other means, electronic, mechanical, photocopying, recording, scanning or otherwise by any party except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act without the prior written permission of the publisher and copyright owner.

Trademarks

All terms mentioned in this book that are known to be trademarks have been capitalized. All other trademarks are the property of their respective owners.

References & Citations

“Welcome to Python.org.” Python.org, www.python.org /.

IDE used in this book is from www.Anaconda.com

Warning and Disclaimer

The information provided in this book is on an “as is” basis. The author is not responsible or liable to any person or entity with respect to any loss or damages arising from the information contained in this book. The author makes no representations or warranties, express

or implied, as to the condition, quality, merchantability or fitness of any product mentioned.

Table of Contents

1. Introduction

1.1 Overview

1.2 What This Book is About

1.3 What's Next?

2. Debugging Overview

2.1 Plan for Debugging

- Start Small
- Keep Multiple Versions of Your Code
- Intended Outcome
- Computational Complexity
- Test Objects
- Test Data Files
- Plan for Tomorrow

2.2 Debugging Steps

- Logbook
- Divide and Conquer
- Backup Files Before Debugging
- Problem Statement

Doubt Everything
Look Around Your Environment
Create a List of Suspects
What Do You Think is the Cause?
Refine Your Experiment
Experiment
Success at Last

2.3 The Debugging Environment

Python
Anaconda
Spyder
Run a Script or Program
Stop a Program or Restart the Kernel

2.4 Help

2.5 What's Next?

3. Python Basics

3.1 Python Syntax

Referencing Object Values
Variables in Imported Modules

3.2 Objects

3.3 Immutable Objects

3.4 Variables

Global Variables
Unpacking

Ignore or Throw Away Variables

3.5 Expressions

Comments
Joining Lines
Escape Sequence

3.6 Types of Data

What is the Data Type?
Converting Data Types
Converting Floats to Ints
Converting Strings to Ints
The ‘NoneType’ Value
NAN

3.7 Numbers

Floating Point Numbers

3.8 Data Structures

3.9 Strings

String Methods
Split String
The String Module
Create a String of Lowercase letters
Create a String of Numbers 0-9
Whitespace Characters
Iterate (Loop) Through Strings

3.10 Lists

Update an Element in a List
Iterate Through Items in a List

Copy or Clone a List
List Comprehension
Remove Characters When Converting
Delete an Item from a List
Change a List to a String
Change a String to a List

3.11 Methods for Lists

Append an Item to a List
Extend a List, or Combine 2 Lists
Index: Location of a List Element
Insert an Item to a List
Join List Elements into a String
Add Character When Joining Strings
Pop (Remove) an Element from a List
Remove an Element from a List
Remove Duplicate Elements from a List
The sort() Method
The sorted() Function
Sort Order and the Key Parameter
Sorting by the Second Element in the List

3.12 Tuple

Tuples are Immutable
Slicing a Tuple
Iterate Through Items in a Tuple
Tuples and Function Return Objects
Repetition and Concatenation
Related Tuples
NamedTuple
The Zip Function Returns Tuples

3.13 Dictionary

Elements in a Dictionary
Create a Dictionary
Append to a Dictionary List Value
Copy a Dictionary
How Many Elements are in the Dictionary List?
Assign a Dictionary Value Using the Key Name
Update a “List” Value in a Dictionary
Find the Value of a Dictionary Item
Find the Type of a Dictionary Element
Add a New Key Pair to an Existing Dictionary
Delete a Key in an Existing Dictionary
Iterate Through Dictionary Key Pair Values
Iterate Through Keys in a Dictionary
Retrieve Keys
Search for a Key Name
Test if Key is in the Dictionary
Title and Value Methods
Combine Dictionaries
Dictionary Comprehensions

3.14 Range

3.15 Generators

3.16 Set

Union
Intersection
Difference
Symmetric Difference

3.17 collections

namedtuples
Counter
defaultdict

3.18 Indexes

Indexing Elements in Nested Lists
Indexing Lists & Tuples in Dictionaries

3.19 Slicing

3.20 Operators

Numerical Operators
Modulo Operator
Integer Division
Concatenation, Repetition and Sequence Operations
Comparison Operators
The Union Operator for Sets
Find a Substring
Find Last Element in a String
Comparing Floats
Comparisons that Return True or False

3.21 Identifiers

Interactive Interpreter
Class-Private Names
System-Defined Names

3.22 Control Statements

For Loop
Loop Until Break
Iteration Variable

3.23 Indented Code (a Suite)

3.24 Functions and Methods

Defining a Function
Calling or Invoking a Function or Method
Parameters
Arguments
Keyword Arguments
Positional Arguments
Unpacking Operator Arguments
Unpacking Operators
How to View the Function Argument Definition
Function Return or Yield Objects
Boolean Return Object
Return the Statement that is True
All Paths Do Not Have a Return Value
More than One Return Object
The Type of Return Value
Recursive Functions
Recursive Memory Stacks
The Zip Function
The Map() Function
Lambda Functions
The Filter() Function
The Print() Function

3.25 Classes

Special Methods and Override Behavior
Create a Class
The DocString
Variables - Attributes
Instance Variables and Class Variables

Create an Instance of the Class
Methods
Dotted Notation for Attributes
Calling a Method
Superclass and Subclass

3.26 Modules and Libraries

Variables in Imported Modules

3.27 Attributes

3.28 Scope, Namespace & Memory

A Function that Accesses a Global Variable
A Function Variable with the Same Name as a Global Variable
Scope in Nested Functions

4. Debugging Tools

4.1 Debugging Overview

4.2 Print Statements

Indenting Loop Print Statements

4.3 Overview of the Editor

4.4 Debug Mode

End Debug Mode

4.5 Variable Explorer

4.6 Example: My Program Loops & Never Ends

4.7 Debug Commands

4.8 Console Interactive Mode

- Increment Counters in the Console
- Watch Out for Changing Values
- iPython Session

4.9 Variables and Objects in Memory

4.10 Introspection

- Using ? in the Console
- dir()
- dir(object)
- help()
- The Inspect Library
- What Version of Python?
- The type() Function
- The id() Function
- The repr() Function
- The len() Function
- The locals() Function
- The globals() Function
- The docstring

4.11 Logging

4.12 The timeit() Function

4.13 Logging Time and Loop Counters

4.14 Focused Testing

Actual Result
Incorrect Code

4.15 Create Test Data

Test Objects

5. Exceptions

5.1 Kinds of Errors

Syntax
Logic or Semantic
Runtime

5.2 The Stack Trace or Traceback

Don’t Be Fooled

5.3 Try and Except

5.4 Raise

5.5 Assert

5.6 Built-in Error Types

ArithmeticError
AssertionError
AttributeError
EOFError
FloatingPointError
ImportError
Indentation Error

IndexError
IOError
KeyError
KeyboardInterrupt
LookupError
MemoryError
ModuleNotFoundError
NameError
OSError
OverflowError
RecursionError
RuntimeError
StopIteration
SyntaxError
TabError
SystemError
SystemExit
TypeError
Tuple Object does not Support Item Assignment
String Indices Must be Integers
UnboundLocalError
ValueError
ZeroDivisionError

6. Try This

6.1 What is the Object Value?

6.2 String and Number Variable Values

Print the Value of a String Variable
Variables in Imported Modules

Inspect a Number Variable in Debug Mode
Inspect a String Value with Interactive Mode

6.3 Tuple Objects and Values

Print All Tuple Item Values
Print a Tuple Item Value
Inspect All Tuple Items in Interactive Mode
Inspect A Tuple Item in Interactive Mode

6.4 List Objects and Values

Print All List Item Values
Print the Value of a List Item
Inspect a List Item in Debug Mode
Inspect All Items of a List in the Console
Inspect a List Item in the Console

6.5 Dictionary Objects and Values

Print the Value of a Dictionary Key-Pair
Inspect All Dictionary Items in the Console
Inspect a Dictionary Item Value in the Console
Inspect a Dictionary Item in Variable Explorer

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

Whitepsace Characters

6.7 What is the Object Type?

6.8 What is the Length of the Object?

6.9 What are the Function Arguments?

The Function Call Signature
Inspect the Docstring

6.10 What Type of Object Does a Function Return?

7. Examples

7.1 List Index Out of Range

7.2 Index Error

7.3 Wrong Variable

7.4 Invalid Assignment

7.5 While Indentation Error

7.6 Incorrect Method Arguments

7.7 Empty Block of Code

7.8 Parentheses Not Matched

7.9 Missing Colon

7.10 Case Sensitive

7.11 Missing Keyword

7.12 Illegal Character

7.13 Undefined Name
7.14 FileNotFound
7.15 Error Adding Numbers
7.16 Misspelled Keyword
7.17 Value is None
7.18 Method Not Found
7.19 Module Not Found
7.20 Key Not in Dictionary
7.21 Incorrect Argument Type
7.22 Name Error
7.23 Value Error
7.24 Divide by Zero Error
7.25 Math Logic Error
7.26 ValueError Assigning Date
7.27 Matching Strings NoneType Error
7.28 Matching Strings Fails
7.29 Whitespace or Special Characters

7.30 Debug: Step Through Your Function
7.31 Key Not in Dictionary
7.32 Error Combining Strings
7.33 Function Returns 2 Values
7.34 Unsupported Operand
7.35 Code Goes Beyond Last Yield Statement
7.36 Missing positional argument
7.37 Reserved Keyword
7.38 Dot Instead of Underscore
7.39 Key error in new dictionary key:pair
7.40 Assign a dictionary key:value
7.41 Too many values to unpack
7.42 Tuple Assignment Error
7.43 ‘str’ object is not callable
7.44 Can only Concatenate Tuple
7.45 Float Comparison

7.46 Unhashable type:dict
7.47 builtin_function_or_method' object is not subscriptable
7.48 String Comparison Error
7.49 Invalid literal for int() with base 10
7.50 Variable Referenced Before Assignment
7.51 Plot: Shape Mismatch
7.52 Unpacking Operator

Conclusion

Appendix - URLs

Arguments
Assert
Attributes
Built-in Functions
Calls
Classes

Comparisons
Containers
doctest
Functions
Glossary
The if Statement
Immutable
Inspect
Interactive Mode
Iterable and Iterations
Logging
Magic Functions
Methods
Objects
Parameters
The pass Statement
The return Statement
State

Statements
timeit
The try Statement
Types
Values
Appendix - Errors
Appendix - Libraries
Index

1. Introduction

The primary goal of this book is to learn the simple basics of debugging a Python script. Debugging is the process of finding and removing “bugs” or defects in a program. Debugging is also useful in getting your code to run in the first place! Chapter 3 presents “Python Basics” because a big part of debugging is knowing the correct syntax for a particular object and task. This book doesn’t try to cover all the nuances of Python but does cover the terms and syntax you’re likely to encounter in your first few weeks or months of programming. The overhead and clutter is gone, leaving behind clear and simple instructions.

Code examples are self-contained so you can copy and paste them into your IDE and run the program. There are certainly more elegant ways to do many of these tasks, but I wanted to demonstrate each concept with a working piece of code limited to a few lines. In Chapter 3 we’ll look at the following topics and more.

- Functions, Lists, Dictionaries, Tuples, Ranges, Comprehensions
- Indexing, Slicing, Comparison Operators, Control Statements

When looking at concepts such as indexing, slicing, scope, or recursive functions, I’ve included lots of diagrams. We’ll look at syntax relevant to each type of object. For example, to add to a “list,” you could use ‘append,’ ‘join’ or ‘extend’ methods. While adding a new key:value pair to a dictionary is straightforward, adding to an existing dictionary is not. In this scenario, it’s not about the value of the object in the key:value pair, but more about the “type” of object in the key:value pair. Once you know the type of the dictionary object,

you'll use the corresponding methods to add or change values, whether it's a "list," "string," "tuple," or even another dictionary.

There are dozens of code examples, and I tried to use real-world examples. Between the detailed Table of Contents and extensive Index, I hope you can quickly find what you're looking for. The Index includes common phrases for those times when you don't know the technical term. For example, to find a string in Python you could use the "in" comparison operator. Whether you look for "find," "search" or "locate," the Index refers you to the "in" comparison operator. The examples are varied so you can work with different types of data, such as:

xml

html

matplotlib (plots/charts/graphs)

OS (files/directories)

*.txt, *.csv

user input

url

datetime

To help my daughter with her first Python class, I looked around for debugging information 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 was missing. 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 has "References" to the related topics covered in earlier chapters. So theoretically, you could jump right into the examples in Chapter 7.

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

Plan for Debugging
Debugging Steps
The Debugging Environment
Python
Anaconda & Spyder
Help
What's Next?

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?

Computational Complexity

While writing code, strive for efficient programs. Sometimes my program runs as expected, except that it takes too long to complete because it's inefficient. Chapter 4 looks at several ways to identify

timing bottlenecks. Let's look at three simple ways to improve run time.

- Working with a *.txt file I could import the file each time which is straightforward, but it's more efficient to import the file into a dictionary and reuse the dictionary.
- It's more efficient to "update" a list than make a copy of the list; this is especially true if you have a large list of potentially millions of elements.
- When looking up data in a list several times, use a sorted list. However, if I only use the data one time, the cost of sorting the list just adds overhead to my program.

When discussing the relationship of running time to the size of data, programmers use asymptotic notation. "Big O" is a common form of asymptotic notation to refer to algorithmic complexity. Evaluate programs looking at these scenarios.

- Best-case
- Expected-case
- Worst-case

Count each "operation," taking into account loops and nested loops.

- Comparisons
- Mathematical operations
- Assignments
- Accessing objects in memory

Constant running time, **O(1)** , is more efficient than other operations and does not change based on the size of data input. One example of constant complexity is a simple mathematical opeartion like **2.02 * 3.03** . There may be loops or recursive calls in the code, but the data input size does not affect constant running time.

Logarithmic running time, **O(log n)** , increases based on the size of the data. A bisection search that divides itself in half is an example of logarithmic complexity.

Linear complexity, **O(n)** , is based on each element “n.” For example, printing a list is based on the length of the list.

With this brief introduction to computational complexity, you may consider an in-depth exploration of the subject

Test Objects

To simplify testing, I’ll often break my functions into external *.py files and later combine them as needed. I write a small “test script” that creates variables and invokes the function. For example, a function **menu()** takes **mydictionary** as an argument - or menu(**mydictionary**).

For testing purposes, I only need **mydictionary** with a few elements. To create a dictionary for testing purposes, first, I run the main program that creates **mydictionary** . Then, in the Console, I type **mydictionary** . The Console prints out all the values, so I can copy a smaller subset of data for a test version of **mydictionary** . In my test script, I use the data in an assignment statement for the “test” dictionary.

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, as you open an Excel file, the app prompts you to “fix” data.

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 debugging style 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 initial defect is that 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. 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 skipping 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 on 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 the first day of the month?
   - Is there a 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 webserver responding to requests? Is one of the integrated systems down?

4. If the program is OK on your machine, and doesn't work on a user's machine, look at what is different. Could it be a timing error caused by the user's slower machine? Is the user working remotely or running other conflicting applications?
5. 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
- Does the program produce the desired outcome?
- Does the program work except on the first day of the month?
- Does the program take too long to run? (Look at the algorithmic complexity of your code).
- 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 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 eliminate these items temporarily 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 develop 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 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.”

A Spyder layout with three panes, is shown below. You can return to the default 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 Spyder

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 code block.

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

Python Syntax
Objects
Immutable Objects
Variables
Expressions
Types of Data
Numbers
Data Structures
Strings
Lists
Methods for Lists
Tuple
Dictionary
Range
Generators
Set
Collections
Indexes
Slicing
Operators
Identifiers
Control Statements
Indented Code (a Suite)
Functions & Methods

Classes
Modules & Libraries
Attributes
Scope, Namespace & Memory

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 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 accommodate programmers who are unfamiliar 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 # hash character 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. Example 7.16 has a misspelled keyword. Example 7.43 uses the reserved keyword “str” incorrectly. A missing keyword in a function call also uses a SyntaxError, as shown in example 7.37. Functions also have keyword arguments .
- An empty Suite (indented block of code) is illegal. See “Indented Code (Suite) ” or Example 7.7 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, 8, 9, 10, 11, 12, and 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

Chapters 3 and 6 walk you through examples with the syntax for the various objects, including items inside of data structures or objects from imported modules or libraries.

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)
```

3.2 Objects

The building blocks of Python are objects . Objects have an identity, type, and value.

- Data with state
- Defined behavior (methods)

State refers to the properties of an object, the attributes or value of the object. The object’s behavior is how the Python Interpreter

interacts with that type of object.

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).

```
from openpyxl import load_workbook
wb2 = load_workbook (aftfile, data_only =
True)
ws2 = wb2[“ExportedData ”]
```

3.3 Immutable Objects

In Python strings, numbers, and “tuple” types are immutable , meaning the values are fixed and can’t change. While you can **not** change an existing strings, numbers (integers or floats) or tuples, you can create a new object with changed data to replace an object.

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]: bfr = ‘Hello’

In [2]: print(**id** (bfr))

Out [1]: 12345678

In [3]: bfr = bfr + 1

In [4]: 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 hashable , and this improves code performance. Another advantage of immutable

objects is understandability, and knowing the object will never change.

The Python definition of "hashable " is an object that "has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() method)." We'll look at these special methods at the end of this chapter.

__hash__()

__eq__()

3.4 Variables

In Python, a variable name refers to an object. An object is a place in memory that has a **value** such as a letter or number. An **assignment statement** or "binding" creates a variable and binds or associates the name with an object. The object is a place in memory. Examples 7.4, 7.41, and 7.50 demonstrate assignment statements.

The left side of an assignment statement must be a variable name, and the right side is a value. The assignment operator is the equals **=** sign.

mynumber = 2000000

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

Indirection is using a name to refer to an object. When you add "mynumber" to a list named "mylist," there are two levels of indirection. When a program runs, the **variable** might be assigned a different value; meaning the variable name might be assigned to a different object in memory , or the object itself might be updated.

Global Variable s

When you have a variable with a different value than you expected, it may be due to scope. When you run the program, the Python Interpreter creates variables and adds them to the “global scope ” or the first memory “stack.” In Python, you can read, but not change, the value of a global variable at any point in your program or from within functions, as long as everything is within the **same *.py file** . We look at scope at the end of this chapter in the topic, “Scope , Namespace, and Memory.”

Each time the code moves into a “function,” a new “local” scope is created. Within the local scope of a function, you can’t **change** the values of objects in the outer global scope. You can only change values within the function’s local scope. When you create objects inside a function or method, those objects or variables are typically not available outside the scope of that function. To change a global variable within a function, use the keyword “global” to make the global variable part of the “local” scope.

A word of warning about global variables, they are dangerous! It’s easy to lose track of which function is updating a global variable. To avoid global variables with recursive function calls, you can pass the data through an optional function parameter.

In the next diagram, the program has “var1” in the main body of the program (lines 5-7) and “var2” (on line 2) within myfunction().

To explore global variables, run this code in debug mode and step through the code, running one line at a time.

1. The Python Interpreter evaluates the function definition on line 1 and quickly moves to line 5. If myfunction() had optional arguments with default values, they are assigned using the “scope” that exists at line 1 (the global scope.)

If there had been an error within the function, the interpreter would stop and raise the error.

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
2 myfunction():
3     var2 = 4
4     print (var2)
5
6 var1 = 7
7 myfunction()
  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.

```
  def
1 myfunction():
2     print (var1)
3
4     var2 = 4
5     print (var2)
6
7 var1 = 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 the UnboundLocalError shown below. The UnboundLocalError is raised because "var1" does not exist within the local scope of "myfunction()."

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

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

```
7 var1 = 7
8 myfunction()
  print(var1)
```

To prevent the UnboundLocalerror use the keyword “global” to declare “var1” a global variable inside myfunction() on line 2. Notice in the next example there is no “**assignment** ” value on line2. When line 3 prints “var1,” it has a value of “7” from the “var1” in the main program or global namespace. Line 5 creates a new var1 within the local scope of myfunction() with a value of “9.” When line 9 prints “var1,” the value is now 9.

You can read more about scope and namespace at the end of this chapter.

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

In a bit, we’ll look at using a function argument with a default value as another way to implement the concept of a “global variable.” There is an example of this process in the “Function” topic “optional arguments ” later in this chapter, as well as the topic recursive functions .

Unpacking

When you **unpack** a tuple or list you assign individual elements to a new variable. We’ll look at unpacking later in this Chapter in the topic, “Functions, Unpacking Operator .

In [1]: mytuple = (1, 2, 3, 4)

In [2]: myvar1, myvar2, myvar3, myvar4 = mytuple

Ignore or Throw Away Variables

Sometimes you'll see code where the underscore character is used when the programmer wants to ignore an element in a tuple or list. These are often called "throw away" variables. In the second statement below, the "2" element is assigned to the underscore __ In the third statement the variable "dummy" is used for clarity.

```
a, b, c, d = (1, 2, 3, 4)
a, _, c, d = (1, 2, 3, 4)
a, dummy, c, d = (1, 2, 3, 4)
```

The PEP 3132 "Extended Iterable Unpacking" specifies a "catch-all" name which is assigned to a list of all items not assigned to a "regular" name.

```
a, *b = (1, 2, 3, 4)
a = 1
b = (2, 3, 4)
```

3.5 Expressions

The actions that a program takes are referred to as "control statements" or "expressions ." Objects and operators are combined to form expressions.

<object><operator><object>

myvar = 7

We'll look at comparison operators, control statements, and loops later in this Chapter.

Comments

Code comments begin with the hash # symbol.

Joining Lines

Python uses explicit line joining . Expressions that span more than one physical line are joined into logical lines with a backslash.

```
print(‘ the swift fox jumped over the \
      lazy dog and then ran into the briar
      patch.’
```

Python also implements implicit line joining . Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line without using backslashes.

Escape Sequence

Special characters in strings are identified with an “escape sequence ” or backslash. For example, the line feed character is ‘\n’ within a string. To ignore the escape sequence, use the ‘ r’ prefix.

```
r‘c:\users
’
```

In the previous example, if you omit the prefix ‘r,’ a SyntaxError is raised.

SyntaxError: (unicode error) ‘unicodeescape’ codec can’t decode bytes in position 2-3: truncated \uXXXX escape

Apostrophes in strings also use escape sequences, with a backslash as shown below.

```
mystr =
‘\”’
```

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, tuple, 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 below 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.”

```
print (type( mystr ) )
print (isinstance (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 .” Example 7.21 looks at a function with the wrong argument type.

Converting Data Types

When working with data, you may need to change or convert the data type. In other programming languages this is called casting. For example, during a calculation, you may want to convert between a **float** and an **int** to remove decimal places.

int(my_var)
str(my_var)
float(my_var)
bool(my_var)

Converting Floats to Ints

Notice in this example the value “45.9” is converted to “45.”

```
myfloat = 45.9
int(myfloat)
```

The next statement rounds the float number up to the whole number “46.”

```
myfloat = 45.9
round(myfloat, 0)
```

Converting Strings to Ints

When concatenating numeric values and strings, the statement str(**my_int**) converts the string to an integer. While I can’t convert the string ‘1.25’ into a whole number **int** , it is legal to convert a float to an int, as shown in Example 7.49.

The ‘ NoneType’ Value

In Python, the absence of a value is called “None,” which is 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 7.17. An “if statement” that tests for a value of “None” is shown below. Example 7.27 demonstrates matching strings and taking into account a NoneType value. The Chapter 6 topic, Does the Object have a Value of NoneType or Whitespace? looks at NoneType and whiteSpace.

```
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** .”

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

NAN

A NAN is a special floating-point value that can’t be converted. NAN stands for “not a number.” The “math” function math.**isnan(x)** returns true if “x” is not a number.

3.7 Numbers

Floats and integer types represent numbers in Python and are scalar objects that have no internal data structure. When assigning integer values to variables, 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, in Python 3.6 and later, you can add

underscores as a separator .

mynumber = 2_000_000

Floating Point Numbers

Non-integer numbers or floats are stored in computer memory as a binary representation of 0's and 1's. Calculations can introduce subtle differences where you may think both float values are 1.08, but the actual binary representation is slightly different.

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

Example 7.45 demonstrates float comparisons, and the "comparison " topic later in this chapter has details on comparing 'floating point' numbers.

3.8 Data Structures

Python has several built-in compound data structures or sequence types for non-scalar objects. Non-scalar objects have an internal structure. A list, tuple, or range is a sequence type. A string is a **text sequence type** . These data structures or "containers" have an ordered sequence of **elements** or items. 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. The **docs.python.org** site refers to these objects as "iterables ." Iterables are objects with a sequence of elements referenced by an index (as outlined below.)

- Lists
- Tuples
- Strings
- Range

While there are other data types in Python, the last common one we'll look at in this chapter is a dictionary. A dictionary is a "mapping type" of data.

- Dictionary

Python uses 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. 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.

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

You use an index to iterate through these containers to access the value of a particular element in a container. A list, tuple, or string also uses these operations:

- Indexing
- Slicing

With the exception of a range, you can use concatenation and multiplication on data structures. For example, to concatenate 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 quotation **“”** marks, as shown below.

b = ‘bookstore’

If you forget the closing apostrophe in your assignment statement, a SyntaxError is raised for “EOL while scanning string literal,” as shown in Example 46.

Strings can be concatenated, indexed, and sliced. In the previous example, the index for the letter ‘s’ is b[4] because Python 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, in 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 ensure an object is of type “string,” and account for uppercase and lowercase letters. 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 with an escape sequence , like the line feed '**\n** ' character. The "string module" also has a function to create a list of whitespace characters. The function **repr()** displays a printable representation of a string including whitespace and is demonstrated in example 7.28

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

String Methods

Let's take a moment to look at some of the common string methods first. In a bit we'll also look the "String Module." To see all the string methods available for your version of Python, checkout the docs.python.org. In the top left corner you can select your language and version, and then on the right side of the page under "Text Sequence Type -- String, click on "String Methods ." After creating a string variable "mystr," in the **Console** , type "**dir(mystr)** " to see additional information.

Syntax	Comments
mystr.isalpha()	Returns True if all characters in the string are alphabetic (a-z) and there is at least one character.
mystr.capitalize()	Converts to Camel Case
mystr.count('39')	How often is ' 39' in "mystr"
mystr.find('39')	Index of first occurrence of ' 39'
mystr.index('9')	Returns the index for ' 9' , or returns error if not found
mystr.isnumeric()	Returns True if all characters are number (0-9) and there is at least one character.
mystr.join(mytuple)	Creates a new string by joining an iterable (tuple, list, set, dictionary.) Elements are separated by the "mystr" value.
mystr.lower()	Change to lowercase
mystr.lstrip()	Remove whitespace on the left
mystr.rindex('9')	Same as index but counting from right
mystr.replace(old, new)	Return a copy of the string with old substring replaced by new.
mylist.rsplit(**sep='a'**)	Returns list of words in the string, using **sep** as the delimiter string. Similar to the idea of tokens in C programming.
mylist.rstrip()	Remove whitespace on the right
mylist.split(**sep='a'**)	Returns a list of words in the string, using **sep** as the delimiter string.
mystr.strip()	Return a copy of the string with the leading and trailing characters removed.
mystr.swapcase()	Return a copy of the string with uppercase characters converted to lowercase and vice versa.
mystr.upper()	Change to uppercase
mystr.zfill(**width**)	Return a copy of the string left filled with ' 0' digits to make the string length **width** . Think of zfill as padding numbers.

Table 3.2 String Methods

An example using the find() method with a string is shown in the “Find a Substring ” topic that follows. The topic “Slicing ” also demonstrates retrieving part of a string.

Split String

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

```
mystr = ‘ hello worl
d’
mystr. split ()
```

And the Console shows:

In [2]: mystr
[‘hello’, ‘world’]

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 3.3 Some String Module Methods

The **string.ascii_letters** is a simple way to build a list of letters. On line 1 I import the string module. The list() function converts the string to a list, as shown below.

```
import string
1 alphabet_string = string.
2 ascii_letters
3 alphabet_list = list
4 (alphabet_string)
5 print (alphabet_list)
```

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
3 all_ltrs = string.
4 ascii_lowercase
5 print (all_ltrs)
```

```
all_ltrs_list = list (all_ltrs)
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’]

Create a String of Numbers 0-9

To create a string of numbers 0-9 use the syntax shown below.

```
import string
str_Numbers =
string.digits
print (str_Numbers)
```

And the Console shows:

In [3]:

0123456789

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.

```
import string
all_ltrs =
string.whitespace
print (repr (all_ltrs))
```

And the Console shows:

In [3]:

\t\n\r\x0b\x0c

In the Chapter 6 topic, “ Does the Object have a Value of NoneType or Whitespace? ” we look at functions to remove whitespace. Example 7.29 demonstrates whitespace errors.

Iterate (Loop) Through Strings

String indices must be integers. The next example of a “for” loop is perfectly legal for the loop expression, and prints the message, ‘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’ used in the loop are not integers.

```
mystr = ‘abc’
for i in mystr :
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.

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

3.10 Lists

A **list** is an ordered collection of values or 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 **[ ]** . An 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** . Examples 7.1 and 7.2 demonstrate an **IndexError** when the list assignment is out of range.

When creating a List, use square 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
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 3.4 List Objects

Update an Element in a List

Use the index to update a particular element in a list. This example updates the second element in the list. See the topic that follows, “Index: Location of a List Element,” to locate a particular index.

```
mylist2[1 ] = 8
```

Iterate Through Items in a List

A “for loop” is one option to iterate through items in a list, as shown below.

```
for j in mylist:

```

```
print (' mylist item is: ' ,
mylist[j])
```

These Chapter 7 examples illustrate a few list errors: example 7.1, 7.2, and 7.3.

Copy or Clone a List

To make a shallow copy of a list, add **[:]** at the end of the expression. A shallow copy inserts references to objects found in the original list and is one level deep.

```
mylist2 = mylist1[:]
```

If you leave off the **[:]** Python creates an “alias” or “reference,” so that mylist2 points to the “mylist1” object in memory. This behavior is referred to as “**indirection** .” With an alias, no matter what the name of the variable, you are actually using “mylist1.”

You can also use this expression to copy a list.

```
mylist2 = list (mylist1)
```

A shallow copy is one level deep, so **mylist1** with values [1, 2, 3, 4] would create a copy mylist2 with values [1, 2, 3, 4]. I can add an object to mylist4 and the additional element is only seen in mylist4. Let’s look at what happens when we **update** a value in the either list after a shallow copy.

In this example, **mylist3 = [[1, 2, 3, 4], [5, 6, 7, 8]** and I created a shallow copy **mylist4** . At this point **mylist4 = [[1, 2, 3, 4], [5, 6, 7, 8].** When I update an element in either list, the corresponding element in the other list is also updated. If I change **mylist3[0][0] = 3** , the change is also seen in **mylist4 = [[3, 2, 3, 4], [5, 6, 7, 8].**

To avoid this shallow copy behavior, import the “copy” library to use **copy.deepcopy()** to create a new list and insert copies of all objects in the original list. The new list is independent of the original list.

List Comprehension

L ist comprehension is an elegant way to create a new list from a comprehension. The only difference between a list comprehension and map() is Python returns a list instead of a map object. The comprehension is an "**expression** " followed by "**for** " loop(s) to iterate over elements. The comprehension may also include conditional "**if** " statement(s).

newlist = **[expression - for - if]**

Let's take the three statements that follow and combine them in a list comprehension. This comprehension computes sales tax for only elements of type "float" in the "sales_list," and creates a new "receipt_list."

```
round (i * 1.065 , 2 )
for i in sales_list
if type (i) == float
```

1. The slash at the end of line 2 below means the list comprehension continues on line 3. The Python Interpreter considers lines 2 and 3 one statement.

On line 2, the first part of the list comprehension computes the price of items with sales tax, and rounds the result to two decimal places.

2. The second statement loops over items in the "sales_list".
3. Finally, the third statement on line 3 checks that elements are a float.

```
1 sales_list = [1 , 'iPad' , 399.99 , 2 , ' charging cabl e' , 29.9 9 ]
2 receipt_list = [ round (i * 1.065 , 2 ) for i in sales_list \
3         if type (i) == float ]
4 print(receipt_list)
```

And the Console shows:

```
In [2 ]: receipt_list
Out [2 ]: [425.99 , 31.9 4 ]
```

This comprehension creates a list of vowels from a string.

```
mystr = ‘ charging cabl e’
newList = [v for v in mystr if v in ‘aeiou’
]
```

And the Console shows:

```
In [2 ]: newList
Out [2 ]: [‘a’ , ‘i’ , ‘a’ , ‘e’ ]
```

Remove Characters When Converting

In the previous example, the console printed the apostrophes as part of the new string. When you convert a list to a string, the square brackets around the original list and the commas between list elements become part of the new string. When converting objects, you might want to remove those extra characters. Example 7.27 demonstrates slicing to remove square brackets and apostrophes. With strings, you can use the **.replace()** function to remove square brackets or commas.

The same principle applies when you convert a string object to a list. A string object begins with an apostrophe and ends with an apostrophe. The apostrophes become part of the new list element.

Delete an Item from a List

This expression uses the del() function to delete the second item in “mylist.”

```
del (mylist[1 ])
```

In another example, the statement “**del(mylist[4])** ” deletes element ‘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', 'l', 'l']
```

Change a List to a String

In case you have a “list” object, and need a “string”, convert the list to a string using “**str(** myList**)** .” In this example I convert a list element to a string.

```
myList = ['h i']
mynewvar = str
(myvar)
```

And the Console shows:

```
In [2]: mynewvar
" ['h', 'i']"
```

In the Console, type “mynewvar.” The object has **double quotes** around the output, indicating the object is now a string. The **square brackets** that were around the original list object as well as the **commas** that separated the original list elements are part of the value of the new string object “mynewvar.” Example 7.27 demonstrates slicing to remove square brackets and apostrophes.

Change a String to a List

In case you have a “string” object and want to use a “list” method with the string, convert the string to a list using “**list(** mystring**)** .” You might need to convert a string to a list to change data. Recall a list is mutable and can be changed.

```
myList = ['hello', 'world'
]
```

```
str (mysList)
```

And the Console shows:

```
In [2 ]: myStr
'hello'
```

3.11 Methods for Lists

In this topic, we'll look at several handy list methods. To see common sequence operations available for your version of Python, check out the **docs.python.org** . In the top left corner, you can select your language and version, and then on the left side of the page under "Sequence Types -- **list, tuple, range** , click on Mutable Sequence Types .

Syntax	Comments
mylist.append ('there')	Adds element to the end of the list
mylist.clear()	Removes all items from mylist
mylist.copy()	Creates a shallow copy of mylist (same as **mylist[:]**)
mylist.count('l')	Count occurences of 'l'
mylist.extend('you')	Add 1 element to end of list
mylist.index[1]	Returns 2nd element
mylist.insert (2 , 'a')	Insert item at index location
mystring = '_' .join(mylist)	Joins list elements into a new string
mylist.pop(2)	Deletes element ' 2' and returns ' 2'
mylist. remove(2)	Removes element ' 2' from mylist
mylist.reverse()	Reverses the items of mylist in place
mylist.sort()	Sorts the original list & returns nothing

Table 3.5 List Methods

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', 'l', 'l', 'o', 'there']

Extend a List, or Combine 2 Lists

The function "extend()" adds each element in the second list as a separate element to the end of the first list. In this example, when I extend the list with ' you,' it separates the letters in 'you' into three elements.

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

And the Console shows:

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

Index: Location of a List Element

To find the index for the element 'e,' use the **.** index method. In this example, **.** index returns "1" which is the index for the letter 'e.' Python begins counting at "**0** ," so the second element 'e' has an

index of “1.” Once you know the index for an element, you can change that element.

```
mylist = [‘h’ , ‘e’ , ‘l’ , ‘l’ , ‘o’ ]
mylist. index [‘e’ ]
```

And the Console shows:

In [2]:
1

Insert an Item to a List

To add an item “a” at position ‘2’ in a list, 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’]

Join List Elements into a String

The join() function is useful for joining elements in an iterable (list, tuple, dictionary, set.) In this example, I am creating a **new string** . The “_” underline character is used as an argument so that the new string “mylist2” has “_” between each letter.

```
mylist = [‘h’ , ‘e’ , ‘l’ , ‘l’ , ‘o’ ]
mystring = ‘ _ ’ . join (mylist)
```

And the Console shows:

In [2]: mystring
h_e_l_l_o

In this next example, I join the elements into a string, and nothing is added between the elements.

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

And the Console shows:

In [2]: mystring
'hello'

> Instead of '_', you could use '\n' to add a line feed between elements.

Add Character When Joining Strings

In this next example, I use the "map" function to add a comma between elements. The output is, "hi, stranger."

```
mylist = [' h i' , 'stranger' ]
mystr = ' , ' . join (map (str ,
mylistvar))
print (myStr)
```

Pop (Remove) an Element from a List

The function "**mylist.pop()** " 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

Remove an Element 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']

Remove Duplicate Elements from a List

To remove duplicate elements in a list, convert the list into a **Set** , as explained in the topic, "Set " later in this chapter.

The sort() Method

There are several ways to sort list items. The mylist**. sort** () method changes the original "mylist" into a sorted list and returns nothing.

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

And the Console shows:

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

The sort() method also has two arguments. The following shows two possible arguments for the list.sort() method.

key=str**.** lower (comparison key for each list element)

reverse=True (reverse sort)

The sorted() Function

The "**sorted** ()" function returns a new, sorted version of your list (or any iterable.) In this example, mylist2 is a new sorted list.

```
mylist = ['h' , 'e' , 'l' , 'l' , 'o' ]
mylist2 =sorted (mylist)
```

And the Console shows:

In [2]: mylist2
['e' , 'h' , 'l' , 'l' , 'o']

Sort Order and the Key Parameter

The sorted() function can also sort in "descending" or "ascending" order with the keyword "**reverse** ." In the next example, "reverse" is a keyword argument of the sorted() function. The argument specifies a function to be called on each list element prior to making comparisons. In this example, I use the **str . lower** method to change strings to lowercase before the **sort** comparison.

```
mylist = ['red' , 'Red' , 'blue' ]
mylist2 = sorted (mylist, key=str. lower, reverse =True)
```

Sorting by the Second Element in the List

The “operator ” library includes the “itemgetter,” which allows you to choose which element in the list you want to sort. In this example, I am sorting by the second element and then the first element.

```
from operator import itemgetter
mylist = [('red' , 3 ), ('Red' , 1 ), ( 'blue' , 2 )]
mylist2 = sorted (mylist, key=itemgetter(1 ,0 ))
```

3.12 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 sequence of indexes to the items. The first item in a tuple has an index of “0.” Tuples use parentheses **( ),** and commas separate items. Tuples are immutable, which means they cannot be changed, as demonstrated in example 7.42.

Tuples can also contain other 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 )
```

When working with functions, you’ll notice they often return a tuple. In the Console, when you type this expression, the Console returns “True.”

```
In [4 ]: mytuple2[2 ] == 'O'
Out [4 ]:True
```

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

Elements are separated by commas. Dictionary keys are always immutable elements. Thus, 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.
one-line swap	a, b = b, a	swap values

Table 3.6 Tuple Objects

Slicing a Tuple

Earlier, I created 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]: (1, 'two' , 3 )
```

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 zero, the 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 use this expression.

```
10 + mytuple2[ 4 ]
```

This expression evaluates to "14."

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])
```

Tuples and Function Return Objects

By definition, a function can only return one object. One way around this is to return a tuple of more than one object. Sometimes you'll see return values "swapped," as shown below. We look at these "unpacking " statements later in this chapter in the topic, "Functions." Also, the function in example 33 returns a tuple.

```
myvar1, myvar2 = myvar2, myvar1
```

The example below is a function that returns one tuple with two objects in the tuple. The first few lines create the function. Line 7 invokes the function and assigns "myint1" and "myint2" to the function's return objects. Because the function returns "var1" and "var2," the result is the same as the expressions that assign **myint1, myint2 = ("var1, "var2").**

```
1
2 def myfunction ()
3         var1 = 3
4         var2 = 4
5         return var1, var2
6                 myint1, myint2 = myfunction()
7
8
```

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

Example 7.52 looks at a tuple index error.

Repetition and Concatenation

Tuples support both concatenation and repetition. In the examples that follow, I am using repetition and addition for tuples. Because tuples are immutable and can't be changed, this addition does not change the value stored in "mytuple." The Console output shows the addition result, but the result is not assigned to an object. 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)

Example 7.43 concatenantes a tuple to a tuple. You can not concatenate a tuple with a string.

	Expression	Returns:
Repetition	2 * (‘H’, 2, ‘O’)	(‘H’, 2, ‘O’, ‘H’, 2, ‘O’)
Addition	mytuple + (‘H’, 2, ‘O’)	(1, ‘two’ , 3, ‘H’, 2, ‘O’)

Table 3.7 Tuple Expressions

Related Tuples

Often, the relationship between elements in two or more tuples is dependent on the “order” of the elements. In this section, we use multidimensional data in simple tuples. Check out **numpy.py** which is specifically for arrays.

```
roster1 = ( ( ‘Joan’ , [ ‘art’ , ‘ p c’ ] , [ 60 , 59 ] ),( ‘Henry’ , [ ‘math’ ] , [ 96 ] ),
    ( ‘John’ , [ ‘english’ , ‘SS’ ] , [ 80 , 87 ] ))
```

- The first element in the tuple is the **student name** .
- The second element in the tuple is a list of **courses** .
- The third element in the tuple is a list with the **grades** for the corresponding course.

Henry data in roster1			
	Name	Course	Grade
index	roster1[1][0]	roster1[1][1][0]	roster1[1][2][0]
value	Henry	math	96

Table 3.8 Index Values for Henry in Roster1

The roster1 tuples

The roster1 tuples
(‘Joan’ , [‘art’ , ‘pc’], [60 , 59] **)**
(‘Henry’ , [‘math’], [96] **)**
(‘John’ , [‘english’ , ‘SS’], [80 , 87] **)**

Table 3.9 Class Roster

You may decide that you want to add a third list for term, so you have a new tuple “roster2” with each student’s data in their own tuple, as shown below.

- The first element in the tuple is the **student name** .
- The second element in the tuple is a list of **courses** .
- The **third** element in the tuple is a new list with the course **terms** for the corresponding course.
- The **fourth** element in the tuple is a list of **grades** for the corresponding courses.

The new roster2 tuple
(‘Joan’ , [‘art’ , ‘pc’], [‘spr’, ‘spr’], [60 , 59])
(‘Henry’ , [‘math’], [‘fall’], [96])
(‘John’ , [‘english’ , ‘SS’], [‘fall’, ‘fall’], [80 , 87])

Henry data in roster2				
	Name	Course	Term	Grade
index	roster2[1][0]	roster2[1][1][0]	roster2[1][2][0]	roster2[1][3][0]
value	Henry	math	fall	96

Table 3.10 New Class Roster

Because you changed the elements in the tuple, the code needs to be "**refactored** " to account for this change. In this example, we need to refactor or change the internal code so it behaves the same way with the course and grade elements.

Henry's grade initially: roster1[1][2][0]

Henry's grade after the change: roster2[1][3][0]

NamedTuple

To avoid having to refactor code, the "collections " library includes a "**namedtuple** " which makes these type of tasks very simple. In the example below, I create a new **namedtuple** "Roster" on line 3. On lines 5-7, I assign values to "Roster."

NamedTuple pt 2.py*

```
1 from collections import namedtuple
2
3 Roster = namedtuple('Roster', ['student', 'course', 'term', 'grade'])
4
5 roster3 = (Roster('Joan', ['art', 'PC'], ['spr', 'spr'], [60, 59]),
6            Roster('Henry', ['math'], ['fall'], [96]),
7            Roster('John', ['english', 'SS'], ['fall', 'fall'], [80, 87]))
8
```

Figure 3.1 Create a namedtuple

Now that we are using a "namedtuple" the syntax to access the data is a little different. First let's look at Henry's data.

Henry's data in roster3			
Name	Course	Term	Grade
roster3[1].student	roster3[1].course[0]	roster3[1].term[0]	roster3[1].grade[0]
Henry	math	fall	96

Table 3.11 New Class Roster3 data for Joan

Joan's data in roster3			
Name	Course	Term	Grade
roster3[0].student	roster3[0].course[1]	roster3[0].term[1]	roster3[0].grade[1]
Joan	PC	spr	59

Table 3.12 New Class Roster3 data for Joan

The Zip Function Returns Tuples

The **zip()** function takes two or more iterables as arguments and returns a "zip" object that is an iterator of **tuples** . For example, when you pass two lists with three elements each, zip() returns three tuples and pairs the items in the tuples.

The first example uses the **zip()** function with two lists, and returns a tuple that is unpacked to the "order" and "color" variables.

```
for order, color in zip ([1 , 2 , 3 ], [ 'green' , 'red' , 'blue' ]):
    print (order, color, '\n' )
```

The Console prints the following text.

```
1 green
2 red
3 blue
```

In this example, I want to create a range of scores for a plot chart using the matplotlib library. I'm going to use 'school_Dict' as my iterable for the **zip()** function. For examples on matplotlib see example 7.19 and 7.22.

First, I use two zip objects to create 'newList.'

```
newList = [dict (zip (school_Dict, col)) for col in zip (* school_Dict.values())]
graphNums = list (newList[2 ].values())
```

Figure 3.2 newList

Then, I assign the third element in newList[2] to another list "graphNums."

Figure 3.3 graphNums

In the next example, I create the "s" tuple for the scores using the asterisk, as shown below. The asterisk * unpacks the sequence into positional arguments and behaves like an "unzip."

```
school_Dict = { 'John' : [ 'Civics' , 'A' , 94 , 'Dr. Brown' ],
        'Mary' : [ 'English' , 'B' , 82 , 'Dr. Smith' ],
        'Herman' : [ 'Physics' , 'C' , 70 , 'Prof. Stanley' ]}

c, g, s , t = list (zip (* myDict.values()))
```

Figure 3.4 The Tuple 's'

3.13 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 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. Dictionaries are mutable. Values can be duplicated, but keys must be

unique and hashable . You can only use immutable elements such as integers or strings as dictionary keys, as shown in example 7.39 and 7.46. 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
Grade	A
Course	Python Programming

Table 3.13 Sample Dictionary

A dictionary has unsorted elements that can grow and shrink as needed. When creating Dictionaries, use curly braces **{}** . In the assignment statement, the key is followed by a colon **:** and a value .

myDict = **{** key:value , key:value , key:value **}**

The syntax for changing dictionary objects depends on the type of object in the dictionary "value." If the value is a list, you can use list methods to change data. If the value is a string, you can't change the string because strings are immutable; however, you can replace the value in the key pair with a new object of any type.

In this section we'll explore these topics.

- Elements in a Dictionary
- Create a Dictionary
- Append to a Dictionary
- Copy a Dictionary
- How Many Elements are in a Dictionary List?
- Assign a Dictionary Value Using the Key Name
- Update a "List" Value in a Dictionary
- Find the Value of a Dictionary Item
- Find the Type of a Dictionary Element
- Add a New Key to an Existing Dictionary
- Delete a Key in an Existing Dictionary
- Iterate Through Dictionary Key-Pairs
- Iterate Through Keys in a Dictionary
- Retrieve Keys

- Search for a Key Name
- Title and Value Methods
- Combine Dictionaries

Earlier I said adding a new key:value pair to a dictionary is straightforward, but adding to an existing dictionary is not. In this case, it's not about the object's value, but more about the "type" of an object in the key:value pair. As you'll see in the next topic, it's important to know the type of the dictionary object so that you can use the corresponding methods to add or change values.

Elements in a Dictionary

This topic explores the relationship between elements in a dictionary. A dictionary is a set of key:value pairs that are not in any particular order. The keys are immutable objects like strings or integers. The values in the key:value pair can be any type of object. Often you'll see nested lists, tuples, or dictionaries in a dictionary **key:value** pair.

In the following example, we combine two lists into one "combined" list. The "combined" list will be the dictionary value. The order of elements in the two lists establishes the relationship between the two lists. The keys are student names, and the value for each student is a "combined" list that contains two lists. Recall that a dictionary is not in any kind of order, so there is no guarantee that the key "Joan" will be the first in the dictionary.

Key	Value
Joan	[[‘art’, ‘social studies’], [‘A’, ‘D’]]
Henry	[[‘math, ‘english’, ‘science’], [‘A’, ‘B’, ‘C’]]
John	[[‘english’, ‘history’, ‘algebra’], [‘B’, ‘B+’, ‘C’]]

Table 3.14 Student Dictionary

Because dictionaries don’t store data in the same order you input the data, the actual data can end up looking like the next example.

Figure 3.5 Variable Explorer Showing Dictionary

When looking at Joan’s data, you can see two lists. The first course list has two elements. The second grade list also has two elements. If Joan is taking four classes, each list would have four elements.

[‘art’, ‘social studies’]

[‘A’, ‘D’]

The “combined” list represents the “value” for the key “Joan.”

[[‘art’, ‘social studies’], [‘A’, ‘D’]]

When looking at this combined list, the first element in the “combined” list is a list of classes. The second element is a list of grades.

[0] -> class_list

[1] - > grade_list

We’re going to create a dictionary with three students, and each student is taking several classes. Notice that we assign the list data when we create the dictionary. We don’t have a separate statement to create the two lists, although we could add an additional statement for clarity.

Joan is taking art and social studies. Joan’s respective grades are A and D.

[‘art’, ‘social studies’]

[‘A’, ‘D’]

John is taking English, history, and algebra. John’s respective grades are B, B+, and C. When we create the dictionary, our class list will look like this example.

[‘english’, ‘history’, ‘algebra’]

The grade list will look like this example.

[‘B’, ‘B’+, ‘C’]

Henry is taking math, English, and science. Henry’s respective grades are A, B, and C.

[‘math’, ‘english’, ‘science’]

[‘A’, ‘B’, ‘C’]

Now let’s create the “students” dictionary with one student.

students = {‘John’: [[‘english’, ‘history’, ‘algebra’], [‘B’, ‘B+’, ‘C’]]}

At this point, let’s say you want to display John’s grade in his history class. Let’s use indexing to look at the data in this dictionary key pair. You can follow along with this code in your iPython Console to see the data returned. To see all of John’s classes type this statement. In this example, the first list is element [0].

students[‘John’][0]

To see all of John’s grades type this statement. In this example, the second list is element [1].

students[‘John’][1]

Because dictionaries don't store data in the same order you input the data, the actual data can end up looking like the next example.

Figure 3.6 John's Grades

To see the class name 'history' type this statement. In this example, you want to see the second element [1] in the first list [0].

```
students['John' ][0 ][1
]
```

Finally, to see the class name algebra with John's grade, type this statement. Here we are using the third element in the first list [0][2], along with the third element in the second list [1][2].

```
print( students['John' ][0 ][2 ], students['John' ][1 ][2 ]
)
```

Recall that a dictionary assignment statement follows this syntax.

```
myDict = { key:value , key:value , key:value
}
```

To create a dictionary with all students use this expression.

```
students = { 'John': [['english', 'history', 'algebra'], ['B',
'B+', 'C']],
  'Henry': [['math', 'english', 'science'], ['A', 'B', 'C']],
  'Joan':['art', 'social studies'], ['A', 'D']] }
```

To view Joan's grade in social studies, use this statement. In this example, we're looking at the second element [1] in the first list [0], and the second element[1] in the second list [1].

```
print (students['Joan'][0][1], students['Joan'][1][1])
```

Figure 3.7 Joan's Grade in Social Studies

Create a Dictionary

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

To create an empty dictionary use the following expression.

```
mydictionary = { }
```

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

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

In the following example, I am creating a dictionary with “list” values. Inside the “list’ is a string value. Because it is a “list” there are square brackets around the string values.

```
mmydictionary2 = {‘Name’ : [
‘Young’ ]
        ‘Grade’ : [ ‘B’ ] ,
        ‘Course’ : [ ‘Excel
        Fundamentals’ ] }
```

Because a dictionary can have any type of object, you must know the object’s type in order to work with the data. Strings are contained in **quotes** (single or double), tuples use **parentheses()** , and lists use **square brackets [ ]** .

Append to a Dictionary List Value

Append adds elements to an existing key that contains a “list.” For example, if you have a key “Name” with a “list” value, you can use append to add additional strings ‘Smith’ and ‘Jones’ to the “list.”

Why should you care about the object type of elements in your dictionary? If you try to **add** elements to a “string” using the “append” method, the Python Interpreter raises 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 of type “list.”

```
In [5 ]: mydictionary2[‘Name’ ] = [‘Zimmerman’]
In [6 ]: 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 square brackets indicate **[** ‘Zimmerman’, ‘Smith’ **]** is a “list.”

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

Copy a Dictionary

Use the **.copy()** method to make a shallow copy of a dictionary.

```
thisdict1 = {3 , ‘k’ }
thisdict2 = dict1.copy()
```

Import the “copy” library and use **copy.deepcopy()** if the dictionary contains mutable objects that can be changed. .

How Many Elements are in the Dictionary List?

Continuing with the earlier example, I can count the number of values associated with the key “Name” by 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’
```

Update a “List” Value in a Dictionary

There will be times when you want to “update” the value for a key in your dictionary. In this example, the key/value pair has a value that is a “list.” First, I check if the key already exists in the dictionary. If the key is there, I append “myvar2” to the “list.” If the key is new, I add the key + list value to the dictionary.

```
mydict = {}
marvar1 = ‘hi’
myvar2 = ‘hello’
if myvar1 not in mydict:
mydict[myvar1] = [ myvar2]
else:
mydict[myvar1].append(myvar2)
```

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’
```

Keep in mind that a dictionary key/pair might contain anything. For example, the key pair value could contain a tuple with several lists, and each list could have multiple elements. In that case, you use indexes to locate the elements that may be nested several layers into the dictionary key pair.

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

> To avoid an error if the key is not in the dictionary, use the .get() method or the collections library "defaultdict ."

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 square 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

Add a New Key Pair to an Existing Dictionary

To add a new key 'Credits' to an existing dictionary, use this syntax. Recall that you can only use immutable elements such as integers or strings as dictionary keys.

mydictionary[' Credits'] = ' 3'

Delete a Key in an Existing Dictionary

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

del(mydictionary[' Credits'])

Iterate Through Dictionary Key Pair Values

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 3.8 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 )
```

Retrieve Keys

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

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

Search for a Key Name

To test if a particular key is in a dictionary, you can use the "in" operator. Continuing with our previous dictionary example, I might look for the keys **"Grade"** or **"Course"** with these expressions. In this example, I am typing in the Console, and the expression returns either False or True.

In [**4**]: **“Grade”** in mydictionary
Out [**4**]: False

In [**5**]: **“Course”** in mydictionary
Out [**5**]: True

To avoid an error if the key is not in the dictionary, use the mydictionary.**get()** method. Example 7.20 demonstrates an error when a key is not in a dictionary.

Test if Key is in the Dictionary

In another example, let’s say you have a dictionary called “mydictionary’ for student IDs, where the key name is the student ID number. This expression to find a student ID “12345” returns “**True** .”

In [**6**]: if ‘12345’ in mydictionary:
Out [**6**]: **True**

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]:

Combine Dictionaries

The **update()** method combines two dictionaries. I'd recommend you only use this method if there are no duplicate, or overlapping, key names.

```
dict1, dict2 = {key1: 'data1', key2: 3}, {key3: '4'}
dict1.update(dict2)
```

Dictionary Comprehensions

Earlier, we looked at list comprehensions. Dictionaries also support comprehensions.

In [16]:	myl = [1, 2, 3, 1, 3, 4, 1]
In [17]:	dd = {num: myl.count(num) for num in myl}
In [18]:	print (dd) {1: 3, 2: 1, 3: 2, 4: 1}

3.14 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** , to the Console 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 an ideal way 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])
```

3.15 Generators

A generator function returns a 'yield' statement and is an iterator. A generator runs up to the first 'yield' statement and returns that value. The next time the generator is invoked, it resumes execution at the point it left off and runs until the next 'yield' statement. If you have a list of a million names, you don't have to create the list in memory. Instead, the generator creates each name when needed.

```
1 # Generator.py
2
3
4 def nameListGen():
5     for i in range(1000000000):
6         studentName = 'student_'
7         studentName += str(i)
8         yield studentName
9
10
11 studentNames = nameListGen()
12
13 for j in range(20):
14     print(studentNames.__next__())
15
16
```

Figure 3.9 A Generator

Line 14 could also use the **next()** method, as shown below.

print (next(studentNames))

3.16 Set

A **Set** contains unordered elements with no duplicates. A Set is mutable and grows or shrinks as needed, but you can't change the immutable elements in the set. Set elements can be different types.

When creating a set, use curly braces **{}** and separate items with a comma. In the example below you can see the second "p" in apple is not included, because a set has no duplicates. Also, the elements are unordered.

In [**4**]: myset = {'apple' }
In [**5**]: myset
Out[**5**]: {'a' , 'e' , 'l' , 'p' }

The set**(** *<iterable>* **)** function creates a new set from an iterable such as a list, tuple, dictionary, or string. While you can create a "set" from a list using the set() function, the elements in the new "set" are immutable and can't be changed.

In this example, on input line 7 I enclose an immutable "string" variable in braces **{}** , creating a set.

In [**6**]: myvar = '123'
In [**7**]: **{** myvar**}**

In [**8**]: type({myvar})

Out[8]: set

Notice a **TypeError** is raised when I try to use a mutable “list” as a set.

In [9]: myvar = [1 , 2 , 3]
In [1 0]: { myvar}
Traceback (most recent call last):
File “<ipython-input-20-597c80421cc7>”, line 1, in <module>
{mylist}
TypeError: unhashable type**: ‘list’**

Description	Syntax
Create a Set and assign values	myset = {‘a’, 7 , ‘c’}
Create an empty Set	myset3 = set()
Create a Set and assign values	myset3 = set(‘abc’)

Table 3.15 Creating Sets

In Variable Explorer Settings, uncheck “Exclude unsupported data types” to view an object of type “set”. When you double click on the set object, Variable Explorer displays set functions, as shown below.

Figure 3.10 A Set in Variable Explorer

We’ll look at these built-in functions next.

- Union
- Intersection
- Difference
- Symmetric

Union

First, let’s use the “union” **|** operator to combine two sets.

```
In [12]: myset1 = {‘apples’}
In [13]: myset2 = {‘pears’}
In [14]: myset1 | myset2
         {‘cherries’, ‘oranges’}
```

```
Out[14]:
```

The union() function accepts other iterator types and returns a **new** set. In the next example, I'm combining a set and a list into a new set.

```
In [15]: myset1 = {'apples'}
In [16]: mylist1 = ['pears']
In [17]: myset1.union(mylist1)
Out[17]: {'cherries', 'oranges'}
```

Intersection

The intersection() function finds elements in common between two sets and returns a set.

```
In [18]: myset1 = {1, 2, 3}
In [19]: myset2 = {3, 4, 5}
In [20]: myset1.intersection(myset2)
Out[20]: {3}
```

The next example uses the intersection operator **&** to create a new set with the elements in common between the two sets.

```
In [21]: myset1 = {1, 2, 3}
In [22]: myset2 = {3, 4, 5}
In [23]: myset1 & myset2
Out[23]: {3}
```

Difference

The difference() function returns a **new set** of elements in myset1 that are not in myset2.

```
In [18]: myset1 = {1, 2, 3}
In [19]: myset2 = {3, 4, 5}
In [20]: myset1.difference(myset2)
Out[20]: {1, 2, }
```

The next example uses the difference operator **-** to create a new set with the elements in myset1 that are not in myset2.

```
In [21]: myset1 = {1, 2, 3}
In [22]: myset2 = {3, 4, 5}
In [23]: myset1 - myset2
Out[23]: {1, 2}
```

Symmetric Difference

The symmetric_difference() function returns a new set of elements in either set that are not in both sets.

```
In [18]: myset1 = {1, 2, 3}
In [19]: myset2 = {3, 4, 5}
In [20]: myset1.symmetric_difference(myset2)
Out[20]: {1, 2, 4, 5}
```

The next example uses the operator **^** to create a new set with the elements in common between the two sets.

```
In [21]: myset1 = { 1, 2, 3}
In [22]: myset2 = {3, 4, 5}
In [23]: myset1 & myset2
Out[23]: {1, 2, 4, 5}
```

3.17 collections

The “collections ” library has some interesting “container datatypes.” Earlier in the “Tuples” topic we looked at “namedtuples” from the “collections” library. Now we’ll also look at a “Counter” and “defaultdict.”

namedtuples

Earlier, in the “Tuple” topic, we looked at an example of a “**namedtuple** ” from the collections library.

Counter

The “Counter” subclass is a dict subclass and is a simple way to keep track of counts, as shown in the next example. I increment a counter on line 5 to count each instance of an element in the “bills” list.

```
from collections import Counter

bills = [‘one’ , ‘five’ , ‘ten’ , ‘one’ ]
cnt = Counter()
for i in bills:
        cnt.update([i])
ones = cnt['one']
print (‘there are %d ones’ % (ones))
```

defaultdict

The “ defaultdict” subclass returns a dictionary-like object. In this example line 4 sets the initial value for my defaultdict “dd” to a “list.” On line 4 I’m using the list “append” method to add items to “dd” object. With this syntax I don’t have to “check” if a key is already in “dd,” and I can use the list method **.append** .

```
from collections import defaultdict

mylist = [(1 , 'one' ), (2 , 'two' )]
dd = defaultdict(list )
cnt = Counter()
for k, v in myList:
        dd[k].append(v)
```

In the Console, I type “dd” to see the values.

```
In [ 2 ]: dd
Out[ 2 ] : defaultdict(list, {1: ['one' ], 2: ['two' ]})
```

3.18 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 square brackets around the index. For example, **mystr[4]** evaluates to “s” in the string “b” example that follows.

In Python, the index starts at position 0. In the example below, ‘**bookstore** ’ is assigned to **mystr** , and there are nine 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.

mystr = “**bookstore** ”

0 start index

8 end index

9 length of string

To find the length of string “**mystr** ” use the len() function.

len(**mystr**) returns **9**

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

Table 3.16 String Index Example

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

Indexing is a fundamental part of Python, so we'll take a moment to look at a couple of "indexes" with data structures that have multiple levels.

Indexing Elements in Nested Lists

Earlier, we looked at how to index two lists inside of a dictionary. In this example, we'll look at three lists nested inside of the main list.

```
list1 = ['apple' , 'tangerine' ]
list2 = ['tangy' , 'sweet' ]
list3 = ['red' , 'orange' ]
mainlist = [list1, list2, list3 ]
```

The first three lists are elements [0], [1], and [2] in "mainlist." Each of the three lists has two strings, elements [0] and [1].

To retrieve 'tangerine,' you access the second element [**1**] in "list1," and "list1" is the first element [**0**] in "mainlist."

```
In [ 1 ]: mainlist[0 ][1 ]
Out [ 1 ]: ' tangerine'
```

To view the data 'sweet,' you access the second element [**1**] in "list2," and "list2" is the second element [**1**] in "mainlist."

```
In [ 2 ]: mainlist[1 ][1 ]
Out [ 1 ]: ' sweet'
```

To view the data 'tangy,' you access the second element [**1**] in "list3," and "list3" is the third element [**2**] in "mainlist."

```
In [ 2 ]: mainlist[2 ][1 ]
Out [ 1 ]: ' orange'
```

Indexing Lists & Tuples in Dictionaries

These examples use strings, lists, and tuples with dictionary key:value pairs. First, I create three lists with various fruit values. The relationship between the lists is such that the first element in each list describes the “apple” - “sweet” and “red.”

```
f1 = [‘apple’ , ‘lemon’ ]
f2 = [‘sweet’ , ‘sour’ ]
f3 = [‘red’ , ‘yellow’ ]
```

Element	Value
f1[0]	apple
f1[1]	lemon
f2[0]	sweet
f2[1]	sour
f3[0]	red
f3[1]	yellow

Table 3.17 Fruit Lists

In the **Console** , I can type the following to see information about the “lemon.” My design is such that the data about the “lemon” is the second element [1] in lists f1, f2, and f3.

```
In [ 1 ]: f1[1 ]
Out [ 1 ]: ‘ lemon’

In [ 2 ]: f2[1 ]
Out [ 2 ]: ‘sour’

In [ 3 ]: f3[1 ]
Out [ 3 ]: ‘yellow’
```

Next, I create some lists for vegetable values.

```
v1 = [‘spinach’ , ‘carrots’ ]
v2 = [‘leafy’ , ‘crunchy’ ]
v3 = [‘green’ , ‘orange’ ]
```

Element	Value
v1[0]	spinach
v1[1]	carrots
v2[0]	leafy
v2[1]	crunchy
v3[0]	green
v3[1]	orange

Table 3.18 Vegetable Lists

Now I create two tuples, and each tuple consists of three lists.

```
f = (f1, f2, f3)
v = (v1, v2, v3)
```

Tuple Element	Value
f[0]	f1
f[1]	f2
f[2]	f3
v[0]	v1
v[1]	v2
v[2]	v3

Table 3.19 Two Tuples

The chart below outlines the indexes for all the values in the “**f** ” tuple.

Tuple (3 list elements)	List 2 string elements	Values	Tuple Index
f[0]	f1[0]	apple	f[0][0]
f[0]	f1[1]	lemon	f[0][1]
f[1]	f2[0]	sweet	f[1][0]
f[1]	f2[1]	sour	f[1][1]
f[2]	f3[0]	red	f[2][0]
f[2]	f3[1]	yellow	f[2][1]

Table 3.20 Tuple Indexes

In the Console, I can type the following to see information about “lemon” in the “**f** ” tuple. The layout of data is such that the data about the “lemon” is the second element [1] in each list. Because Python starts counting at zero, the second element index is [1].

In [**1**]: f[0][1]
Out [1]: ‘lemon’

In [**2**]: f[1][1]
Out [2]: ‘sour’

In [**3**]: f[2][1]
Out [3]: ‘yellow’

Finally, I create a dictionary with two key:value pairs. The values are the “**f** ” and “**v** ” tuples.

d = {‘fruit’ : f, ‘vegie’ : v}

Element	Value
d[‘fruit’]	f
d[‘vegie’]	v

Table 3.21 Dictionary

At a glance, the dictionary might look simplistic. However, when we look inside the dictionary, you see there is quite a bit of data.

Dictionary (2 tuple elements)	“f” and “v” tuples	f1,f2, f3, v1, v2, v3 Lists	Values (strings)
d[‘fruit’]	f[0]	f1[0]	apple
d[‘fruit’]	f[0]	f1[0]	lemon
d[‘fruit’]	f[1]	f2[1]	sweet
d[‘fruit’]	f[1]	f2[1]	sour
d[‘fruit’]	f[2]	f3[0]	red
d[‘fruit’]	f[2]	f3[0]	yellow
d[‘vegie’]	v[0]	v1[1]	spinach
d[‘vegie’]	v[0]	v1[1]	carrots
d[‘vegie’]	v[1]	v2[0]	leafy
d[‘vegie’]	v[1]	v2[0]	crunchy
d[‘vegie’]	v[2]	v3[1]	green
d[‘vegie’]	v[2]	v3[1]	orange

Table 3.22 Dictionary Elements

So far, we’ve looked at indexing for lists and tuples. Now, I want to look at the values in the dictionary, beginning with “apple.” In the **Console** , I can access the data using the dictionary key name and indexing, as shown below and in the chart that follows..

In [1]: d[‘fruit’][0][0]
Out [1]: ‘apple’

In [2]: d[‘fruit’][1][0]
Out [1]: ‘sweet’

In [3]: d[‘fruit’][2][0]
Out [3]: ‘red ’

In the previous examples, the key name is “fruit” and I’m accessing the first element [0] in all three lists. The three lists are index [0], [1], [2].

Dictionary Key	“f” Tuple	f1, f2, f3” List	Values	Dictionary key name and indexes
d[‘fruit’]	f[0]	f1[0]	apple	d[‘fruit’][0][0]
d[‘fruit’]	f[0]	f1[1]	lemon	
d[‘fruit’]	f[1]	f2[0]	sweet	d[‘fruit’][1][0]
d[‘fruit’]	f[1]	f2[1]	sour	
d[‘fruit’]	f[2]	f3[0]	red	d[‘fruit’][2][0]
d[‘fruit’]	f[2]	f3[1]	yellow	

Table 3.23 Dictionary Indexes

Example 7.52 in Chapter 7 looks at a tuple index error.

3.19 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 square brackets **[]** and takes three arguments separated by colons. The first argument “start,” tells the function where to start slicing 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 from the previous example, follow along as we look at this slicing. Notice the stop index is **9** . Earlier, we saw the function **len(b)** returns **9** , so the string’s length is **9** characters.

b = “**bookstore** ”

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

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

The Console prints: **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 3.24 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

Don't forget the colon at the end of the start argument!

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

In 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 Chapter 7, example 7.27 uses slicing to remove square brackets and apostrophes after converting a list to a string.

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

The last 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 three indices from right to left and can’t move anymore.

In the topic “Find a Substring ” that follows, we’ll use slicing to find the word “from” in a string.

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

3.20 Operators

Now let's take a look at using operators for numerical operations, concatenation, and comparisons. The docs.python.org has a complete list in the topic "Mapping Operators to Functions ."

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 or exponentiation

Table 3.25 Numerical Operators

One way to select odd or even index 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 even index 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 = 1
while i < len (myTuple1):
    myTuple2 += (myTuple1[i],)
    i += 2
print (myTuple2)
```

Modulo Operator

The "**modulo** " operator "**%** " returns the remainder when dividing two numbers. If you divide any number by 10, the modulo is the last digit in a number. This example returns '7' in the Console. The topic "Recursive Functions " includes an example using the modulo operator and integer division.

```
n = 12345678907
last_digit = n % 10
print(last_digit)
In [1]: 7
```

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

```
for i in range(0 , 20 ):
if i % 2 != 0 :
print ("i is an odd number" , i)
```

Integer Division

The '//' operator does integer division. This example returns '12' in the Console, in effect reducing the number by one digit.

```
n = 123
test_next = n // 10
print(last_digit)
In [2]: 12
```

Concatenation, Repetition and Sequence Operations

The **+** operator is used to concatenate strings objects. You can also use the **+** operator to concatenate two lists. In Chapter 7, Examples 32, 34 and 44 demonstrate concatenation.

The * repetition operator is supported by most sequence types. An example of repetition is **3 * 'ho'** , which evaluates to "ho ho ho."

To see common sequence operations available for your version of Python, check out the **docs.python.org** . In the top left corner, you can select your language and version, and then on the left side of the page under, "Sequence Types, click on "Common Sequence Operations ."

Operator	Description
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s
s.count(x)	total number of occurances of x in s
in	Test for membership
not in	Test for membership

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

Table 3.26 Comparison Operators

To test whether a character or substring is in a string, use the "in" comparison operator, as shown below. Example 7.48 illustrates comparing two strings.

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

In the previous example, I searched for the string "a" in "mystr." In this example the left operand and the right operand are both strings which is valid syntax. A TypeError is raised if you use a "list" as the left operand and a string as the right operand. Example 7.48 illustrates this example.

> 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
```

The Union Operator for Sets

Earlier in the “Sets” topic we looked at the “union” **|** operator combines two sets.

Find a Substring

Slicing is a simple way to parse a string into substrings. In the next example, I use the find() method to locate all instances of the substring “**the** .” I’m using a “**start** ” variable for the beginning of the string and a “stop” variable for the end of the string. As I loop through the string, each time I find a match, I change the **start** value, so I search the “remaining” string.

```
original_str = “the apple fell far from the tree”
found_str = ‘’
search_str = original_str
start = 0
while start < len(original_str):
    start = search_str.find(‘from’)
    if start != -1:
        stop = start + 4 # ‘the’ is 3 characters
        found_str = search_str[start :stop ]
        print(found_str
        search_str = search_str[stop :]
        start = 0
```

```
else:
    break
```

Find Last Element in a String

This next example uses the split() method with a string to find the last element in a list.

```
In [16 ]: my_str = "the apple fell far from the tree"
In [17 ]: print(my_str.split(' ')[-1 ])
Out[1 6 ]: tree
```

The next example uses the rindex() method to return the start index of the last list element. In this example, the stop index in the square brackets **[]** is set by **[my_str.rindex(' ')]** .

```
In [18]: print(my_str[my_str.rindex (' ')])
Out [18 ]: tree
```

Comparing Floats

Non-integer numbers or floats are stored in computer memory as a binary representation of 0's and 1's. Calculations can introduce subtle differences where you may think both float values are 1.08, but the actual binary representation is slightly different. Comparing two floats, as shown below, could potentially return **False** .

```
x == y
```

Instead of the == equals comparison for floats, use an arbitrarily small positive number (epsilon) to compare two floats. In the next statement, the epsilon is .000001, and the statement returns 'True' if the float values are within .000001 of each other, which is good enough. The abs() function converts floats to positive numbers for the comparison. Example 7.45 demonstrates float comparisons.

```
if abs(x-y) < .000001:
```

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())
```

3.21 Identifiers

Classes identified by patterns of leading and training underscore characters have special meanings .

Interactive Interpreter

The special identifier _ underscore is used in the interactive interpreter (or Console) to store the result of the last evaluation.

Class-Private Names

Within the context of a class definition, class-private names are rewritten to avoid clashes between “private” attributes of base and

derived classes. For example, a private class variable/method/class may begin with an underscore.

System-Defined Names

Special method names refer to system-defined or "dunder " names. These "**special method names** " that begin and end with two underline characters. These methods are invoked by special syntax such as arithmetic operators or subscripting and slicing . This is Pythons approach to *operator overloading* .

__add__

__eq__

__hash__

__init__

__len__

__lt__

__main__

__name__

__new__

__str__

__sub__

__version__

A module's __name__ is set equal to '__main__' when read from standard input, a script, or from an interactive prompt. A module can discover whether or not it is running in the main scope by checking its own __name__.

In the **Console** , type the module name plus **.__version__** to view the version of the module.

openpyxl.__version__

The name of the class is found in the . **__name__ method** .

openpyxl.__name__

3.22 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 "for" or "while" loop repeats itself until a condition is met. Example 7.5 demonstrates a while loop. The tests in an "if" or "else" statement move the program into different branches depending on whether the test is "True" or "False."

For Loop

A control statement iterates or 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.

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
```

Loop Until Break

In another example, the statement below is always **True** , so the program runs until you break out of the loop.

```
while True :
```

Iteration Variable

An iteration variable can also be used to iterate through elements in a container. A container could be a list, tuple, or dictionary, As an example of an iterator, when the variable “i” is a number with a value of “0.” The first time the loop runs list[i] refers to list[0]. As the program loops, 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

A **StopIteration** error is raised to indicate the end of an iterator, as shown in example 7.35.

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

3.23 Indented Code (a Suite)

In Python, the first line of a control statement, and all the indented lines that follow, is called a “Suite” of code. 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. The following is an example of 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.

Figure 3.11 An Indented “Suite” or Block of Code

The Chapter 4 topic, “My Program Loops and Never Ends ” is an example of incorrect indentation levels. 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 7.7. The Chapter 7 examples 7.1 and 7.2 illustrate a few list errors.

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

3.24 Functions and Methods

Functions are a sequence of statements. When you define a function within a class, it is called a **method** . Functions are an example of decomposition, where you break a program into smaller, self-contained pieces. First, you define a function. Once a function is defined, you can use it as many times as you like by calling the function. Abstraction refers to the fact you don't need to know how something works, you just need to know what it does. When considering a function, look for these features.

1. What are the function inputs (arguments?)
2. What does the function do?
3. What are the function outputs (the return object?)

```
1 def menu (meal, special=False
2 ):
  <some code>
```

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
3     msg = ""
4
5     if special is True :
6
7         msg = 'The specials today are
          Mimosas. '
```

```
    if meal == 'breakfast' :
        msg = 'Breakfast is eggs and
        toast. '
    else :
        msg = 'Sorry, we ran out of food. '
    return msg
```

In this example, 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.

Calling or Invoking a Function or Method

When calling or invoking a function, you pass arguments with values to the function inside the parenthesis. If there are no arguments, the parenthesis are empty but are still required to indicate the functino call, as shown in example 7.37. In the example below, the last line calls the function "**menu** ."

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

```
def menu (meal, special=False ):
    msg = ""
    if special is True :
        msg = 'The specials today are
        Mimosas. '
```

```
9
1
0
    if meal == 'breakfast' :
        msg = 'Breakfast is eggs and
        toast. '
    else :
        msg = 'Sorry, we ran out of food. '
    return msg
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 referred to as an argument that has a value of “**breakfast** .

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 3.27 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

Order	Parameter Name	Argument Value
2	special	True

When I call this function, the parameters are now referred to as “arguments” because I am calling the function. Example 7.21 demonstrates an argument that is the wrong type.

Keyword Arguments

PEP 3102 defines keyword arguments which are optional because there is a default value in the function definition. When calling a function, a name or “keyword” precedes the keyword argument. List keyword arguments at the end of the parenthesized list, after positional arguments. The second parameter in my earlier function definition includes the **keyword** “special” with a default value of “False .”

Positional Arguments

The “meal” argument is a positional argument because it does not have a keyword. Position arguments are mandatory or required since there is no default value. You must list positional arguments before any keyword arguments.

Positional arguments must come before keyword arguments. The example below is **invalid** because a positional argument is after a keyword argument.

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

Example 7.36 has an error because it is missing one positional argument.

Optional Arguments

In the previous example of the menu() function, the “special” argument has a default value. When calling the menu() function, “special” is an **optional** argument. If you don’t provide the argument when calling the function, Python uses the default value “False .”

If a function has optional arguments with default values, the values are assigned using the “scope” that exists at the time of the function definition. The global scope is used in the previous example where the menu() function definition is on line 1. In the case of nested functions, this might be the enclosing scope, as shown at the end of this chapter in the namespace topic.

> An optional argument with a default value is another way to implement the concept of a “global variable .” Let’s say you want a running “total” value. The first time you call the function, you set the optional argument “total=0”. Within the function, you update the total value. As you make recursive function calls, you pass the latest “total” value as an argument to the recursive function call statement.

Unpacking Operator 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:

```
def print_tickets(number_of_tickets, *name):
    i = 0
    while i < number_of_tickets:
        print (name[i])
        i += 1
print_tickets(2 , 'John' , 'Alice' )
```

The function definition on Line 1 includes an asterisk ***** or "unpacking operator" to indicate there is an arbitrary number of **name** arguments (values) passed to the function. For more information on unpacking operators, see the next sections.

number_of_tickets

***name**

When I call the function on line 8, I pass it three arguments. Two of the arguments are names.

```
print_tickets(2 , 'John' , 'Alice' )
```

Unpacking Operators

PEP 448 defines "Additional Unpacking Generalizations." You can define functions to take *x and **y arguments. Unpacking operators allow a function to accept any number of arguments that aren't specifically named in the declaration.

The asterisk * or 'iterable unpacking operator' unpacks the iterable into positional arguments.

The double asterisk ** or 'dictionary unpacking operator' provides the same behavior for dictionaries. You can pass arguments stored in a dictionary to a function using ****** , as shown below. This example is an arbitrary keyword argument dictionary.

```
In [1 ]: kwargs = {color: 'blue ' , height: 4}
In [2 ]: myfunction(**kwargs)
```

To pass arguments stored in a list or tuple, use the ***** unpacking operator. This example has an arbitrary argument list for positional

arguments.

```
In [1 ]: names = [‘John’ , ‘Alice’ ]
In [2 ]: print_tickets(*names)
```

If you leave off the * unpacking operator in the function call on line 2, an error is raised, as shown in example 7.52. Example 7.41 demonstrates an error when there are too many values to unpack.

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 [4 ]: from inspect import signature
In [5 ]: print (str (signature (menu)))
(meal, special=False)
In [6 ]:
menu('breakfast' , special=False )
menu('breakfast' )
```

In Chapter 6, the topic, "What are the Function Arguments," explores function definitions. In Chapter 7, example 7.6 illustrates an **AttributeError** caused by missing keyword names when calling a method.

Function Return or Yield Objects

A function "returns" or "yields" 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 "T uples and Function Return Objects. " When a function doesn't specify a return value, it returns the special value "NoneType " discussed earlier. As the Python Interpreter moves through the code in a function, when it encounters the keyword "return," it stops execution and returns the value in the return expression. Nothing after the return statement is executed.

> When a function uses the 'yield' statement instead of the 'return' statement, the function is a **generator** , as discussed previously.

In Chapter 6, the topic, "What Type of Object Does a Function Return?" has additional information on return objects. 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 three 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 function definition is created, I can use “type” in the Console 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())
```

Return the Statement that is True

The next example of a return statement would return the value of “y” or ‘hello’ because y is a string, and x is not an integer. In this case, whichever express is “True” would be returned.

```
x = ‘john’

y = ‘hello’

return isinstance(x, int) or isinstance (y, str)
```

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.

```
  def menu (meal, special=False ):
        msg = ""
1       if special is True :
2             msg = 'The specials today are
3             Mimosas. '
4             return msg
5       if meal == 'breakfast' :
6             msg = 'Breakfast is eggs and
7             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 7.17 in Chapter 7 illustrates how to identify the value “None.”

More than One Return Object

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** . “**mytxt** ” is now a tuple because the function returns two values on line 10 in a tuple. When I try to print **mytxt** on line 17, the Console displays an error. Example 7.33 demonstrates this behavior.

```
Ch3_Multiple_Return_Values.py

1  # Ch3_Multiple_Return_Values.py
2  def menu(meal, special=False):
3      msg, msg2 = ""
4      if special is True:
5          msg = 'The specials today are Mimosas.'
6      if meal == 'breakfast':
7          msg = 'Breakfast is eggs and toast.'
8      else:
9          msg = 'Sorry, we ran out of food.'
10     return msg, msg2
11
12
13 mytxt = menu('lunch')
14 if mytxt is None:
15     pass
16 else:
17     print(mytxt)
18
```

Figure 3.12 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 important if you're using it as the argument for another function. In the earlier example, the "print" function expects a string, and my "**menu** " function returns a tuple. In this new example, the Python Interpreter raises an error. 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**]: myTuple = (msg, msg2)
In [**2**]: type(myTuple)
Out [2]: tuple

Recursive Functions

A recursive function calls itself until a statement terminates the function. Typically a recursive call solves problems at a high level, reducing the problem's size at each recursive call, until finally, you reach your base case, which is the simplest form of the problem. The recursive call terminates at your base case. The solution to the "Towers of Hanoi" is a famous example of a recursive problem.

In the next example, the function calls itself recursively on line 5 until **bookcnt** is less than the length of “books” on line 2. If you omit the termination expression on line 2, a **RecursiveError** is eventually raised.

```
def printbooks (books, bookcnt= 0 ):
        while bookcnt < len (books):
                print (books[bookcnt])
                bookcnt += 1
                bookcnt = printbooks(books, bookcnt)
        return bookcnt


books = [‘bk1’ , ‘bk2’ , ‘bk3’ ]
cnt = int (printbooks(books))
print (‘There are %d books in %s’ % (cnt, books))
```

Recursive Memory Stacks

This topic looks at what happens with memory stacks as you make recursive calls.

```
1
2
3 def c(b, bc=0 ):
4         i = '\nStack'
5         j = ' - bc value is'
6         k = "'bc' identifier is "
7         while bc < len (b):
8                 bc += 1
9                 i = i + str (bc + 1 ) + j + str (bc)
10                print (i)
11                print (k, id(bc))
12                bc = c(b, bc)
13        return bc
14 books = ['bk1' , 'bk2' , 'bk3' ]
15 print ('\nStack 1 (global namespace)' )
16 cnt = c(books)
17 print ('\nThere are %d books in %s' % (cnt,
   books))
```

When I run the program, the **Console** outputs the following.

```
Stack 1 (global namespace)
Stack2 - bc value is 1
' b c' identifier is 4485190800
Stack3 - bc value is 2
' b c' identifier is 4485190832
Stack4 - bc value is 3
' b c' identifier is 4485190864
There are 3 books in [ ' bk1 ' , ' bk2 ' , ' bk3 '
]
```

Now let's look at a visual picture of how the Python Interpreter moves up and down the stacks. When the program initially runs, it creates the global namespace or **Stack 1** . Line 16 calls the function **c()** and creates memory **Stack 2** . In memory **Stack 2** , after the program runs line 6, my counter "bc" has a value of 1, and the

identifier for “bc” is 4485190800. On line 10 , the program makes a recursive call to function **c()** and creates memory **Stack 3** .

In memory **Stack 3** , after the program runs line 6 again, “bc” has a value of 2, and the identifier for “bc” is 4485190832. While “bc” has the same name, the identifiers are different, indicating these are two different variables. On line 10, the program makes a recursive call to function **c()** and creates memory **Stack 4** .

In memory **Stack 4** , the program again runs line 6 . Now “bc” has a value of 3, and the identifier is 4485190864. On line 10, the program makes a recursive call to function **c()** and creates memory **Stack 5** , as shown below.

In memory **Stack 5** , when line 5 compares the value of bc to the length of “b,” the Python Interpreter returns False. At this point, bc = 3, and the length of “b” is 3. In **Stack 5** , the Python Interpreter runs line 11 and returns the value of “bc” to the calling function in **Stack 4** , as shown in the next diagram.

line 5: 3 < 3 is False

In **Stack 4** , the Python Interpreter runs line 11 and returns the value of “bc” to the calling function in **Stack 3** .

Next, the Python Interpreter returns “bc” to the calling function in **Stack 2** . Finally, “bc” is returned to **Stack 1** to line 16, the initial function call.

The Zip Function

Earlier, when discussing tuples, we looked at the **zip()** function that takes two or more iterables as arguments and returns a “zip” object that is an iterator of **tuples** . Since we are looking at interesting functions, I wanted to recap that information here. When you pass two lists with three elements each, zip() returns three tuples and pairs the items in the tuples.

The first example uses the **zip()** function with two lists and returns a tuple that is unpacked to the “order” and “color” variables.

```
for order, color in zip ([1 , 2 , 3 ], [‘green’ , ‘red’ , ‘blue’ ]):
    print (order, color, ‘\n’ )
```

The Console prints the following text.

```
In [1
]:
1 green
2 red
3 blue
```

The Map() Function

The map() function takes a unary function and a **data structure** or “iterable” as arguments and returns an iterator of type “map” that applies the function to all items in the **data structure.**

map(<*function* >, <*iterator* >)

In this example, the map applies the function str() to all elements in “mylist.”

```
map (str , mylist)
```

The map() function arguments can also be a function with several arguments. A function “max” with two list arguments that returns a “map” object is shown below.

```
1 list1, list2 = [2, 4, 6 ], [1, 3, 5
2 ]
  map (max , list1, list2)
```

In the next example, I convert the “map” object type to a “list,” so you can see the values returned. The max function is applied to each list item so that 2 > 1, 4 > 3, and 6 >5.

```
In [1 ]: list1, list2 = [2, 4, 6 ], [1, 3, 5
]
In [2 ]: list (map (max , list1, list2))
Out [2 ]: tuple [2, 4, 6 ]
```

This next example uses map to invoke the **f_to_c** function for each element in the **temps_f** list and return a new **temps_c** of the values.

```
1 def gbp_to_usd(temp)
2     return round (temp * .8 , 2 )
3
4
5 usd = []
6 gbp = [6 .70 , 32 .5 1 ]
7 for m in map (gbp_to_usd, gbp):
8     usd.append (m)
9 print(usd)
```

The Console prints the following.

```
In [1 ]:
[5.36, 26.0 1
]
```

Earlier, when we looked at list comprehensions , I compared them to the map() function. Lines 7 and 8 in the previous example could be rewritten as a list comprehension, as shown below on line 7.

```
def gbp_to_usd(temp)
    return round (temp * .8 , 2 )


usd = []
gbp = [6 .70 , 32 .5 1 ]
usd = [gbp_to_usd(temp) for temp in gbp]
print(usd)
```

Lambda Functions

Lambda expressions are used to create anonymous functions. These functions are not bound to a name and typically are simple expressions, often used with the map() or filter() functions.

lambda *parameters* : *expression*

In this lambda expression, I am passing the sort() function the second element [1] in the list “pairs.” The parameter is “m.”

```
pairs = [(1 , ‘Jan’ ), (2 , ‘Feb’ ), (3 , ‘April’ )]
pairs.sort(key=lambda m: m[1 ])
```

The Console prints out the values sorted by the second element.

```
In [2 ]: pairs
Out[ 2 ] : [(3, ‘April’ ), (2, ‘Feb’ ), (1, ‘Jan’ )]
```

In this next example, a lambda expression is combined with the map() function. The lambda parameter is “x” and the expression is **x * 4** .

```
for i in map (lambda x: x * 4 , [1 , 3 , 6 , 7
```

```
2 ]):
        print (i)
```

The Console prints out the value shown below.

```
In [1
]:
4
12
24
28
```

The Filter() Function

The filter() function takes a *function* and a **data structure** (*iterable*) as arguments. Filter() returns items in an iterable (of "filter" object type) for each <*iterator* > where the <f*unction* > returns **True** for that <*iterator* >.

filter(<*function* >, <*iterator* >)

In this example, filter() returns matched items from list1 as a "filter" object type, which I convert to a "list."

```
def myfunc(mystr):
   if mystr in 'Hello':
      return True
   else:
      return False
myvar = ('H' , 'i' )
print (list (filter (myfunc, myvar))
```

The Console prints out the value shown below.

```
['H '
]
```

You can also use a "lambda" expression as your function. In this example, I convert the "filter" type to a "tuple."

```
myvar = filter (lambda x: x > 4 , [1 , 3 , 6 , 7
])
print (tuple (myvar))
```

The Console prints out the value shown below.

```
(6, 7)
```

The Print() Function

Throughout this chapter, you've seen many examples of the print() function. The print() function prints out to the Console and is a handy debugging tool.

The default behavior is to print a line return, but if you provide the "**end** " argument with no value, there is no line return. In this example, the **\n** adds a line feed before 'Hello World."

```
print('\nHello World' , end='')
```

We've seen several examples already where I changed an integer to a string before using the print() function. You can also print integers or floats. This example demonstrates how to print the result of an arithmetic expression.

```
print(2 *3 )
```

The console output is "6 ."

When we looked at recursive functions, you may have noticed the last line printed an integer and strings.

```
print ('There are %d books in %s ' % (cnt,
books))
```

3.25 Classes

This topic provides a brief overview of classes. The **docs.python.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

When working with classes, first you define or create a “class,” then you create an “instance” of the class. In the examples that follow, my class name is “Car,” and my instance of “Car” is “my_car.” I can reuse the class “Car,” creating many instances of “Car.”

Classes implement data abstractions. To work with a class, you don’t need to know the details inside the class or how it gets the job done. You only need to know the data attributes and methods of the class.

You can define a class based on another class. So, for example, I could define a class Convertible(Car.) When you create a class, the superclass is in parenthesis. The class Convertible(Car) is using “Car” as the superclass. “Convertible” is the subclass of “Car.”

The variable “*self* ” refers to any object you create of type “Car.” Continuing the earlier example, “ *self* ” refers to the instance “my_car.” The variable “*self* ” is implicitly passed as the first parameter to class attributes and methods.

Special Methods and Override Behavior

Earlier, we looked at special method names that refer to system-defined or “dunder ” names that begin with two underline characters. Classes inherit these special methods from parent classes but can override that behavior.

For example, in Python, you see **<** in comparisons, but you could override the **__lt__ (less than)** behavior to do something entirely different. “Data attributes” associated with a class definition are called “class variables.” When associated with an instance of a class, they are called instance variables. Sometimes you’ll see a class variable used for “counting” instead of a global variable.

A class definition begins with **def __init__(self)** , as shown below on line 4.

```
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
6      variables."""
7      self.model = model
8      self.make = make
9      self.year = year
10 def drive(self):
11     """Move the car."""
12     print(self.model.title() + " is now moving.")
13 def parallelpark(self):
14     """Parallel park the car."""
15     print(self.model.title() + " is now parking.")
16 my_car = Car('Subaru', 'Crosstrek', 2019)
17 print(my_car.model, my_car.make, my_car.year)
18 my_car.parallelpark()
```

Create a Class

In the previous class example, line 1 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

Continuing with the car class example, 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 raises 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** . However, all instances of a class share class variables and methods. 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 referred to as a "**method** " or "**method attributes** ." 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)
```

Calling a Method

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

```
my_car.parallelpark()
```

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

```
Car.drive(my_car)
my_car.drive()
```

Superclass and Subclass

In this example the class “Family(Person)” is reusing the Person.__ init__(self, name). This makes “name” and “alive” available to the class “Family.”

```
class Person(object ):
   def __init__(self , name):
         self .name = name
         self .alive = True
class Family(Person):
   def __init__(self , name, relationshiop):
         Person.__init__(self , name)
         self .relationship = relationship
```

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

3.26 Modules and Libraries

Variables in Imported Modules

Often modules are broken into separate files or libraries, and you add these to your code with the **import** statement. This idea of breaking code into smaller chunks of code that are easy to debug and reuse independently is known as **modularity** . Typically you don't need to know anything about the internal code in a function or module. All you really care about is the function inputs (arguments), what the function does, and what the function outputs (the return object.) This concept is known as **abstraction** .

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)
```

3.27 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 the example below 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()
```

3.28 Scope, Namespace & Memory

Each time a program runs and creates variables, the Python Interpreter adds the variables to the "global namespace" or "**Stack 1** ." The *global namespace* is the first memory "stack." Variable Explorer displays objects in the current "scope." Variable Explorer is empty until you run the program to create the program's memory "namespace."

The variable "name" and memory "space" (or namespace) uniquely identifies a variable. When you step through your code, the "local scope" or local namespace reflects the objects in memory at that point in time. Scope changes when your code moves into a method or function, and a new "local scope" is created while you're inside that function. Lexical scoping or "static scoping" refers to the

line of code that created a variable and limits the variable to that local scope or namespace.

The LEGB rule refers to the **L** ocal -> **E** nclosing -> **G** lobal -> **B** uilt-in namespaces. Nested functions consist of a “local” namespace located within the “enclosing” function. All functions are stacked onto the “global” namespace, which in turn is stacked on top of Python’s built-in namespace.

In Python, you can read, but not change, the value of a global variable at any point in your program and from within functions, as long as everything is within the **same *.py file** . It is possible to have two variables with the same name and different values because they are in two different “scopes.” You’ll notice this behavior as you step through the code and watch the list of variables in Variable Explorer. When you step into a function, the variable names reflect the “local scope.” Suppose you have variable “x” in the global namespace, and variable “x” is also in a function’s local scope.

If you want to change a variable across “scopes,” you can use the “**global** ” keyword to change the variable into a “global variable .” The keyword “**nonlocal** ” refers to a variable in the “enclosing” namespace. We looked at global variables earlier with an example of how object values change as “scope” changes. In Chapter 7, example 7.50 demonstrates a scope issue.

There are several functions to view scope, and we’ll look at a few in the following pages. As you work with these functions, you’ll see that namespaces in Python are stored in the form of a dictionary.

dir(__builtins__)

locals()

globals()

id()

We looked at memory stacks for recursive functions earlier.

A Function that Accesses a Global Variable

We'll begin looking at namespaces and scope in the example below. The function definition for **blockParty()** begins on line 1, and the function has no arguments. The indented lines that follow through line 3 make up the body of the **blockParty()** function. Lines 1 to 3 are a "suite of code." The main body of the program begins on line 6.

- The "food" variable is defined on line 6 in the main body of the program, which means "food" is in the *global namespace* . Any function in this *.py file can access the global "food." In this example, within the **blockParty()** function, the global variable "food" is accessed on line 3.
- The **blockParty()** function is **not allowed to change** the value in the global "food" variable.

```
1 def blockParty ()
2
3     name = 'John'
4     print (name, 'has' ,
5     food)
6 food = 'chips'
7 blockParty()
```

Variables in the main program are in the *Global Namespace* of the program. The main program begins on line 6. In the diagram that follows this is memory "Stack 1." In order to see the scope changing "live" in Variable Explorer, we're going to debug the file.

1. Type the sample code into the Editor. On the Spyder toolbar, click on **Debug file** (Ctrl+F5).

If it's not already open, open Variable Explorer from the **View** menu by clicking on **Panes** . At this point, there are no variables displayed because we haven't run the program to initialize the variables and assign values. That's about to change!

2. In the Editor pane, line 1 is highlighted . Because line 1 is a function definition, when I click **Run current line** (Ctrl+F10), the Python Interpreter analyzes the function definition and moves to line 6.

In the iPython Console, to the left of line 6 is an arrow ----> indicating the Python Interpreter is about to run line 6.

In the Spyder toolbar, click **Run current line** . Line 6 runs. Now, in the Editor pane, the cursor moves down and highlights line 7. In the iPython Console, the arrow ----> now points to line 7.

Variable Explorer displays the variable “food” with a value of “chips.” Notice the variable is a “str” type.

3. Line 7 invokes the function **blockParty()** . We want to step through the code inside the function. In the Spyder toolbar, click on Step into function or method of current line (Ctrl+F11). Line 7 runs invoking the **blockParty()** function.

If you click **Run current line** instead of “Step into function or method of current line,” *the program runs all lines in the* **blockParty()** *function and exits debug mode*.

In the iPython Console, debug moves the cursor to the function **blockParty()** on line 1. An arrow ----> points to line 1.

The Editor highlights line 1.

In the Spyder toolbar, click **Run current line** to run line 1. The local scope changes to the **blockParty()** function and a new “stack” of memory is created. In the next diagram, this is “**Stack 2** .” Variable Explorer still shows the “food” variable because “food” is in the *global namespace* . The global scope variables can be accessed or “read,” but not changed, within the **blockParty()** function.

4. The previous diagram outlines the *global namespace* in **Stack 1** , and the *local namespace* of the function **blockParty()** 's in **Stack 2** .

In the iPython Console, let's look at the *global namespace* with the built-in function **globals()** . Below, I've abbreviated the output text to show the last line, which is my global variable "food." The interpreter actually prints out a large dictionary for the namespace, as indicated by the curly braces **{}** .

ipdb> globals()

{ .... 'food': 'chips'}

5. Continue stepping through the code. In the function **blockParty()** after line 2 runs, Variable Explorer also shows the "name" variable. This variable is in **Stack 2** or the *local namespace* .

When you run line 3 , the iPython Console output is "John has chips."

6. The locals() function shows all variables in the current or *local namespace* **Stack 2** . In the iPython Console, type "locals()."

ipdb> locals()

{‘name’: ‘John’}

7. Continue stepping through the code. In the function **blockParty()** , after line 3 runs, the Python Interpreter moves down to line 7. When line 7 is highlighted, you are back in the main program or *global namespace* . The *global namespace* , or **Stack 1** , has the “food” variable with a value of ‘chips.’

Variable Explorer no longer shows the “name” variable because **Stack 2** is discarded when you exit the “**blockParty()** ” function.

A Function Variable with the Same Name as a Global Variable

The previous example is pretty straightforward. Now we are going to repeat the same steps with a small variation. This time we add a “food” variable assignment on line 3 within the function “**blockParty()** .”

As we step through the code, you’ll see two variables named “food,” but with different values and identifiers.

```
1 def blockParty ()
2
3       name = ‘John’
4
5       food = ‘salsa’
6
7       print (name, ‘has’ ,
        food)
8 food = ‘chips’
  blockParty()
```

The main program begins on line 7. and the “food” variable from line 7 is in the *Global Namespace* of the program. The diagram in Step 4 that follows shows this as memory “**Stack 1** .”

Now we’ll debug the file.

1. Type the sample code into the Editor. On the Spyder toolbar, click on **Debug file** (Ctrl+F5).

If it’s not already open, open Variable Explorer from the **View** menu by clicking on **Panes** .

2. In the Editor pane, line 1 is highlighted . Because line 1 is a function definition, when you click **Run current line** (Ctrl+F10) the Python Interpreter evaluates the function definition and moves on to line 7.

In the iPython Console, to the left of line 7 is an arrow ----> indicating the Python Interpreter is about to run line 7.

In the Spyder toolbar, click **Run current line** . Line 7 runs. The program is still in the “*global namespace* ” or “**Stack 1** .”

In the Editor pane, debug moves down and highlights line 8. The iPython Console displays an arrow next to line 8, as shown in the next diagram.

3. Line 8 invokes the function **blockParty()** . We want to step through the code inside the function. In the Spyder toolbar, click on Step into function or method of current line (Ctrl+F11). Line 8 runs invoking the **blockParty()** function.

In the iPython Console, debug moves the cursor to the function **blockParty()** on line 1. An arrow ----> points to line 1. In the Editor, line 1 is also highlighted. In the Spyder toolbar, click **Run current line** to run line 1.

The local scope changes to the **blockParty()** function, and a new “stack” of memory is created. This is “**Stack 2** .” Variable Explorer still shows the “food” variable with a value of “chips” from the *global namespace* .

4. Continue stepping through the code. In the function **blockParty()** , after line 2 runs, Variable Explorer shows the “name” and “food” variables. The “food” variable is from the *global namespace* .

Stack 2 now has two variables: the local variable “name” and the global variable “food.”

5. When you run line 3, a new variable called “food” is created in “**Stack 2** ” with a different value of “salsa,” as shown below in Variable Explorer.

This local variable in **Stack 2** has the same name as the global variable but points to a different location in memory - in this case, **Stack 2** ’s “food” with the value “salsa.”

6. This time, the locals() function shows two variables in the current or *local namespace* , which is **Stack 2** . In the iPython Console, type “locals().”

ipdb> locals()

{‘name’: ‘John’, ‘food’: ‘salsa’}

Looking at the printout in the iPython Console, you can tell the namespace variables are stored in a dictionary because of the **curly braces {}** . The program is within the **Stack 2** *local namespace* of **blockParty()** . If I were to update the local variable “food,” the other “food” variable in the outer scope or *global namespace* would not change.

To see the identifier of the objects, let’s use the id() function to get the ID for the **Stack 2** “food” variable. Because line 4 is highlighted in the Editor, I know I am still in the *local namespace* of the blockParty() function. Your computer output for ID will be different than what is shown below.

ipdb> id (food)

140309822725104

7. Continue stepping through the code. After line 4 runs, the code moves down to line 7. When line 7 is highlighted, you are back in the *global namespace* or **Stack 1** .

Variable Explorer no longer shows the “name” variable because Stack 2 is discarded when you exit the “blockParty()” function. Also, you’ll notice the value of the “food” variable changed back to “chips.” The program is back in the *global namespace* or “Stack 1.

8. In the iPython Console, use the id() function to view the identifier for this “food” variable in the *global namespace* . This output will show a different ID number.

ipdb> id(food)

140309822723824

Scope in Nested Functions

Earlier we looked at the LEGB rule for **L** ocal -> **E** nclosing -> **G** lobal -> **B** uilt-in namespaces. Nested functions consist of a

function's "local" namespace located within the "enclosing" function. To illustrate the "enclosing namespace," the next example has a nested function, "**games()** ."

```
1
2
3 def blockParty ()
4     def games( )
5         game = 'darts'
6         print (game, 'starts at
7         noon' )
8     name = 'John'
9     food = 'salsa'
10    print (name, 'has' , food)
11    games()
12
13 food = 'chips'
14 blockParty()
```

The scope of the function **blockParty()** is now in the "enclosing namespace" and is **Stack 2** . The function **games()** is nested inside **blockParty()** and is **Stack 3** .

Earlier I said the "global" keyword is used when you want to update a global variable from within a function. Similarly, the nested function **games()** can update "name" in **blockparty()'s** "enclosing" namespace if you add a statement on line 3 with the keyword "nonlocal."

```
1 def blockParty ()
```

```
2       def games( )
3           nonlocal name
4           game = 'darts'
5           print (game, 'starts at
6           noon' )
7       name = 'John'
8       food = 'salsa'
9       print (name, 'has' , food)
10      games()
11  food = 'chips'
12  blockParty()
13
```

Finally, let's add another function to the code and review scope. In this example, the **events()** function definition is on line 11 in the main body of the program.

```
1
2
3
4
5   def blockParty ()
6       def games( )
7           game = 'darts'
8           print (game, 'starts at
9           noon' )
10
11      name = 'John'
12      food = 'salsa'
13      print (name, 'has' , food)
14      games()
15  def events ()
16      type = 'party'
17      print (type)
18  food = 'chips'
    blockParty()
    events()
```

In this case, the main program has two paths. The functions blockParty() and games() have no access to the variables in the

events() function, and vice versa

When you're analyzing scopes, diagram the path through the program to determine namespaces.

4. Debugging Tools

In this chapter we discuss

Debugging Overview
Print Statements
Debug Mode
Variable Explorer
Example: Program Loops & Never Ends
Debug Commands
Console Interactive Mode
Introspection
Logging
The timeit() Function
Logging Time and Loop Counters
Focused Testing
Create Test Data

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. In the topic “Introspection” that follows, we’ll also touch on the inspect library. 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.

1. Inspect Objects and Variables
2. Add Print Statements to a Script
3. Debug Mode
4. Variable Explorer
5. Interactive Mode

1. 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, and to locate where a variable is used or changed.

2. 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.

3. 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.

4. 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.

5. 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 already ran the line of code that **creates the object** .

4.2 Print Statements

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.

While it’s not exactly elegant, I frequently add a print statement to my code that simply says something like, “about to have a problem.” Then I add a debug “**breakpoint** ” at that location. 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.

```
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[j])
              print (“j is: ”, j)
              j = j + 1
        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 helpful, in this example, 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 7.15 uses print statements with exception handling logic. Example 7.35 uses print statements to debug a generator script. 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.

```
meals = [‘breakfast’, ‘lunch’, ‘snack’, ‘dinner’
]
fruits = [‘apple’, ‘orange’, ‘grape’]
i = 0
level = ““
while i < 4:
        level = level + “.... ”
        print(“my meal is: ”, fmeals[i])
        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)

```
 |  ----------------------------------------------------------------------
 |  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 4.3 Console Display of Object Attributes

> In the Editor, position your cursor over “thedate” and wait a few seconds. All instances of “thedate” variable are highlighted. This is a great way to find when a variable is used.

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 below displays the oldest date in the list “thedates.”

Figure 4.4 The max() Function

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,” as demonstrated in Examples 7.2, 7.20 and 7.52. 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 to create the program’s memory “namespace .”

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 **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

As variables are created in your main program they are added to the “global scope” or namespace. The global scope is the first memory “stack.” Variable Explorer displays objects in the current “scope” and is empty untiil you run the code to create the program’s memory “namespace.”

At this point in our example, **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. If you’re unsure if the variables are in scope, you could use the **locals()** function to see local variables.

To start fresh and clear program memory, in the Console use the magic function **%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.
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** .

Figure 4.7 Add a Breakpoint

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 line 9 and then looping back to line 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 7.1 demonstrates an infinite loop.

4.7 Debug Commands

In Debug Mode, type **?** in the **Console** 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 interactive interpreter displays the value in the **Console** , as shown below.

In [1]: mystring
Out[1]: ‘ purple peanut s’

> 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 special identifier _ underscore is used in the interactive interpreter (or Console) to store the result of the last evaluation.

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, 2, and 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 14 reads a TXT file with the OS library. The function **readline()** moves to the next line of the TXT file 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 Variables and Objects in Memory

The Python Interpreter creates variables and objects when you run a program or script. The collection of these objects 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 for the current scope . 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.

4.10 Introspection

Introspection is the ability to determine information about live objects such as modules, classes, methods, functions and tracebacks. You can easily tell the type of the object at runtime. Several functions help with introspection, as well as the “ **inspect** ” library.

objectname **?**

dir()

help()

id()

repr()

type()

locals()

globals()

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)

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
```

This next example returns help for the max() function. You may want to compare this output to the __doc__ attribute later in this section.

```
In [ 5 ]: max?
Docstring:
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value
With a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty.
With two or more arguments, return the largest argument.
Type: builtin_function_or_method
```

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()

In Chapter 7, Example 7.38 uses “dir()” to debug an AttributeError.

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 **openpyxl** , “**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, or 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. Example 7.19 uses the help() function. Example 7.21 invokes help with the shortcut Cntrl + I

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. The next example uses the **signature()** function in the **Console** .

```
In [ 2 ]: from inspect import signature
In [ 3 ]: from openpyxl import load_workbook
In [ 4 ]: str(signature(load_workbook))
(filename, read_only=False, keep_vba=False, data_only=False,
keep_links=True)
```

What Version of Python?

The **version_info** attribute displays the current version of Python, as shown below.

```
In [ 5 ]: import sys
In [ 6 ]: sys.version_info
Out[ 7 ]: sys.version_info(major=3, minor=7, micro=4, releaselevel= '
fina l' , serial=0)
```

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. Example 7.45 demonstrates a float comparison error.

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.

The locals() Function

You may recall from Chapter 3, the locals() function shows all objects in the current or local namespace . In the iPython Console, type “locals().”

```
In [ 5 ]: locals()
```

The globals() Function

The globals() function shows all objects in the current or global namespace . In the iPython Console, type “globals().”

```
In [ 5 ]: globals ()
```

The docstring

The **__doc__** attribute prints a function's docstring if it exists. A *docstring* might cover these topics.

- What the function does.
- What type of arguments the function takes.
- What the function returns, if anything.

This next example is the docstring for the max() function. Earlier we used **?** to see information on the max() function.

```
In [5]: max.__doc__
Out[5]: ' max(iterable, *[, default=obj, key=func]) -> value\nmax(arg1, arg2, *args, *[, key=func]) -> value\n\nWith a single iterable argument, return its biggest item. The\ndefault keyword-only argument specifies an object to return if\nthe provided iterable is empty.\nWith two or more arguments, return the largest argument. '
```

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

```
from timeit import timeit
mycode = '''
colors = ('blue', 'red', 'green')
for color in colors:
print(color)
'''
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 75 million comparisons makes me very happy I use Python!

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: -0.00030199999999995804
i is 50
Time so far is: -0.0014359999999999928
i is 2050
Time so far is: -0.0018299999999999983
i is 3050
Time so far is: -0.0022180000000009414
i is 4050
Time so far is: -0.002792000000001238
i is 5050
Time so far is: -0.0032180000000003872
i is 6050
Time so far is: -0.0036020000000007713
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. I also might make a very simple “function” that I can test and debug, and later merge it into a library. 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

Test Objects

In Chapter 2 we talked about focused testing, and a simple way to create test objects. I wanted to recap that information again because it's such a time saver.

For testing purposes, I only need **mydictionary** with a few elements. To create a dictionary for testing purposes, first, I run the main program that creates **mydictionary** . Then, in the Console, I type **mydictionary** . The Console prints out all the values, so I can copy a smaller subset of data for a test version of **mydictionary** . In my test script, I use the data in an assignment statement for the "test" dictionary.

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 example demonstrates a sample **Console** Traceback message. I abbreviated the file path for readability.

```
In [1]: 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 7.1, 7.2, 7.3, and 7.6 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 add handlers to control how exceptions are handled, and prevent your program from unexpectedly halting. There are four blocks to a try:except statement. The “else” also runs only if there is no exception. The “finally” block always runs, and is often used to close files and cleanup memory.

```
try:
except Exception:
else:
finally:
```

In this next example with “except,” you can see where I added custom messages.

```
try :
        gbpex = float (tables2[3 ].string[:6 ])
1       gbp = gbp/gbpex
2       print (“gbp converted to USD is:” , gbp)
3
4 except TypeError:
5       print (“Type error when converting exchange
6       rate” )
7
8 except ZeroDivisionError:
9       print (‘ZeroDivisionError where gbpex is:’ ,
1       gbpex)
0
1 except Exception as exceptdetails:
1       print (exceptdetails, ‘gbpex is:’ , gbpex)
1
2               finally:
        print (“Done calculating the gbp exchange
        rate.” )
```

The **finally** clause on line 11 runs whether the **try** clause has an exception or not. You can also raise your own type of error, as shown on line 4.

```
try
    <some code>
except Exception:
    raise ValueError(mymsg)
```

Examples 7.15, 7.21, and 7.24 demonstrate the try and except syntax.

5.4 Raise

At any point in your program, 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

With an “**assert** ” statement program execution stops or halts when an expected condition is not met. 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)
- No duplicates
- A string is not None

> In debugging when your program stops an assert statement, you know the “bug” must be somewhere before the assertion.

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
1 thedate = datetime.datetime.now()
2 assert type (thedate) == datetime.datetime, “this isn’t a
3 date”
4
5 if thedate is not None and type (thedate) == datetime.datetime
      print (‘everything is ok’)
```

With a small modification on line 2 “thedate” becomes an “int.” Now “thedate” is not a datetime type, and the assertion fails.

```
  import datetime
1 thedate = datetime.datetime.today().day-1
2 assert type (thedate) == datetime.datetime, “this isn’t a
3 date”
4
5 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.

The “AttributeError ‘list’ object ‘extend’ is read only ” means you forgot the parentheses in the function call.

mylist.extend()

Examples 6, 38 and 39 demonstrate an AttributeError. 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.

For example, beyond the limitis of the list. I have four objects in the list and use index “5” or mylist[4]. Python starts counting at 0, so this example is beyond the bounds of the list.

If you try to assign a value to an index that doesn’t exist, it causes an error.

IndexError list assignment index out of range.

Instead, to add add items to a list use .insert(), .extend(), or .append().

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 7.20 that demonstrates this type of error.

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

You may have noticed in many examples I say, “run the program to create variables, then...” If an object doesn’t exist in the namespace it means the variable hasn’t been declared. The NameError is 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 highlights 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

The OverflowError is 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 similar to an IndexError where the index is out of bounds. A recursive loop counts down to the end of the loop as shown in the Chapter 3 topic, “Recursive Functions .”. A RecursionError is raised when the maximum recursion depth is exceeded.

RuntimeError

A “RuntimeError” is 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 it can’t parse the program.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
- divide init by a string (3/'a')
- mixing types
- missing a function argument

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])
```

UnboundLocalError

An UnboundLocalError is raised when you assign a value to a variable that doesn’t exist in the program scope. In Chapter 3, in the topic “global variables ” we looked at an example of this type of error.

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.

If an index[1] expected a number and you passed a string you'll get a ValueError.

ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The ZeroDivisionError is a subclass of ArithmeticError. Example 7-24 shows a divide by zero error.

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

```
mystring = "purple peanuts"
print(mystring)
```

```
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 element 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
3
4
print(mytuple)
```

```
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
2   print(mytuple[0 ])
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
Refe rence: 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 object “mytuple,” type “mytuple” in the **Console** . The first element in the tuple has an index value of **0** .

A Tuple Item
Identifier: mytuple[**0**]
Value: Name: Apple
Refer ence: 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 mylist = [‘soda’, ‘water’, ‘coffee’]print(mylist)
3
```

```
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
```

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

```
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** .

Figur
e 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' :
2 'Zimmerman' ,
3     'Grade' : 'A' ,
4     'Course' : 'Python
5     Programming' }
print(dictionary['Name' ])
```

```
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 passing arguments to functions that expect a particular type or value for an object. For example, the DateTime library function “**.strftime** ” expects an argument of type “datetime”, “time”, or “date.” If you pass an object that has a value of “None” to the “**.strftime** ” function, 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 Characters

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 string 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.

To identify whitespace characters you can use the repr() function as outlined in the Chapter 3 topic, “Whitespace Characters.” Example 7.29 demonstrates whitespace errors.

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** .

```
from inspect import signature
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__** ."

```
from inspect import *
def myfunction(a: int, b: float):
'''
this function takes an int and a
float
and doesn't do anything
'''
pass
from inspect import *
print (getfullargspec(myfunction))
print (signature(myfunction))
```

```
print (myfunction.__doc__)
```

6.10 What Type of Object Does a Function Return?

In Chapter 3, we looked at "functions that return a tuple " and "function return or yield objects ." 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
1	List index out of range	IndexError
2	List index out of range (Example 1 continued)	IndexError
3	Wrong Variable Name	NameError
4	Invalid assignment	Runtime
5	While statement not indented	IndentationError
6	Method arguments incorrect	AttributeError
7	Empty block is illegal	IndentationError
8	Parentheses not matched	SyntaxError
9	Colon misssing	SyntaxError
10	Case sensitive	NameError
11	Keyword missing	SyntaxError
12	Illegal characters or keyword	SyntaxError
13	Misspelled identifier	NameError
14	File doesn’t exist	FileNotFoundError

Ex	Description	Built-in Error Type
15	Adding incorrect types	TypeError
16	Misspelled keyword	SyntaxError
17	Value is none	TypeError
18	Module not found	AttributeError
19	Module not found	NameError
20	Key not in dictionary	KeyError
21	Arugment is incorrect type	ValueError
22	Object not found - NameError	NameError
23	Invalid data passed to method	ValueError
24	Calculation causes a ZeroDivisionError	ZeroDivision Error
25	There is a mistake in a math calculation	
26	Assigning datetime value causes ValueError	ValueError
27	Matching strings NoneType error	NoneTypeError
28	Matching strings fails	TypeError
29	Whitespace or special characters	
30	Debug: Step through your function	
31	Key not in dictionary	KeyError
32	‘2’ + 2 Concatenate fails	TypeError
33	Function returns 2 values	TypeError

Ex	Description	Built-in Error Type
34	Unsupported operand	
35	Generator object not subsciptable	
36	Missing 1 required positional argument	TypeError
37	Reserved keyword	TypeError
38	Invoking a class method raises an error	AttributeError

Ex	Description	Built-in Error Type
39	Key error creating a dictionary key:pair.	AttributeError
40	Illegal target for annotation	
41	Too many values to unpack	ValueError
42	Tuple object does not support item assignment	TypeError
43a	An object is not callable	TypeError
44	Can only concatenate tuple not ‘str’	TypeError
45	Float comparison fails	
46	Unhashable type: ‘dict’	TypeError
47	‘builtin_function_or_method’ object is not subscriptable	TypeError
48	String comparison error	TypeError
49	Invalid literal for int() with base 10	ValueError
50	A variable is referenced before assignment	UnboundLocalError
51	Plot - shape mismatch	ValueError
52	Unpacking operator	IndexError

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.

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.

```
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 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

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 ]
3
```

```
4 fruits = [‘apple’ , ‘orange’ , ‘grape’ ]
5 i = 0
6 while i < 4 :
7         j = 0
8         print(“my meal is: “ , meals[i])
9         while j < 4 :
10                print (“My choice of fruit is: “, fruits[i])
                  j = j + 1
          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 “list index out of range.” 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 ]
3 fruits = [‘apple’ , ‘orange’ , ‘grape’ ]
4 i = 0
5 while i < 4 :
6
7
```

```
8        j = 0
9        print("my meal is: " , meals[i])
1        while j < 3:
0                print (“My choice of fruit is: “, fruits[j])
                 j = j + 1
         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

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: list index out of range.

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
Chapter 4 - Debug Mode Variable Explorer
Chapter 6 - Check Object Type

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.

```
1 mylist = [‘soda’ , ‘water’ , ‘coffee’ ]
2 i = 0
3 while i < 3:
4
5         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’
1]
2i = 0
3
4while i < 3:
5        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 Loop t

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 error. There is an arrow pointing to “t” in the word “print” on line 4.

IndentationError: expected an indented block

Incorrect Code

This is the Example 5 code before any changes.

```
wadofcash = [111, 222, 333, 44 4
1]
2i = 0
3
4x = 3
5while i <= x:
6print(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

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.

AttributeError: ‘int’ object has no attribute ‘upper’

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.

```
1 from openpyxl import load_workbook
2
3 wb1 = load_workbook(‘ Before.xls x’ ,
4 data_only=True)
5 ws1 = wb1[“ExportedData” ]
6
7 bfr = 2
8 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 atttribute “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
1
2 wb1 = load_workbook(' Before.xls x' ,
3 data_only=True)
4 ws1 = wb1["ExportedData" ]
5
6 bfr = 2
7 while bfr <= ws1.max_row:
8
9     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

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.

IndentationError: expected an indented block

Incorrect Code

This is the Example 7 code before any changes.

```
for mynum in [157, 19, 5 6 ]:

    if mynum == 157 :
```

```
4     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, 5 6 ]:
1     if mynum == 157 :
2
3           pass
4     else:
5           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

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. There is an arrow highlighting the location where the parentheses is missing.

SyntaxError: invalid syntax

Incorrect Code

This is the Example 8 code before any changes.

```
1 sales = 150.00
2 days = 31
3 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

```
1 sales = 150.00
2 days = 31
3 projectedsales = ((sales/days)*31
  )
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - SyntaxError

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. An arrow points to the location of the arrow on line 2.

SyntaxError: invalid syntax

Incorrect Code

This is the Example 9 code before any changes.

```
1 mylist = [‘soda’ , ‘water’ , ‘coffee’
2 ]
3 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’
1 ]
2 for i in range (3) :
3         print (mylist[i])
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Help()
Chapter 5 - SyntaxError

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 on line 3.

NameError: name ‘mylist’ is not defined

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

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.

SyntaxError: invalid syntax

Figure 7.3 Function with Error

Incorrect Code

This is the Example 11 code before any changes.

```
1 myfunction(x,
2 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

```
1 def myfunction(x,
2 y):
    return x+y
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 5 - SyntaxError

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. When I run the program, the Console displays an arrow pointing to the invalid character in “my$str.”

```
Ex_12.py” , line 1
    my$str = ‘hello’
    ^
SyntaxError: invalid
syntax
In [ 2 ]:
```

Incorrect Code

This is the Example 12 code before any changes.

```
1 my$str =
2 ‘hello’
  print (my$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.

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 =
1 ‘hello’
2 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

7.13 Undefined Name

Description : Undefined name when variable is misspelled. Traceback shows a NameError.

Intended Outcome

Line 2 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.

NameError: name ‘royaltie’ is not defined

Incorrect Code

This is the Example 13 code before any changes.

```
1 royalties = 300
2 profit = royaltie - 25
```

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

```
1 royalties = 300
2
```

```
profit = royalties -
25
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 5 - NameError

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."

FileNotFoundError: [Errno 2] No such file or directory: 'file.txt'

Incorrect Code

This is the Example 14 code before any changes. This syntax would work for a .txt or .csv file.

```
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.tx t’ , ‘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)

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 my custom error message if there is another type of Exception. Chapter 4 has more information on "try" and "except."

```
Type error when adding; eur euro ; usd
3.45
```

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' ), eur, ';
    usd' , usd)
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; eu r' , eur, ' ; us d' ,
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.”

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 - Try and Except
Chapter 5 - TypeError

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 3. When I run the program it halts with a “SyntaxError.” There is an arrow pointing to the end of line 3.

SyntaxError: invalid syntax

Incorrect Code

This is the Example 16 code before any changes.

```
1 if 4 < 5:
2     pass
3 esle:
4     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

```
1 if 4 < 5:
2     pass
3 else:
4     print(“Python
      rocks”)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Reserved Keywords
Chapter 3 - Keyword Arguments
Chapter 4 - Help()
Chapter 5 - Traceback
Chapter 5 - NameError
Chapter 5 - SyntaxError

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** .

TypeError: unsupported operand type(s) for /: ‘NoneType’ and ‘int’

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 3 - NoneType
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Does the Object have a Value of NoneType or Whitespace?
Chapter 6 - Check Object Type
Chapter 6 - Value is None

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

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

NameError: name ‘plt’ is not defined

Incorrect Code

This is the Example 19 code before any changes.

```
1 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

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 3 - Import Library
Chapter 4 - Help()
Chapter 5 - Traceback
Chapter 5 - ModuleNotFound

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 7.4 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.xls x’ , data_only=True
)
ws = wb[“Sheet1” ]
ft = styles.Fontt(color=’ 4F81B D’ , 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 with a KeyError, which points to a dictionary. 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
    1 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.

In [**2**]: myStr

Out [**1**]: 'hello'

Figure 7.5 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.

In [**2**]: myStr

Out [**1**]: 'hello'

Figure 7.6 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 7.7 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.

```
1 # 20_CORRECT_KeyError.py
2 from openpyxl import load_workbook, styles
3 wb = load_workbook(' before.xls x' , data_only=True
4 )
5 ws = wb["ExportedData" ]
6 ft = styles.Fontt(color=' 4F81B D' , 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
```

```
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

7.21 Incorrect Argument Type

Description: A Function Arguments is an incorrect type causing a ValueError.

Intended Outcome

The program asks for a month as user 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 it prompts for user input. If I enter the string “July” for the input, a ValueError is displayed in the Console . A ValueError is raised when a function gets an argument of correct type but improper value.

ValueError: invalid literal for int() with base 10: ‘July’

Incorrect Code

This is the Example 21 code before any changes.

```
1 birthmo = int (input (‘ what month were you born?’
2 ))
3 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 3 - Function Arguments
Chapter 5 - Traceback
Chapter 5 - Try and Except
Chapter 5 - ValueError
Chapter 6 - Check Object Type

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 histogram chart.

Actual Result

When I run the code, the Console displays a NameError and highlights line 4.

```
In [3 ]: ..Ex_22_WRONG.py" , line 4 , in <module>
   plt.plot([1, 2, 3, 4], [25, 30, 29, 31])
NameError: name 'plt' is not defined
```

Incorrect Code

This is the Example 22 code before any changes.

```
1 import matplotlib.pyplot as plot
2
3 myL = [13.7, 4.1, 14.4, 23.3, 27, 22, 31.99, 36. 2
4 ]
5 bins = [0, 11, 21, 31, 4 0 ]
6 plt.hist(myL, bins, rwidth=0.5) # histogram graph
7 plt.xticks([10, 20, 30, 40 ])
8
9 plt.xlabel('age' )
1 plt.ylabel('participants' )
0 plt.show()
```

Debugging Steps

The NameError indicates the object can't be found. In the Editor, if I hover over the word "plot" on line 1, I can see there are no other instances of "plot." When I imported the library on line 1, I used "plot," and on the lines 6-10 I used "plt."

How to Resolve the Issue

Update line 1 to use "plt."

Good Code

```
import matplotlib.pyplot as plt
myL = [13.7, 4.1, 14.4, 23.3, 27, 22, 31.99, 36. 2
1 ]
2 bins = [0, 11, 21, 31, 4 0 ]
3 plt.hist(myL, bins, rwidth=0.5) # histogram graph
4 plt.xticks([10, 20, 30, 40 ])
5 plt.xlabel('age' )
plt.ylabel('participants' )
plt.show()
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - NameError

7.23 Value Error

Descriptio n: 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.

```
1 fruits = ['apple' , 'orange' , 'grape' ]
2
3 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 "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

```
1 fruits = ['apple' , 'orange' , 'grape' ]
2
```

```
3 myfruit = 'orange'
  fruits.remove(myfruit)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Lists
Chapter 3 - remove
Chapter 4 - Interactive Mode
Chapter 4 - Help()
Chapter 5 - ValueError
Chapter 5 - RuntimeError

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

ZeroDivisionError: float divison by zero

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.co m'
)
soup2 = BeautifulSoup(html2, 'lxml' )
tables2 = soup2.findChildren(' t d' )
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 try and except 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.co m' )
soup2 = BeautifulSoup(html2, 'lxml' )
tables2 = soup2.findChildren(' t d' )
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 - Try and Except
Chapter 5 - ZeroDivisionError

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)
```

7.26 ValueError Assigning Date

Description: This is an example of an operation on incompatible types. There is a 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.

ValueError: month must be in 1..12

Incorrect Code

This is the Example 26 code before any changes.

```
1 from datetime import datetime
2 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 2. It's obvious there is no month "13" and the statement on line 2 is invalid.

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 2 needs updated to use "12" for the month instead of "13."

Good Code

```
1 from datetime import datetime
2 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

7.27 Matching Strings NoneType Error

Description: This is the first example looking at a comparison expression. I am comparing two strings inside two list objects.

While I have accounted for object types and case, this code has another problem that may not be obvious. When iterating through container objects, sometimes you are not sure what values are stored in each item. Take for example a large XML file that contains a series of nodes. If you write code to iterate through the XML nodes, you may encounter a node without a value. Python assigns this type of object to 'NoneType .' NoneType is the special Python type that indicates there is no value.

Intended Outcome

The expression comparing two string objects should evaluate to "true" and print the message on line 9 when there is a match.

Actual Result

When the program runs, it prints "no match" three times.

Incorrect Code

This is the Example 27 code before any changes.

```
1 s1 = [['history' ], ['math' ], ['social studies'
2
```

```
]]
s2 = ['Math' , None , 'Social Studies' ]
for c in range (0 , 3 ):
        course1, course2 = '' , ''
        course1 = s1[c]
        course2 = s2[c]
        if course1 == course2:
                print ('Repeating class: ' ,
                course2)
        else :
                print ("no match" )
```

Debugging Steps

1. In the **Console** type the following comparison statement to see if the Python Interpreter returns 'True' or 'False.'

When you type s2[2] and s1[2] the Python Interpreter outputs a string and a list,as shown below.

```
In [2 ]: s1[2 ] == s2[2 ]
Out[ 2 ]: False
In [3 ]: s2[2 ]
Out[ 3 ]: ' Social Studie s'
In [4 ]: s1[2 ]
Out[ 4 ]: [' social studie s'
]
```

2. In Variable Explorer, notice the values for the third element in S1 and S2 are different. The type of object is also shown.

How to Resolve the Issue

On line 6 I added try and except logic. I convert the variable to a string, and use the **.upper()** method to change the case. I use slicing **[2:-2]** to remove the square brackets and apostrophes on line 6 after I convert the list to a string.

Good Code

```
s1 = [['history'], ['math'], ['social studies']]
s2 = ['Math', None, 'Social Studies']
for c in range(0, 3):
    course1, course2 = '', ''
    try:
        course1 = str(s1[c])[2:-2].upper()
    except Exception:
        course1 = ''
    try:
        course2 = s2[c].upper
    except Exception:
        course1 = ''
    if course1 == course2:
        print('Repeating class: ', course2)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - NoneType
Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type

Chapter 6 - Value is None

Chapter 6 - Does the Object have a Value of NoneType or Whitespace?

Chapter 6 - Object Value

7.28 Matching Strings Fails

Description: Let's continue with example 7.27. While I have accounted for object types and case, this code has another problem that may not be obvious. When iterating through container objects, sometimes you are not sure what values are stored in each item. Take for example a large XML file that contains a series of nodes. If you write code to iterate through the XML nodes, you may encounter a node without a value. Python assigns this type of object to 'NoneType .' NoneType is the special Python type that indicates there is no value.

Intended Outcome

The expression comparing two string objects should evaluate to "true."

Actual Result

When the program runs, now it halts with a traceback message.

AttributeError: 'NoneType' object has no attribute 'upper'.

Incorrect Code

This is the Example 28 code before any changes.

```
s1 = [['history' ], ['math' ], ['social studies'
]]
```

```
4 s2 = [‘Math’ , None , ‘Social Studies’ ]
5
6 for c in range (0 , 3 ):
7         course1 = str (s1[c])[2 :-2 ].upper()
          course2 = s2[c].upper
          if course1 == course2:
                print (‘Repeating class: ’ ,
                course2)
```

Debugging Steps

The Traceback shows and error on line 5 where I am converting the s2 element to uppercase.

1. In Variable Explorer, check the values of the second element are the same. The second element in list “s2” has a value of “None.” In the Console, I check the type of s2[1].

```
In [5 ]: type(s2[1 ])
Out[ 5 ]:
NoneType
```

How to Resolve the Issue

1. Since the “NoneType” isn’t a string the .upper method raises an error. I’ll add try and except logic beginning on line 5. When an exception occurs, the variable is assigned a value of ‘’.

Good Code

```
1 s1 = [[‘history’ ], [‘math’ ], [‘social studies’
2 ]]
3
4 s2 = [‘Math’ , None , ‘Social Studies’ ]
5 for c in range (0 , 3 ):
6
7         course1, course2 = ‘’ , ‘’
8         try:
```

```
        course1 = str (s1[c])[2 :-2 ].upper()
    except Exception:
        course1 = ''
    try:
        course2 = s2[c].upper
    except Exception:
        course1 = ''
    if course1 == course2:
        print ('Repeating class: ' ,
        course2)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - NoneType
Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type
Chapter 6 - Object Value

7.29 Whitespace or Special Characters

Description: This example demonstrates a whitespace line feed character. When comparing strings, you may have to remove unseen whitespace characters. This is not a real world example but does show the 'repr' function that prints hidden whitespace characters.

Intended Outcome

This example should print 'hello world' in the Console.

Actual Result

When the program runs, the Python Interpreter prints ‘hello world\n.’

Incorrect Code

This is the Example 29 code before any changes.

```
In [4 ]: mystr = ‘ hello world\
n’
In [5 ]: print(repr(mystr))
Out[ 5 ]: ‘ hello world\n’
```

Debugging Steps

It’s not obvious in Variable Explorer that there is a paragraph return at the end of “mystr.” The Console displays “\n” at the end of the string.

How to Resolve the Issue

The .rstrip() function removes the whitespace character from the right side of the string.

Good Code

```
In [4 ]: mystr = ‘ hello world\
n’
In [5 ]: mystr = mystr.rstrip()
In [6 ]: print(repr(mystr))
Out[ 6 ]: ‘ hello worl d’
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - NoneType
Chapter 3 - Does the Object have a Value of NoneType or Whitespace?
Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type
Chapter 6 - Object Value

7.30 Debug: Step Through Your Function

Description: The program halts and the exception is an AttributeError. I want to step through the program in debug mode and look at variable values inside my function. I start debug mode and click on “Run current line.” The program runs line 3, line 6 and skips down to line 21, followed by line 22, and finally line 23. The Python Interpreter skips over my function code.

Intended Outcome

My function takes a birth date and name and prints out ‘Happy Birthday” if today is the birthday.

Actual Result

When the program runs it halts with an exception and the Traceback message is an AttributeError.

AttributeError: ‘builtin_function_or_method’ object has no attribute ‘month’

Incorrect Code

This is the Example 30 code before any changes.

```
1
2
3
4
5 import datetime
6 def happybirthday(dob, name):
7     thetype = type (dob)
8     thedate = datetime.datetime.now
9     month = thedate.month
10    day = thedate.day
11    if thetype == datetime.datetime:
12        dobmonth = dob.month
13        dobday = dob.day
14        if dobmonth == month and
15        dobday == day:
16            print( ‘Happy Birthday’,
17            name)
18        else:
19            print( ‘Please enter a
20            birthday’)
name = ‘Mary Lee’
dob = datetime.datetime(1976, 6, 15 )
happybirthday(dob, name)
```

Debugging Steps

The Traceback shows and error on line 6. When I start debug mode, I choose “Run current line” until line 20 is highlighted. At this point, I use “Run until current function or method returns” to **step into** my function code.

Now the debugger moves to line 4, and I continue to click on “run current line” until line 6 when the AttributeError occurs. In the Console window I type “**type(thedate)** ” to see the object type for “thedate.”

ipdb> **type(thedate)**

How to Resolve the Issue

Although I was expecting a **datetime.datetime** type, the Console returns builtin_function_or_method. When I search the internet for help on this datetime “now” method I find the statement should end in **.now()** , as shown below.

thedate = datetime.datetime.now()

Good Code

```
import datetime
def happybirthday(dob, name):
    thetype = type (dob)
    thedate = datetime.datetime.now()
    month = thedate.month
    day = thedate.day
    if thetype == datetime.datetime:
        dobmonth = dob.month
        dobday = dob.day
        if dobmonth == month and
        dobday == day:
            print( ‘Happy Birthday’,
            name)
        else:
            print( ‘Please enter a
            birthday’)
name = ‘Mary Lee’
dob = datetime.datetime(1976, 6, 15 )
happybirthday(dob, name)
```

```
1
9
2
0
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - NoneType
Chapter 3 - Invoke a Function
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type
Chapter 6 - Object Value

7.31 Key Not in Dictionary

Description: When I try to retrieve a key value there is a key error.

Intended Outcome

Line 6 prints the value of key ‘4.’

Actual Result

The program raises a KeyError, pointing to line 5. This error message means there is no key ‘4’ in myDict.

```
KeyError: 4
```

Incorrect Code

This is the Example 31 code before any changes.

```
myDict = {1 : 'A' , 2 : 'B' , 3 : 'C'
}
if myDict[4 ] == 'D' :
        print ('4 is:' , myDict[4 ])
```

Debugging Steps

Open Variable Explorer and double click on “myDict.” The popup window shows there is no key ‘4.’

How to Resolve the Issue

Instead of directly referencing the key in myDict, I change the code on line 5 to use the .get method. The get() method returns 0 by default and does not raise an error.

Good Code

```
myDict = {1 : 'A' , 2 : 'B' , 3 : 'C' }
if myDict[.get(4 , 0 ) == 0 :
        print ('4 is not in the
        dictionary' )
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Dictionary
Chapter 4 - Debug Mode - Variable Explorer
Chapter 5 - Traceback

7.32 Error Combining Strings

Description: This code cocatenates two strings.

Intended Outcome

myvar3 should equal 23.

Actual Result

When the program runs, this error is raised pointing to line 5.

TypeError: can only concatenate str (not “int”) to str

Incorrect Code

This is the Example 32 code before any changes.

```
1 myvar1 = ‘2’
2 myvar2 = 3
3 myvar3 = myvar1 +
  myvar2
```

Debugging Steps

Variable Explorer shows the type for all variables. **myvar1** is a ‘str’, but myvar2 is an “int.”

How to Resolve the Issue

On line 5, I use the str() function to change the value of myvar2 to a “str.” You can only concatenate a “str” value with another “str” value.

Good Code

```
myvar1 = '2'
myvar2 = 3
myvar3 = myvar1 + str(myvar2)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 6 - Check Object Type

7.33 Function Returns 2 Values

Description: Example 33 prints the result from a function. The function returns a tuple of two elements, and the code should use the first element.

Intended Outcome

The function should calculate and print the birth year.

Actual Result

The program prints two tuple values.

(33, ‘John’) was born in 1987

Incorrect Code

This is the example 33 code before any changes.

```
1
2 from datetime import date
3 def myfunction():
4       return 33 , ‘John’
5 year = date.today().year
6 name = myfunction()[1 ]
7 birthyr = year - myfunction()[0 ]
8 print (name, ‘was born in’ ,
9 birthyr)
10
11
```

Debugging Steps

The Console printout of “name” is a tuple as indicated by the parentheses, which is also shown in Variable Explorer.

How to Resolve the Issue

On line 11 I add indexing to select the second element.

Good Code

```
1 from datetime import date
2 def myfunction():
3       return 33 , ‘John’
4 year = date.today().year
5 name = myfunction()
6
7
```

```
birthyr = year - myfunction()[ 1 ]
print (name, 'was born in' ,
birthyr)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Interactive Mode
Chapter 5 - Traceback

7.34 Unsupported Operand

Description: Trying to concatenate two strings raises an error This is a simple syntax error.

Intended Outcome

The variable “mystr” is assigned to “greenapples’

Actual Result

When the program runs an error is raised.

TypeError: bad operand type for unary +: ‘str’

Incorrect Code

This is the Example 34 code before any changes.

```
mystr = 'green' ++ 'apples
```

```
’
```

Debugging Steps

There is no ++ operand in Python.

How to Resolve the Issue

Update the code to use “+.”

Good Code

```
mystr = ‘green’ +
‘apples’
```

7.35 Code Goes Beyond Last Yield Statement

Description: Example 35 prints the results from a function. The code halts and raises a StopIteration error because the code goes beyond the last yield statement in a generator.

Intended Outcome

The code should print a series of numbers.

Actual Result

When the program runs, the Console shows a StopIteration error pointing to line 10.

Incorrect Code

This is the Example 35 code before any changes.

```
def genTest():
1       yield 1
2       yield 2
3       print ('inside genTest, i
4       is', i)
5
6 foo = genTest()
7 for i in range (3 ):
8       print ('item ',
9       foo.__next__())
```

Debugging Steps

The Traceback shows an error on line 8. Variable Explorer only shows the value of i. I add a print statement on line 4 within the function so I can see where the error is occurring. As I run the program in Debug Mode, when it prints line 4 “i” has a value of 2. This means there are three elements, and valid indexes are 0, 1, and 2.

How to Resolve the Issue

The range(2) on line 8 needs updated.

Good Code

```
1 def genTest():
2       yield 1
3       yield 2
4       print ('inside genTest, i
5       is', i)
6
7 foo = genTest()
8
```

```
for i in range (2 ):
    print (‘item ’ ,
    foo.__next__())
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Debug Mode - Variable Explorer
Chapter 4 - Interactive Mode
Chapter 5 - Traceback

7.36 Missing positional argument

Description: Instead of printing to the Console, the sing() function raises a TypeError.

Intended Outcome

The sing() function should print, “Happy Birthday dear John.”

Actual Result

When the program runs, the Console shows:

TypeError: sing() missing 1 required positional argument: ‘person’

Incorrect Code

This is the Example 36 code before any changes.

```
1 def sing(words,
2 person):
3
4       print (words,
5       person)
  sing('John' )
```

Debugging Steps

The Traceback shows the error is on line 5 with a missing positional argument 'person.' The function definition on line 1 shows two Positional Arguments , and the function call on line 5 has only one argument.

How to Resolve the Issue

My function call on line 5 needs updated with two arguments.

Good Code

```
1 def sing(words, person):
2
3       print (words, person)
4 sing('Happy Birthday dear' , 'John'
5 )
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Positional Arguments
Chapter 5 - Traceback

7.37 Reserved Keyword

Description: This example combines two strings and prints the result to the Console. The print statement raises an error.

Intended Outcome

The Console prints, “Hello John.”

Actual Result

When the program runs the Console shows:

TypeError : ‘str’ object is not callable

Incorrect Code

This is the Example 37 code before any changes.

```
1 str = ‘hello’
2 str2 = ‘John’
3 print (str (str + ‘ ’ +
  str2))
```

Debugging Steps

The Traceback shows and error on line 3.

How to Resolve the Issue

The keyword “str” is reserved and can’t be used as a variable name. I’ll update lines 1 and 2 with a new variable name.

Good Code

```
1 str1 = 'hello'
2
3 str2 = 'John'
  print (str (str1 + ' ' +
  str2))
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Reserved Keywords
Chapter 5 - Traceback

7.38 Dot Instead of Underscore

Description: Calling a class method raises an AttributeError.

Intended Outcome

After creating a new instance of the "Car" class, in line 6 I want to invoke the parallelpark() method.

Actual Result

When the program runs, an AttributeError is raised.

In [2]: runfile..Ex_38_Wrong.py", line 7, in <module>

myCar.parallel_park()

AttributeError: 'Car' object has no attribute 'parallel_park'

In [3]:

Incorrect Code

This is the Example 38 code before any changes.

```
1 # Ex_38_Wrong.py
2 from car import Car
3
4 myCar = Car('Subaru' , 'Crosstrek' , 2019
5 )
6 myCar.parallel_park()
```

Debugging Steps

An AttributeError is raised when you try to access an object's attribute that doesn't exist. The Traceback message pointed to line 6 where I attempted to use the "parallel_park()" method for the object "myCar." In the Console, I type "**help(myCar)** " to view a list of attributes and methods for the "myCar" object.

In [**2**]: help(myCar)

```
class Car(builtins.object
| Car(model, make, year
|
| Class represents a car
| Data properties:
| .model
| .make
| .year
| Behavirors/Operations:
| .drive
| .parallelpark
```

The "Car" class has a docstring that lists the data peroperties and behaviors for the class. After reviewing the docstring help, I realize I misspelld the "parrallel_park" method.

Note: I could also have typed one of these commands to see information about "myCar."

In [**3**]: dir(myCar)

In [**4**]: ? myCar

How to Resolve the Issue

Since I know the method name is wrong, I update line 6 with the correct name.

Good Code

```
1 # Ex_38_Wrong.py
2 from car import Car
3
4 myCar = Car(‘Subaru’ , ‘Crosstrek’ , 2019
5 )
6 myCar.parallel_park()
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 5 - Traceback

7.39 Key error in new dictionary key:pair

Description: A TypeError is raised when I try to create a new dictionary key:pair.

Intended Outcome

This example should create a new dictionary object with a list value.

Actual Result

When the program runs the Console shows:

```
In [2]: ..Ex_39_wrong.py", line 2, in <module>
mydict1 = {['2019', '2020']: ['great yr', 'bad yr']}
TypeError: unhashable type: 'list'
```

Incorrect Code

This is the Example 39 code before any changes.

```
1 #Ex 39 wrong.py
2 myDict = {}
3 myDict('a') =
4 3
```

Debugging Steps

The Traceback shows and error on line 2. Dictionary keys must be a hashable, immutable object. In this example, I tried to use a "list" as my dictionary key, and a list is mutable and not hashable.

How to Resolve the Issue

On line 4 I change the code to use square brackets around the key name.

Good Code

```
1 #Ex 39
2 correct.py
3 myDict = {}
4 myDict['a'] = 3
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - immutable
Chapter 5 - Traceback

7.40 Assign a dictionary key:value

Description: When I try to create a new key:value pair, the Python interpreter raises an error.

Intended Outcome

The code should create a new key:value pair.

Actual Result

When the program runs, and the Console shows:

In [2]: SyntaxError: can't assign to a functional

Out [1]: 'hello'

Incorrect Code

This is the Example 40 code before any changes.

```
myDict = {}
myDict('a') =
3
```

Debugging Steps

The Traceback says the SyntaxError is on Line 2.

How to Resolve the Issue

The key name should be enclosed in square brackets instead of paranthesis.

Good Code

```
myDict = {}
myDict['a' ] =
3
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback

7.41 Too many values to unpack

Description: Too many values to unpack when assigning values to three variables.

Intended Outcome

The assignment statement on line 1 should create three new variables

Actual Result

The program halts and the Traceback shows an error.

In [2]: ..Ex_41_Wrong.py", line 2, in <module>
var1, var2, var3 = 0, 1, 0 ,0

ValueError: too many values to unpack (expected 3)

Incorrect Code

This is the Example 41 code before any changes.

```
1 var1, var2, var3 = 0 , 1 , 0 ,
  0
```

Debugging Steps

This is an example of getting off to a rough start. The Traceback shows and error on line 1.

How to Resolve the Issue

While it is legal to assign values to multiple variables on the same line, you must have an equal number of variables and values on the left and right side of the expression. I will remove the last "0" on line 1.

Good Code

```
1 var1, var2, var3 = 0 , 1 ,
  0
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback

7.42 Tuple Assignment Error

Description: Trying to change tuple element [0] raises an error.

Intended Outcome

I want to change the first element in the tuple.

Actual Result

When the program runs it halts with a TypeError.

In [2]: ..Ex_42_wrong.py", line 3, in <module>
mytuple[0] = 1

TypeError: 'tuple' object does not support item assignment

Incorrect Code

This is the Example 42 code before any changes.

```
1 mytuple = (2 , 3 , 4
2 )
  mytuple[0 ] = 1
```

Debugging Steps

The Traceback shows and error on line 3. Tuples are immutable and can not be changed.

How to Resolve the Issue

Since I can't change one element ina tuple, I have to rethink the code. I can replace the entire tuple with a new tuple with three elements, or use a list object instead of a tuple. In this example, I switch to a list which is mutable and can be changed.

Good Code

```
myList = [2, 3, 4]
myList[0] = 1
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Debug Mode - Variable Explorer

7.43 'str' object is not callable

Description: When I try to contact two strings and print the result, a TypeError 'str' object is not callable is raised.

Intended Outcome

In this code, I try to concatenate two strings and print the result.

Actual Result

When the program runs the Console shows:

In [2]: ..Ex_43_Wrong.py", line 5, in <module>
print (str(str + mystr))

TypeError: 'str' object is not callable

Incorrect Code

This is the Example 43 code before any changes.

```
1 str = 'hello'
2 mystr2 = 'world'
3 print(str (str +
  mystr2)
```

Debugging Steps

The Traceback shows and error on line 3. In the Editor, I notice that both "str" is purple on line 3, indicating this is a reserved keyword or function. The problem started on line 3 when I used a reserved keyword as a variable name.

How to Resolve the Issue

On line 3 I need to change my variable name, and then I also need to update line 5 to reflect the new variable name.

Good Code

```
1 str = 'hello'
2 mystr2 = 'world'
3 print(str (mystr1 +
  mystr2)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Reserved Keywords
Chapter 3 - Keyword Arguments
Chapter 5 - Traceback

7.44 Can only Concatenate Tuple

Description: While trying to concatenate a tuple, the Python interpreter raises an TypeError.

Intended Outcome

The code should combine two tuples.

Actual Result

When the program runs the Console shows:

```
In [2 ]: ..Ex_44_Concatenate_Tuple_wrong.py", line 5, in <module>
myTuple3 = myTuple1 + myTuple2[0 ]
TypeError: can only concatenate tuple (not "int") to tuple
```

Incorrect Code

This is the Example 44 code before any changes.

```
1 #
2 Ex_44_Concatenate_Tuple_Wrong.py
3 myTuple1 = (1, 'a' , 2, 'b' )
4 myTuple2 = (3, 'c' )
```

```
5 myTuple3 = myTuple1 + myTuple2[0
  ]
```

Debugging Steps

The Traceback shows an error on line 5, and mentions an “int”. In the Console, I type “myTuple2[0]” and type(myTuple2[0]. The second object is an “int” with a value of 3.

```
In [2]: myTuple2[0]
Out [2]: 3

In [3]: type (myTuple2[0])
Out [3]: int
```

How to Resolve the Issue

Because I can’t concatenate a “tuple” and an “int,” I remove the index on line 5.

Good Code

```
1 #
2 Ex_44_Concatenate_Tuple_Correct.py
3 myTuple1 = (1, ‘a’ , 2, ‘b’ )
4 myTuple2 = (3, ‘c’ )
5 myTuple3 = myTuple1 + myTuple2
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 5 - Traceback

7.45 Float Comparison

Description: Comparing two float values fails.

Intended Outcome

The two float values are the same and should print the statement on line 6.

Actual Result

When the program runs the comparison on line 5 is False and the print statement is not executed.

Incorrect Code

You'll have to use your imagination that my computer stored these float values differently. Although the float values look the same to me, internally the computer can store them differently.

```
myfloat1 = .017
myfloat2 = .017
if myfloat1 == myfloat2:
    print ('they are the same')
```

Debugging Steps

When two float values don't match and you know they should, use the abs() function.

How to Resolve the Issue

Rewrite the code to use the abs() function in the comparison.

Good Code

```
myfloat1 = .017
myfloat2 = .017
if abs( myfloat1 - myfloat2) < .0001
:
    print (‘they are the same’
    )
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Comparing Floats

7.46 Unhashable type:dict

Description: A TypeError is raised when I try to create a new dictionary key:value pair with an “int” value.

Intended Outcome

This example should create a new dictionary object with an “int” value.

Actual Result

When the program runs the Console shows:

In [2]: ..Ex_46/Ex_46_Wrong.py”, line 5, in <module>

mydict1[mydict2] = 8

TypeError: unhashable type: ‘dict’’

Incorrect Code

This is the Example 46 code before any changes.

```
1 # Ex 46 Wrong.py
2 mydict1 = {}
3 mydict2 = {2 : ‘orange’
4 }
5 mydict1[mydict2 ] = 8
```

Debugging Steps

The Traceback shows and error on line 5 and states unhashable type “dict.” Dictionary keys must be a hashable , immutable object. In this example, I tried to use a “dictionary” which is mutable as my dictionary key.

How to Resolve the Issue

Rather than using the “mydict2” object as the key name, I need to use the individual key name. Key names are always an immutable object. I add a for loop on line 5 to iterate through the dictionary keys.

Good Code

```
1 # Ex 46 Correct.py
2 mydict1 = {}
3 mydict2 = {2 : ‘orange’
4 }
5 for key in mydict2:
6 mydict1[key] = 8
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - immutable
Chapter 5 - Traceback

7.47 builtin_function_or_method' object is not subscriptable

Description: When printing the error "'builtin_function_or_method' object is not subscriptable" is raised.

Intended Outcome

This expression should print "mystr."

Actual Result

When the program runs,

And the Console shows:

```
In [2]: Ex_47_Wrong.py", line 2, in <module>
  print[mystr]
TypeError: 'builtin_function_or_method' object is not subscriptable
Out [1]: 'hello'
```

Incorrect Code

This is the Example 47 code before any changes.

```
mystr =
1 'hello'
2 print [mystr]
```

Debugging Steps

The Traceback shows and error on line 2. After reviewing the code, I realize I used square brackets around “mystr” instead of parenthesis. Because print is a function I should use parenthesis around the argument “mystr.”

How to Resolve the Issue

On line 2 I change the square brackets to parentheses.

Good Code

```
mystr =
1 'hello'
2 print (mystr)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Print Statements
Chapter 5 - Traceback

7.48 String Comparison Error

Description: The expression to test if a word is in a string raises a TypeError.

Intended Outcome

On line 6 I am testing if a word is in the string “myphrase.” I want to print a phrase on line 7 if the expression is true.

Actual Result

When the program runs the Console shows:

```
In [3 ]: Ex_48_Wrong.py”, line 6, in <module>
  if words in myphrase:
TypeError: ‘ in <string>’ requires string as left operand, not list
```

Incorrect Code

This is the Example 48 code before any changes.

```
 # Ex 48 Wrong.py
1 words = [‘hello’ , ‘19 99’ ]
2
3 myphrase = ‘hello world it is 199
4 9’
5 mydict = {‘words’ : myphrase}
6
7 if words in myphrase:
      print (“ It’s 1999” )
```

Debugging Steps

The Traceback shows and error on line 6, and that my left operand must be a string instead of a list. In the Console I can use the “type()” function to find the type of “words”.

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

Out [1]: list

How to Resolve the Issue

On line 7 I added a for loop to iterate over the list “words.” I also changed the “if” statement on line 8 to use the iterator “word.” Now the test on line 8 is comparing each individual word in the “words” list.

Good Code

```
# Ex 48 Wrong.py
words = [‘hello’ , ‘1999’ ]

myphrase = ‘hello world it is 1999’

mydict = {‘words’ : myphrase}
for word in words:
    if word in myphrase:
        print (“It’s 1999” )
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type

7.49 Invalid literal for int() with base 10

Description: This code raises an error when converting a string into an integer.

Intended Outcome

This code converts a string into an integer for an arithmetic operation.

Actual Result

When the program runs,

And the Console shows:

```
In [2 ]: ..Ex_49/Ex_49_Wrong.py", line 4, in <module>
    total = 567 + int(mystr)
ValueError: invalid literal for int() with base 10: ' 1.25'
```

Incorrect Code

This is the Example 49 code before any changes.

```
1 # Ex 49 Wrong.py
2 mystr = '1.25'
3 total = 567 + int
4 (mystr)
```

Debugging Steps

The Traceback shows and error on line 4 that indicates something is wrong with my **int()** function. I know this expression is valid to convert "4" into the integer four, so something must be wrong with the "value" of the "mystr" variable. In fact, the Traceback showed ValueError.

The int() function is one of the "Built-in Functions " in the Python Standard Library. The online help for int(x) says "x" must be a string that represents an integer literal. While I can't convert the string '1.25' into a whole number int, it is legal to convert a float to an int.

How to Resolve the Issue

Since changing a "1.25" into an "int" is invalid, I first change the string into a float. As shown below, I updated line 4.

Good Code

```
1 # Ex 49 Correct.py
2 mystr = '1.25'
3 total = 567 + int (float
4 (mystr))
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback
Chapter 6 - Check Object Type
Chapter 6 - Object Value

7.50 Variable Referenced Before Assignment

Description: A variable is referenced before assignment.

Intended Outcome

My function “myfunc()” should increment my “transaction” variable.

Actual Result

When the program runs the Console shows an error on line 3.

In [3]: Ex_50_Wrong.py”, line 3, in myfunc
transaction += 1

UnboundLocalError: local variable ‘transaction’ referenced before assignment

Incorrect Code

This is the Example 50 code before any changes.

```
1 def myfunc():
2     transaction +=
3     1
4     print
5     (transaction)
6 myfunc()
```

Debugging Steps

The Traceback shows and error on line 3. The variable “transaction” is inside a function definition, which is why the error is an “UnboundLocalError.” If this expression was contained in the main program instead of a function definition, the error would be a “NameError” that the variable “transaction” was not defined.

How to Resolve the Issue

The error message says I used “transaction” on line 3 before it was assigned a value, so I insert an assignment statement on line 2.

Good Code

```
def myfunc():
    transaction = 0
    transaction += 1
    print(transaction)
myfunc()
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback

7.51 Plot: Shape Mismatch

Description: While create a bar chart a ValueError is raised.

Intended Outcome

This code draws a bar chart.

Actual Result

When the program runs, the Console shows:

In [2]: ValueError: shape mismatch: objects cannot be broadcast to a single shape

Incorrect Code

This is the Example 51 code before any changes.

```
import matplotlib.pyplot as plot
import statistics
myL = [1.41 , 2.3 , 1.99 , 1. 2 ]
total = str (len (myL))
bins = [0 , 1 , 2 , 3 , 4 ] # change to whole numbers
avg = str (round (statistics.mean(myL), 2 ))
xlbl = total + ‘ Purchases, Avg: $’ + avg
xl = [‘Item1’ , ‘item2’ , ‘item3’ , ‘item4’ ]
fig, ax = plot.subplots()
ax.bar(bins, myL, width=.3 , color=‘b’ )
plot.xticks(bins, xl, color=‘cornflowerblue’ )
plot.yticks(color=‘g’ )
plot.ylabel(‘cost’ , color=‘g’ )
plot.xlabel(xlbl, color=‘g’ , style=‘italic’ )
ax.set_title(‘Avg Cost Per Purchase’ , color=‘r’ ,
                fontweight=‘bold’, fontsize=12)
ax.spines[‘top’ ].set_color(‘y’ )
ax.spines[‘bottom’ ].set_color(‘y’ )
ax.spines[‘left’ ].set_color(‘y’ )
ax.spines[‘right’ ].set_color(‘y ’ )
fig.subplots_adjust(top=1.1 ) # add space at top
plot.text(3.6 , 1.8 , ‘2020’ , bbox={‘alpha’ : 0.1 })
plot.show()
```

Debugging Steps

The Traceback shows and error on line 12, which is the function call bar(). The error is not immediately obvious. I can search the internet for the exact error message, or I can look at the arguments to see if they are reasonable for this function. I experiment with only one element in both lists, and realize Matplotlib requires the x and y iterators have the same number of elements. Line 12 uses the “bins” and “myL” lists created on lines 5 and 7, which do not have the same number of elements.

How to Resolve the Issue

On line 7, I change my “bins” list.

Good Code

```
import matplotlib.pyplot as plot
import statistics
myL = [1.41 , 2.3 , 1.99 , 1. 2 ]
total = str (len (myL))
bins = [1 , 2 , 3 , 4 ] # change to whole numbers
avg = str (round (statistics.mean(myL), 2 ))
xlbl = total + ‘ Purchases, Avg: $’ + avg
xl = [‘Item1’ , ‘item2’ , ‘item3’ , ‘item4’ ]
fig, ax = plot.subplots()
ax.bar(bins, myL, width=.3 , color=‘b’ )
plot.xticks(bins, xl, color=‘cornflowerblue’ )
plot.yticks(color=‘g’ )
plot.ylabel(‘cost’ , color=‘g’ )
plot.xlabel(xlbl, color=‘g’ , style=‘italic’ )
ax.set_title(‘Avg Cost Per Purchase’ , color=‘r’ ,
                fontweight=‘bold’, fontsize=12)
ax.spines[‘top’ ].set_color(‘y’ )
ax.spines[‘bottom’ ].set_color(‘y’ )
ax.spines[‘left’ ].set_color(‘y’ )
ax.spines[‘right’ ].set_color(‘y’ )
```

```
fig.subplots_adjust(top=1.1 ) # add space at top
plot.text(3.6 , 1.8 , '2020' , bbox={'alpha' : 0.1 })
plot.show()plot.show()
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 5 - Traceback

7.52 Unpacking Operator

Description: In a function call, I leave off the ***** unpacking operator and an IndexError is raised.

Intended Outcome

This code prints tickets for each name from the “name” tuple.

Actual Result

When the program runs, the Console shows an error from line 4 .

```
In [2 ]: .. line 4, in print_tickets
    print(name[i])
IndexError: tuple index out of range
```

Incorrect Code

This is the Example 52 code before any changes.

```
1 def print_tickets(number_of_tickets,
2 *name):
3     i = 0
4         while i < number_of_tickets:
5         print (name[i])
6         i += 1
7
8 names = ('John' , 'Alice' )
9 print_tickets(2 , names)
```

Debugging Steps

The Traceback shows and index error on line 4, indicating the tuple indexing I'm using is wrong. Because the program has ended, I need to enter "debug mode" to see the value of the "name" variable.

1. In the Console, type %debug to start "debug mode."

```
In [ 2 ]: %debug
    2 i = 0
    3 while i < number_of_tickets:
----> 4 print(name[i])
    5 i += 1
    6
ipdb>
```

2. The Console displays an arrow next to line 4, and the prompt changes to ipdb>. In the Console, type name.

ipdb> name

((‘John’, ‘Alice’),)

The Python Interpreter returns a tuple in a tuple. I was expecting the tuple show below.

(‘John’, ‘Alice’)

I realize I left the * unpacking operator off my function call on line 9.

How to Resolve the Issue

I realize I left the * unpacking operator off my function call on line 9.

Good Code

On line 9 I add the unpacking operator *.

```
1 def print_tickets(number_of_tickets,
2 *name):
3     i = 0
4         while i < number_of_tickets:
5         print (name[i])
6         i += 1
7
8 names = (‘John’ , ‘Alice’ )
9 print_tickets(2 , *names)
```

Reference

These topics from previous chapters are a good reference for this example.

Chapter 3 - Unpacking Operators

Chapter 4 - Debug Mode
Chapter 4 - Interactive Mode
Chapter 5 - Traceback
Chapter 6 - Check Object Type
Chapter 6 - Object Value

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.

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.

Built-in Functions in the Python Standard Library.

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

Appendix - Errors

Example	Traceback Error
1	N/A
2	list index out of range
3	list index out of range
4	N/A
5	expected an indented block
6	‘int’ object has no attribute ‘upper
7	expected an indented block
8	invalid syntax
9	invalid syntax
10	name <object> is not defined
11	invalid syntax
12	invalid syntax
13	name <object> is not defined

14	No such file or directory: ‘file.txt’
15	Custom error message is raised
16	invalid syntax
17	unsupported operand type(s) for /: ‘NoneType’ and ‘int’
18	module ‘math’ has no attribute ‘cube’
19	name ‘plt’ is not defined
20	‘Worksheet Sheet1 does not exist.’
21	invalid literal for int() with base 10:
22	name ‘plt’ is not defined
23	list.remove(x): x not in list
24	float divison by zero
25	N/A

Example	Traceback Error
26	month must be in 1..12
27	N/A’
28	NoneType object has no attribute ‘upper’
29	N/A
30	N/A
31	KeyError: 4
32	can only concatenate str (not “int”) to str
33	N/A
34	bad operand type for unary +: ‘str’
35	StopIteration
36	missing 1 required positional argument:
37	str object is not callable
38	object has no attribute
39	unhashable type: ‘list’
40	can’t assign to a functional
41	too many values to unpack
42	‘tuple’ object does not support item assignment
43	‘str’ object is not callable
44	can only concatenate tuple (not “int”) to tuple
45	N/A
46	unhashable type: ‘dict’
47	‘builtin_function_or_method’ object is not subscriptable
48	‘in <string>’ requires string as left operand, not list
49	invalid literal for int() with base 10:
50	local variable <variable> referenced before assignment
51	shape mismatch: objects cannot be broadcast to a single shape
52	tuple index out of range

bad operand type for unary +: ‘str’ -- Example 34

builtin_function_or_method’ object is not subscriptable -- Example 47

can only concatenate str (not “int”) to str -- Example 32

can only concatenate tuple (not “int”) to tuple -- Example 44

can’t assign to a functional -- Example 40

Custom error message is raised -- Example 15

expected an indented block -- Example 5

expected an indented block -- Example 7

float divison by zero -- Example 24

in <string>’ requires string as left operand, not list -- Example 48

int object has no attribute ‘upper -- Example 6

invalid literal for int() with base 10: -- Example 21

invalid literal for int() with base 10: -- Example 49

invalid syntax -- Example 8

invalid syntax -- Example 9

invalid syntax -- Example 11

invalid syntax -- Example 12

invalid syntax -- Example 16

KeyError: 4 -- Example 31

list index out of range -- Example 2

list index out of range -- Example 3

list.remove(x): x not in list -- Example 23

local variable <variable> referenced before assignment -- Example 50

missing 1 required positional argument: -- Example 36

module ‘math’ has no attribute ‘cube’ -- Example 18

month must be in 1..12 -- Example 26

name ‘plt’ is not defined -- Example 19

name ‘plt’ is not defined -- Example 22

name <object> is not defined -- Example 10

name <object> is not defined -- Example 13

No such file or directory: ‘file.txt’ -- Example 14

NoneType object has no attribute ‘upper’ -- Example 28

object has no attribute -- Example 38

shape mismatch: objects cannot be broadcast to a single shape -- Ex 51

StopIteration -- Example 35
str object is not callable -- Example 37, 43
too many values to unpack -- Example 41
tuple index out of range -- Example 52
tuple object does not support item assignment -- Example 42
unhashable type: ‘dict’ -- Example 46
unhashable type: ‘list’ -- Example 39
unsupported operand type(s) for /: ‘NoneType’ and ‘int’ -- Example 17
Worksheet Sheet1 does not exist.’ -- Example 20

Appendix - Libraries

This is a list of some of my favorite Python libraries.

beautifulsoup
bs4
calendar
Collections
copy
csv
datetime
inspect
logging
math
matplotlib
numpy
openpyxl
operator
os
pillow
re
smtplib
string
time
timeit
urllib

Index

_ -- Section: 3.4, 3.21, 4.8
‘’ -- Section: 3.9
“” -- Section: 3.9
() -- Section: 3.13
[ ] -- Section: 3.13
[] -- Section: 3.18, 3.19
{} -- Section: 3.13, 3.16, 3.28
* -- Section: 3.12, 3.24
** -- Section: 3.24
& -- Section: 3.16
^ -- Section: 3.17
= -- Section: 3.4
| -- Section: 3.16, 3.20
// Integer division -- Section: 3.20
% modulo -- Section: 3.20
abstraction -- Section: 3.25, 3.26
Abstraction -- Section: 3.24
add -- Section: 3.21
Add a New Key Pair to an Existing Dictionary -- Section: 3.13
Add Character When Joining Strings -- Section: 3.11
algorithmic complexity -- Section: 2.1
alias -- Section: 3.10, 3.20
alphabet -- Section: 3.9
Anaconda -- Section: 2.3
append -- Section: 3.18

Append element to list -- Section: 3.11
Append to a Dictionary -- Section: 3.13
arbitrary argument -- Section: 3.24
arbitrary keyword argument -- Section: 3.24
arguments -- Section: 3.24
Arithmetic -- Section: 3.20
ArithmeticError -- Section: 5.6
arrays -- Section: 3.12
assert -- Section: 5.0, 5.5
Assign -- Section: 4.3
assignment operator -- Section: 3.17
assignment statement -- Section: 3.4, 3.9, 3.13
asymptotic notation -- Section: 2.1
attribute -- Section: 3.25, 3.25, 3.28, 4.3, 4.10, 5.6
AttributeError -- Section: 3.13, 3.25, 4.9, 5.6
bar chart -- Section: 7.51
BeautifulSoup -- Section: 7.24
behavior -- Section: 3.10, 3.12, 3.24, 3.24
Big O -- Section: 2.1
binary -- Section: 3.7
bind -- Section: 3.4
block of code -- Section: 3.1, 3.22, 3.23
boolean -- Section: 3.6, 3.24
brackets -- Section: 3.18, 3.19
branch -- Section: 3.22
break -- Section: 2.4, 3.22, 4.8
Breakpoint -- Section: 4.2, 4.4, 4.6, 4.8

bs4 -- Section: 7.24
built-in -- Section: 3.9, 5.1, 5.6
Calling a Function -- Section: 3.24, 7.52
Calls -- Section: 3.24, 7.52
Call Signature -- Section: 3.24, 3.24, 3.25, 4.10
case sensitive -- Section: 3.1, 7.10
chart -- Section: 3.12, 7.22
Classes -- Section: 3.1, 3.25, 3.28
Clear Console -- Section: 4.8, 4.8
Clone -- Section: 3.10
Cntrl C -- Section: 2.4
Code Block -- Section: 2.4, 3.23
code completion -- Section: 4.3
Collections -- Section: 3.12, 3.17
Comparison Operators -- Section: 3.8, 3.15, 3.20
Complexity -- Section: 2.1
comprehension -- Section: 3.13
concatenate -- Section: 3.9, 3.12, 3.20
concatenation -- Section: 3.9, 3.12, 3.20, 3.20
Console -- Section: 2.3, 2.4, 3.9, 4.2, 4.4, 4.5
Constant complexity -- Section: 2.1
Container -- Section: 3.6, 3.24
continues -- Section: 3.10
Control Statements -- Section: 3.22, 3.24
Convert Data Type -- Section: 3.6, 3.9
copy -- Section: 3.10
count -- Section: 3.11, 3.20

Counter -- Section: 3.17, 3.23, 4.8, 4.13
csv -- Section: 7.14
cube -- Section: 7.18
Data scrubbing -- Section: 2.1
Data Structures -- Section: 3.1, 3.8, 3.22, 4.10, 4.10
Data type -- Section: 3.6, 3.10
datetime -- Section: 5.6, 6.7
day -- Section: 4.3, 4.14
Debug Commands -- Section: 4.8
Debug Mode -- Section: 4.4
decimal -- Section: 3.9
decomposition -- Section: 3.24
deepcopy -- Section: 3.10
defaultdict -- Section: 3.18
Define a Function -- Section: 3.24, 3.24
Defined Behavior -- Section: 7.52
del() - delete -- Section: 3.10, 3.13
dictionary -- Section: 3.18
Dictionary -- Section: 3.13
Dictionary Comprehensions -- Section: 3.13
dictionary unpacking operator -- Section: 3.24
dir() -- Section: 4.10
Dir() -- Section: 3.9, 4.10
Divide and Conquer -- Section: 2.2, 2.2
docstring -- Section: 4.10, 4.10
Dotted Notation -- Section: 3.1, 3.25, 3.25, 3.25, 3.26
down -- Section: 4.8

dummy -- Section: 3.4
duplicate -- Section: 3.13
Editor -- Section: 2.3, 2.3, 3.1, 4.1, 4.2, 4.3
efficient -- Section: 2.1
Elements -- Section: 2.2, 3.8, 3.12, 3.13, 3.13, 3.18, 3.19, 3.24
Else Statement -- Section: 3.22, 3.24
Endless loop -- Section: 7.1, 7.4, 7.6
equal -- Section: 3.21
Even -- Section: 3.20
Excel -- Section: 7.6, 7.20
except -- Section: 7.15
Exit -- Section: 4.8
exponent -- Section: 3.20
extend -- Section: 3.11
extend() -- Section: 3.11
file -- Section: 7.14
filter() -- Section: 3.24
find -- Section: 3.8, 3.9, 3.20
float -- Section: 3.6, 3.6, 3.7, 3.20, 4.10
Float rounding errors -- Section: 4.10
Focused testing -- Section: 4.14
for -- Section: 3.12, 3.13, 3.15, 3.22, 3.22
fsum -- Section: 7.18
function -- Section: 3.24
Function Return Object -- Section: 3.12, 3.20, 3.24, 3.24
generator -- Section: 3.15
getfullargspec -- Section: 6.10

global -- Section: 3.4
Global -- Section: 3.1, 3.4, 3.24, 3.28, 4.10
globals() -- Section: 4.10
Graph -- Section: 7.19
hash -- Section: 3.21
hashable -- Section: 3.4, 3.13
help() -- Section: 3.9, 4.10
highlight -- Section: 4.1
histogram -- Section: 7.22
HTML -- Section: 4.15, 7.24
id() -- Section: 4.10
IDE -- Section: 1.2
Identifier -- Section: 3.1, 3.3, 3.9, 3.20, 3.25
Identity -- Section: 4.10
ignore -- Section: 3.4
immutable -- Section: 3.3, 3.6, 3.9, 3.12
import -- Section: 3.26, 5.6
in -- Section: 3.20
indentation -- Section: 3.23
index -- Section: 3.9, 3.11, 3.18
IndexError -- Section: 3.12, 3.18
indirection -- Section: 3.4, 3.10
inefficient -- Section: 2.1
init -- Section: 3.21
initialize -- Section: 3.28
input -- Section: 7.21
input() -- Section: 5.6, 7.21

inputs -- Section: 3.24
insert -- Section: 3.11
Inspect -- Section: 3.24, 4.10, 4.10
instance -- Section: 3.25
Instance -- Section: 3.25
instatiation -- Section: 3.25
integer -- Section: 3.7, 3.10, 3.12, 3.15
integer division -- Section: 3.20
integers -- Section: 3.10
Interactive mode -- Section: 4.8
Interactive Mode -- Section: 4.8
Introspection -- Section: 4.1, 4.3, 4.10
invoke -- Section: 3.21
Invoke a Function -- Section: 3.12, 3.12, 3.24
ipdb -- Section: 4.8
iPython -- Section: 4.5, 4.9
isalpha -- Section: 3.9
is, comparison operator -- Section: 3.20
isinstance -- Section: 3.6
itemgetter -- Section: 3.12
items -- Section: 3.8
iterables -- Section: 3.8, 3.12, 3.24
iterable unpacking operator -- Section: 3.24
iterate -- Section: 3.22
Iterate -- Section: 3.22
Iteration Variable -- Section: 3.22
iterator -- Section: 3.23

join -- Section: 3.9, 3.11
join() -- Section: 3.11
kernel -- Section: 7.1
keyboard interrupt -- Section: 2.4
keyword -- Section: 3.1, 3.24, 3.24, 7.6, 7.11
Keyword -- Section: 3.24, 7.16
lambda -- Section: 3.24, 3.24
last evaluation -- Section: 4.8
LEGB -- Section: 3.28, 3.28
len -- Section: 3.21
len() -- Section: 3.10, 4.10
less than -- Section: 3.25
lexical scoping -- Section: 3.28
libraries -- Section: 3.26
Linear complexity -- Section: 2.1
line return -- Section: 6.7
List comprehension -- Section: 3.10
List Index Out of Range -- Section: 7.1
Lists -- Section: 6.10
local -- Section: 3.4, 3.28, 4.8, 4.10, 4.10
Local -- Section: 4.10
locals() -- Section: 4.10
Logarithmic complexity -- Section: 2.1
logging -- Section: 4.1, 4.11, 4.12, 7.25
loop -- Section: 3.13, 3.20, 3.22
lower -- Section: 3.9
lstrip -- Section: 3.9

lstrip() -- Section: 6.7

lt -- Section: 3.21

magic function -- Section: 4.4, 4.6

main -- Section: 3.21

map -- Section: 3.11, 3.24

map() -- Section: 3.10

math -- Section: 4.8, 7.18

matplotlib -- Section: 3.12, 7.19, 7.22, 7.23

Matplotlib -- Section: 7.51

max -- Section: 3.20, 3.24

membership -- Section: 3.20

Memory -- Section: 4.9

method -- Section: 3.3, 3.4, 3.9, 3.24, 3.24, 3.25, 3.25, 7.6, 7.18, 7.23

min -- Section: 3.20

modularity -- Section: 3.26

modules -- Section: 3.26

modulo -- Section: 3.20

Modulo -- Section: 3.20

multidimensional -- Section: 3.12

mutable -- Section: 3.6, 3.10

name -- Section: 3.21

namedtuple -- Section: 3.12, 3.17

namedtuples -- Section: 3.17

namespace -- Section: 3.28

Namespace -- Section: 4.9, 4.10

NAN -- Section: 3.7

nested -- Section: 3.23, 4.2
new -- Section: 3.21
next -- Section: 4.8
None -- Section: 3.6, 6.6, 7.17
NoneType -- Section: 3.6, 3.6, 7.27, 7.28
nonlocal. -- Section: 3.28
non-scalar -- Section: 3.6
Non-scalar -- Section: 3.8
not a number -- Section: 3.7
numpy.py -- Section: 3.12
Objects -- Section: 3.3
Odd -- Section: 3.20
oldest date -- Section: 4.3
one-line swap -- Section: 3.12
open() -- Section: 7.14
operator -- Section: 3.12
operator overloading -- Section: 3.21
optional argument -- Section: 3.24
os -- Section: 7.14
Outline Pane -- Section: 3.24, 4.2
outputs -- Section: 3.24
override -- Section: 3.25
parameters -- Section: 3.24
pass -- Section: 3.24
plot -- Section: 3.12, 7.22
Plot -- Section: 7.19, 7.51
pop() -- Section: 3.11

Power -- Section: 3.20
quit -- Section: 4.8
range -- Section: 3.10, 3.15, 3.22
read() -- Section: 7.14
RecursionError -- Section: 5.6
recursive -- Section: 3.4, 3.24, 3.24
Recursive -- Section: 3.20
refactor -- Section: 3.12
reference -- Section: 3.10
Remainder -- Section: 3.20, 3.20
remove() -- Section: 3.11
repetition -- Section: 3.12, 3.20
replace -- Section: 3.9
repr() -- Section: 3.8, 3.9, 3.9, 4.10
reserved keywords -- Section: 3.1
restart -- Section: 7.1
return -- Section: 3.24, 6.10
Return object -- Section: 3.12, 3.20, 3.24
rindex -- Section: 3.9
rounding errors -- Section: 4.10
rsplit -- Section: 3.9
rstrip -- Section: 3.9
rstrip() -- Section: 6.7
sales tax -- Section: 3.10
scalar -- Section: 3.6, 3.7, 3.9
Scope -- Section: 3.4, 3.28
Search -- Section: 3.20

self -- Section: 3.25
sequence type -- Section: 3.8
set -- Section: 3.11, 3.16
shallow copy -- Section: 3.10
shell -- Section: 2.3
Shell -- Section: 2.3, 4.1, 4.2
signature() -- Section: 4.10
Slicing -- Section: 3.19
sort() -- Section: 3.11
sorted -- Section: 3.11
Special characters -- Section: 4.10
special method -- Section: 3.4, 3.21
special methods -- Section: 3.25
split -- Section: 3.9
sqrt -- Section: 4.8
stack -- Section: 3.4, 3.28, 4.5
stacks -- Section: 3.24
stop -- Section: 7.1
str -- Section: 3.21
strftime -- Section: 6.7
string indices must be integers -- Section: 3.10
strip -- Section: 3.9
strip() -- Section: 6.7
sub -- Section: 3.21
subclass -- Section: 3.25
suite -- Section: 3.23
superclass -- Section: 3.25

swapcase -- Section: 3.9
swap values -- Section: 3.12, 3.12
switch -- Section: 4.14
syntax -- Section: 3.1
tab -- Section: 6.7
text sequence type -- Section: 3.8
throw away -- Section: 3.4
time -- Section: 4.13, 6.7
timeit -- Section: 4.12
Tuple -- Section: 3.12
tuple object does not support item assignment -- Section: 3.12
txt -- Section: 4.9, 7.14
type() -- Section: 4.10
Type -- Section: 3.6, 3.13
TypeError -- Section: 3.10
underline -- Section: 3.21
underscore -- Section: 3.7, 3.21
unpack -- Section: 3.4
unpacking operator -- Section: 3.24
upper -- Section: 3.9
urllib -- Section: 7.24
urlopen -- Section: 7.24
user reports -- Section: 4.11
version -- Section: 3.21, 4.10
vowels -- Section: 3.10
while -- Section: 3.12, 3.22
whitespace -- Section: 3.9, 3.9, 6.7

Whitespace -- Section: 4.10
XML -- Section: 7.24, 7.27
year -- Section: 4.3
yield -- Section: 3.16, 3.24
zero -- Section: 6.7
zfill -- Section: 3.9
zip() -- Section: 3.12, 3.24