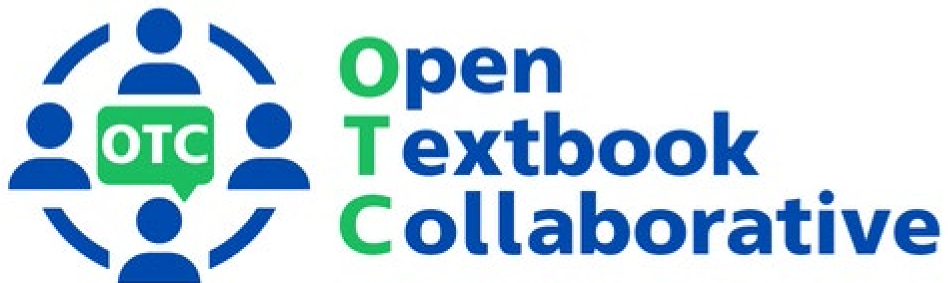
Introduction to Computer Programming with Python

Chris Simber

Computer Programming with Python: Starting Out with IDLE

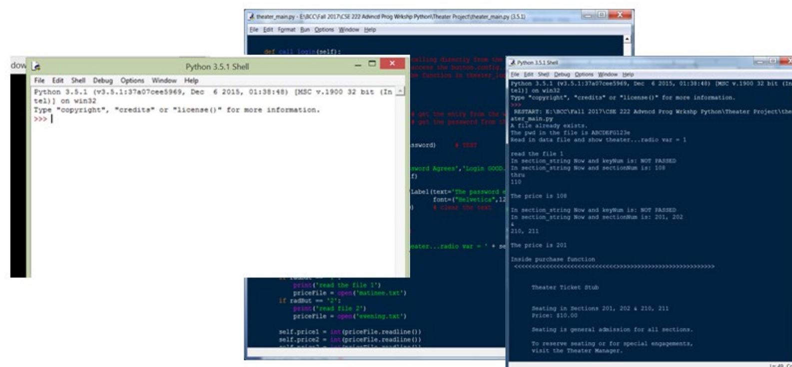
Contributing Authors

Chris Simber, Rowan College at Burlington County



Computer Programming: Python

Starting Out with IDLE



Chris Simber

Assistant Professor, Computer Science

Rowan College at Burlington County

Cataloging Data

Names: Simber, Chris, author.

Title: Introduction to Python Programming

Starting Out with IDLE

Subjects: Python (Computer Programming Language)

(2021)

Chris Simber

Assistant Professor of Computer Science

Rowan College at Burlington County

Author contact: csimber@RCBC.edu

Attribution-NoDerivs

CC BY-ND

This work is licensed under CC BY-ND 4.0. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/

Preface

This book is designed for use in an introductory course in programming using the Python programming language, and is intended for students who are not familiar with computer programming. The book follows the flow of an introductory text in computer programming and introduces general computer information and basic computer operations, as well as software engineering principles and processes used in industry.

The IDLE integrated development environment (IDE) is used throughout the text which is installed with Python, has a simplified interface, and provides for a short learning curve. The goal is to provide students with an overview of computers, software engineering tools and techniques, and to introduce programming in Python quickly.

The examples and exercises in the text follow the PEP 8 Style Guide for Python Code and reinforce the material being introduced while building on previous material covered. The chapter exercises are numbered for clarity using a shaded box, and can be used for assignment purposes.

There are end-of-chapter assignments, and an answer key, exams, and accompanying lecture slides are available.

Instructions for obtaining and installing Python with IDLE are provided in Appendix B. Instructions for using the PIP installer which is included in Python are provided in Appendix C for utilizing matplotlib. Links to the Python web site, Python Tutorials, and the PEP 8 Style Guide are included in Appendix D.

Python version 3.9.5 was in use at the time of this writing. The modules utilized include Tkinter, random, PhotoImage, and matplotlib.

Contributors:

Steven Chudnick, Project Coordinator

Alison Cole, Librarian, Felician University

Joshua Gaul, Educational Technology Manager, Edge

Robert Hilliker, Curriculum Council Manager

Janet Marler, Innovation and Technology Curriculum Committee Chair

Laura Wingler, Instructional Designer, Ocean County College

Contents

Chapter 1	Introduction	1
Chapter 2	The Python Shell and IDLE	19
Chapter 3	Getting Started in Python	29
Chapter 4	Decision Structures and Boolean Logic	65
Chapter 5	Logic, Loops, and Functions	89
Chapter 6	Functions	111
Chapter 7	File Operations and Dialogs	139
Chapter 8	Strings, Lists, Dictionaries, and Sets	161
Chapter 9	Classes and Objects	193
Chapter 10	Graphical User Interfaces (GUIs)	219
Chapter 11	Menus, Images, and Windows	251
Appendix A	ASCII Representations	269
Appendix B	Installing Python with IDLE	270
Appendix C	the PIP Installer	272
Appendix D	Resource Links	274

Index

"Five minutes of design time, will save hours of programming" –
Chris Simber

Chapter 1

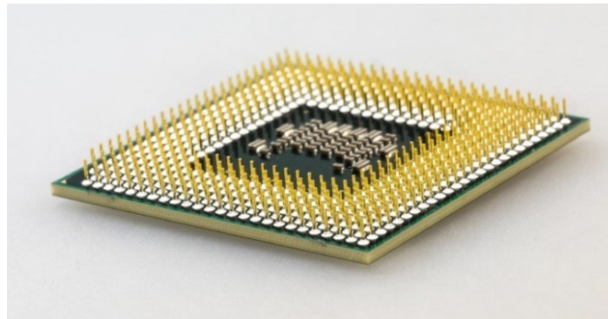
Introduction

Computers are simply data processing devices. They are machines that receive input, process it in some way, and produce output. Computer programs are statements that tell a computer what to do and in what order to do it. These programs provide a sequence of small steps for the computer to execute and produce the desired result. This may seem odd to the casual computer user, but millions or even billions of these small steps are being executed by the computer as we run programs. The small steps combine to provide what appear to be seamless operations as we interact with the computer. Programs are referred to as software and they are needed to make the computer useful…tell it what to do and in what order to do it. People who design, write, and test software are commonly referred to as software engineers, software developers, or computer programmers. To better understand the computing process and programming, a familiarity with the parts that make up a computer is necessary.

The Central Processing Unit (CPU)

Any part of the computer that we can physically touch (including inside the casing) is referred to as ***hardware***. The "Brains" or heart of the computer is the ***CPU*** (Central Processing Unit). The CPU performs the basic instructions and controls computer operations. There are two parts to the CPU. The Arithmetic Logic Unit (ALU) handles basic arithmetic and the comparison of data such as

less than, greater than, or equivalence. The Control Unit retrieves and decodes program instructions and coordinates activities within the computer.



Central Processing Unit (CPU)

The CPU plugs into a socket located on a circuit board called the Motherboard which houses other components including ***main memory*** or RAM.

Main Memory

RAM stands for Random Access Memory and is a series of memory chips on a circuit board installed in the computer, and is often referred to as main memory. The memory in these chips is a series of memory addresses that enable the computer to locate information. These memory addresses are ***volatile*** meaning that when the computer is turned off, RAM no longer holds data in these addresses. It is erased. When RAM is energized (the computer is running), there is always a value in the memory addresses. This data is commonly referred to as "garbage data" since it is random and was not loaded into memory by a program. When a program is launched, the program is copied into RAM overwriting the garbage data.

Laptop Main Memory (RAM)

The CPU can access instructions and data from RAM very quickly, and RAM is usually located close to the CPU on the motherboard. The RAM circuit card for a laptop computer is shown above. The large rectangles are the memory chips, the notches are an aid for inserting the RAM into position on the motherboard, and the gold edge makes the connection for data access. Since RAM is erased when the computer is turned off, ***secondary storage*** is used to retain information permanently.

Secondary Storage

Secondary storage devices are ***non-volatile*** and the information stored in them is not erased when they are not energized. Secondary storage devices include the hard drive inside the computer, external drives connected to the computer, and flash drives. The hard drive inside the computer may be a disk drive which houses a rotating disk and data access arm, or a solid-state drive which has no moving parts and operates faster than a traditional disk drive.

Hard Drive (cover removed)

External drives are typically solid-state and connect to the computer through a cable plugged into a USB (Universal Serial Bus) connector, or plug directly into the USB port of the computer as in the case of flash drives. These drives use flash memory, and do not have a disk drive.

Input and Output Devices

Input devices are anything that provides input or data for a computer. Common input devices are the keyboard, mouse, microphone, and camera.

Output devices include anything that accepts computer output such as monitors, speakers, and printers. Since data files located on storage devices can be used for reading data into a computer or writing output from a computer, they could be considered both input and output devices.

Software

Computers are machines that follow an input, processing, output sequence of operations and need to be told what to do and in what order to do it. This is accomplished through sets of instructions called ***software***. There are essentially two types of software: ***system software*** (operating systems), and ***application programs*** (all other software).

The operating system (OS) provides an interface for us to use computers more easily. A computer does not need an OS, but it is much easier for us to interact with the computer through an operating system. Originally a command line interface was used to operate a computer. Menu driven programs followed soon after, and the Graphical User Interface (GUI) replaced them and has been used since. The operating system provides the Graphical User Interface that is now commonly used to interact with the computer, and acts like an orchestra conductor by controlling computer hardware, managing devices connected to the computer, and interacting with programs that are running. Operating systems commonly in use today are Windows, macOS, and Linux.

Application software programs are the software that we commonly use to accomplish work on a computer such as word processors, spreadsheet applications, gaming software, and presentation programs.

The Language of Computers

The language of computers is a ***binary*** language consisting of ones and zeros. This is the beauty and simplicity of the computer. Everything is a 1 or a 0, is either On or Off, Yes or No, True or False. The ***bit***, or binary digit used in

computing represents a logical state that can be 0 or 1. It is the smallest information representation. A ***Byte*** is a combination of 8 bits of 0s and 1s, and in computers, each letter, number, and special character consists of a binary representation. For example, a lower case "f" is represented in binary as 01100110 in accordance with the ***ASCII*** standard for information exchange. ASCII (pronounced askee) stands for the American Standard Code for Information Interchange which was developed as a character encoding standard.

The ASCII table consists of binary representations for the 26 uppercase and 26 lowercase letters, and the 9 digits, as well as special characters, punctuation marks, and other symbols and keyboard keys. Appendix A provides a partial list of the binary and decimal representations for the upper and lowercase letters, digits, and punctuation. A portion is shown here.

Decimal	Binary	ASCII
64	0100 0000	@
65	0100 0001	A
66	0100 0010	B
67	0100 0011	C
68	0100 0100	D

ASCII Table (excerpt)

The Unicode standard incorporates the 256 item ASCII standard and expands to include the binary representations for symbols and the text used in most of the world's languages. Unicode 13.0 contains representations for 143,859 characters. Whatever we want to tell the computer to do must be in its' binary language. This includes numbers that are used in computations. Numeric ***integers*** (whole numbers) are represented in computers using the positions of the bits and powers of 2 starting from right and working toward the left.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Binary Number Bit Representations

As an example, the number 90 would be represented in binary as 01011010. Each bit is either 0 or 1 and is multiplied by the power of 2 at its position, and the results are added together.

0 1 0 1 1 0 1 0

$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$(0 \times 128) + (1 \times 64) + (0 \times 32) + (1 \times 16) + (1 \times 8) + (0 \times 4) + (0 \times 2) + (0 \times 1)$

$0 + 64 + 0 + 16 + 8 + 0 + 2 + 0$

$= 90$

Binary Number Conversion

The limit to the numbers that can be stored using 8 bits is 255 with all 8 bits being 1's. To store larger numbers, two Bytes would be used with a combination of sixteen 1's and 0's. In case you are curious, that would allow storing numbers as large as 65535. To store larger numbers, more Bytes could be used. To store negative numbers and floating point numbers (numbers with a decimal or fractional part), other numbering schemes are used such as the two's complement and floating-point notation.

Since any data stored in the computer must be in binary, images (which are made up of pixels) are stored by converting each pixel to a numeric value which is then stored in binary. Sound is stored using samples that are converted to the nearest numeric value.

Programming Languages

Although the computers' language, referred to as machine language, is a binary language, it would be tedious for us to write instructions for computers in Binary. Machine language is one of the two languages commonly referred to as ***low-level languages*** for computers, the other being Assembly language. Assembly language consists of very basic instructions like move a value into memory, add another value to that one, and store the result in memory. Both of these low-level languages mirror the operations of the CPU in simplicity and basic operations.

CPU operations include a process that is repeated many times very quickly. It completes a ***machine cycle*** in which it performs the same series of simple steps: fetch, decode, execute, and store. These machine cycles (or clock cycles) are occurring millions or even billions of times per second.

1. fetches the required piece of data or instruction from memory
2. decodes the instruction
3. executes the instruction
4. stores the result of the instruction

CPU Machine Cycle Steps

A CPU's processing power is measured in hertz (cycles-per-second), and a one-gigahertz (1 GHz) CPU can execute one billion cycles per second.

Writing software (programming) in low-level languages is possible, but ***high-level languages*** provide a much easier way. As more and more software was being written, high-level languages were introduced to make programming easier and more efficient. In high-level languages, multiple instructions are combined into a single statement. Some of these have been used extensively, others not so much. Today there are hundreds of high-level languages with approximately 250 in use. Each has benefits and limitations as well as a following, proponents, and detractors. The following is a short list of some popular high-level languages and their intended uses.

Ada	Department of Defense programs
BASIC	Beginners All-purpose Symbolic Instruction Code
C, C++	powerful general-purpose programming
COBOL	Common Business-Oriented Language - business programs
FORTRAN	FORmula TRANslator for math and science
Pascal	teaching programming
Java	applications running over the internet
Python	general-purpose applications and data handling

Popular High-level Programming Languages

Writing software in any of these languages is much easier than the low-level languages of Machine and Assembly, but the computer is still only interested in machine language. To translate programs written in a high-level language to the machine language for the computer, compilers and interpreters are used.

A ***compiler*** translates the high-level language into a separate machine language program. Software engineers refer to this as compiling or "building" the program. A "Build" is a compiled version of the software and the program can then be run whenever needed because it is a stand-alone executable program.

An ***interpreter*** on the other hand, reads, translates, and executes the program one line at a time. As an instruction in the program is read by the interpreter, it converts the instruction into machine language and then executes that instruction. The Python programming language uses an interpreter. Since interpreters translate and execute instructions, they do not typically create stand-alone machine language programs. However, there are applications such as PyInstaller that package Python programs into stand-alone executables.

The instructions written by programmers are referred to as ***source code***. The code is written using a text editor in an ***Integrated Development Environment (IDE)***. IDE's are software applications that include integrated tools for software developers to write, execute, and test software.

Syntax and Grammar

Each programming language has some characteristics and rules that must be followed when writing programs in that language. Two of these are the syntax and grammar of the language.

The ***syntax*** of a programming language refers to the rules for properly combining symbols, operators, and punctuation, as well as the proper use of operators.

The ***grammar*** of a programming language determines the structure of the sentences containing the symbols, operators, and punctuation that make up the instructions for the computer.

In addition, a characteristic of languages is the use of ***keywords***. Keywords are reserved by the language for a specific use and cannot be used for another

purpose. Most IDEs will display them using a color font to highlight them. The following is a list of some of the Python keywords.

```
False      except     else       import     in
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Python Key Words (partial list)

Developing Software

There are specific phases in the process of developing software including design, development, test and integration, and delivery and maintenance. But before any work can begin, a complete understanding of what the program is supposed to do is required. This is derived from the project or program requirements.

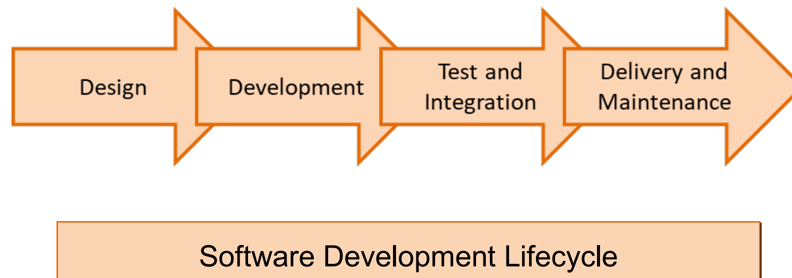
Requirements

The requirements for a computer program detail what the program is supposed to do. How it will do what it is supposed to do will be determined as the design phase is completed during the software development phase. ***Requirements Decomposition*** is the act of discovering in detail from the requirements what the program is to accomplish. This process also assists in decomposing the project into manageable "chunks" in terms of the schedule and team assignments for development. Once the requirements are vetted, the software development lifecycle begins.

Software Development Life Cycle (SDLC)

The Software Development Life Cycle includes the steps necessary to design, develop, deliver, and maintain the computer program. Although the phases follow one another and are often accomplished by different teams, they overlap to a degree as questions and issues arise in the process. As an example, a

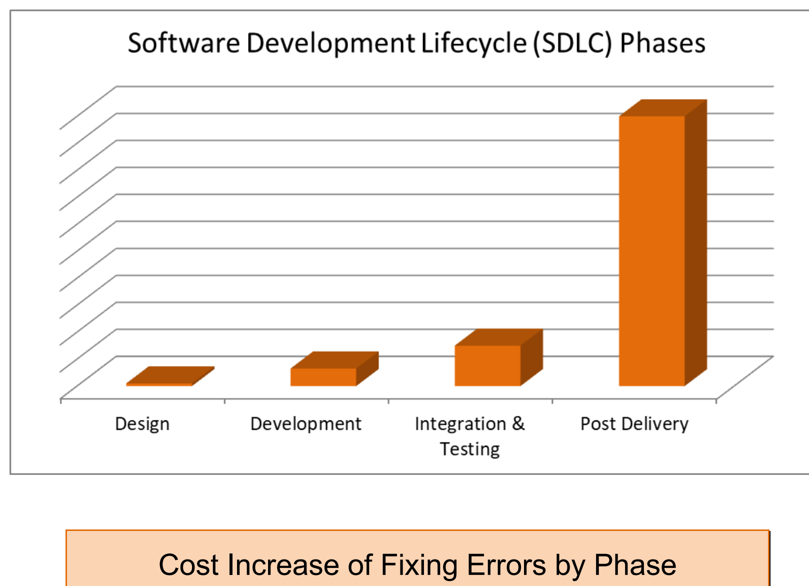
developer may meet with a design engineer to clarify information in the design, or a Test Team member may contact a developer regarding test results.



Software Development Lifecycle

Design

As the requirements are decomposed and documented, the design phase begins, and the break-down of required tasks and logical steps in the program are developed. Design is a very important part of the software development cycle because of the cost increase of changes and fixing errors further on in the process. This is highlighted in the chart below from the IBM Systems Sciences Institute.



Cost Increase of Fixing Errors by Phase

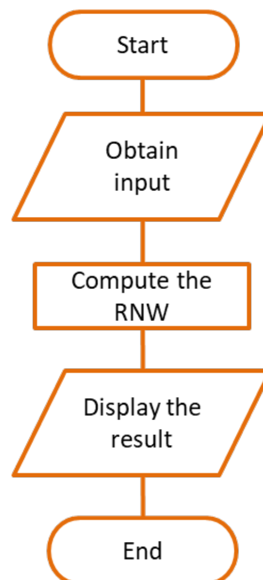
Software engineering tools that assist in the design (and development stage as well) include ***pseudocode*** (sort of code). Pseudocode is a short-hand version of the order of operations for a program. Consider a requirement that a program

obtain user input, compute Recommended Net Worth, and display the results. The pseudocode for the solution might be:

Step 1	Start the program
Step 2	Obtain age and salary information
Step 3	Compute the RNW (age x salary divided / 10)
Step 4	Display the output
Step 5	End the program

Pseudocode

Since we think in pictures and not text, a ***flowchart*** often provides a faster and clearer depiction of the ***algorithm*** (logical steps to the solution). A flowchart is a diagram of the steps in a program. Various geometric shapes are used to indicate different processes. The order of operations is typically top down, and lines with arrows can be used to indicate the order.



Flowchart

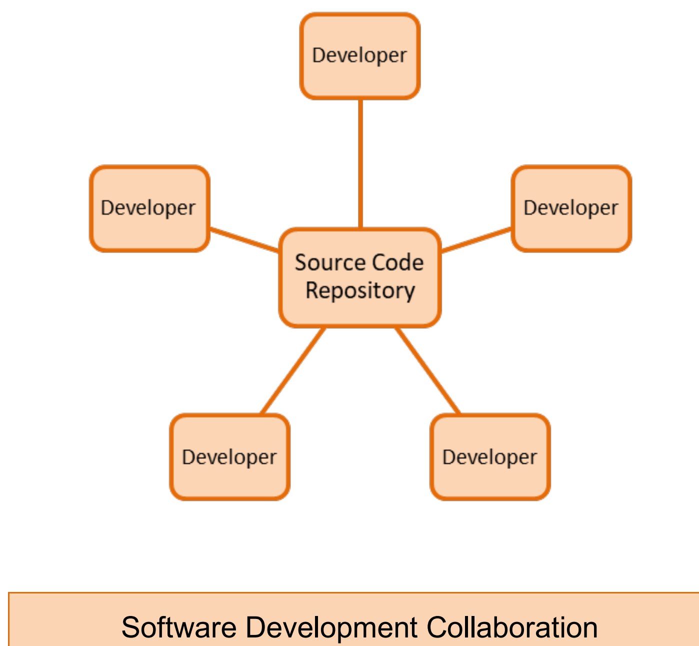
Flowcharts can ensure that steps in the process haven't been overlooked and that there is a complete understanding of the operational flow of the program. It is also common for large organizations to divide the design and development tasks

among teams or to subcontract software development out-of-house (to another company). In these instances, flowcharts are often required to be delivered to the development team or subcontractor together with specific requirements.

Many software engineers use a combination of tools. Pseudocode may be used for a high-level description of the program or a program area, and a flowchart might be used for more complex sections. Either way, the goal is to have a comprehensive understanding of the requirements at every level to ensure that the final product meets the requirements.

Development

Once a design is complete (or nearly complete since some aspects of the solution may not be knowable during design), the development phase begins. The development phase includes writing the code that will be executed to produce the desired result and meet the requirements. Often the development of a program is divided among multiple programmers and requires collaboration and regular discussion to ensure a cohesive solution. To manage software development projects and enable multiple people to work on the same program at the same time, a ***Configuration Management System (CMS)*** is used with a source code repository that stores and maintains all of the program files.



Software Development Collaboration

Programmers access this repository to obtain a copy of a file and add functionality or make modifications to the code. The code is written in the copied file, and this changed file is tested with the other files in the source code repository. After testing, the modified file is placed into the repository and is used by all of the other programmers in place of the original file. The original file is retained by the configuration management tool as a version control mechanism.

If a new file needs to be created, it is created in the configuration management tool and added to the source code repository when completed and tested. CMS tools provide for collaborative development, and version control of the files and the overall project, and many industries and clients require their use.

Many configuration management systems have integrated suites that include: scheduling and tracking, task assignment, defect reporting, and issue tracking systems. In addition, tools for software teams and software project managers are commonly used in industry to plan and measure project progress, and to provide visibility into the design, schedule status, cost, and quality of the code.

Software Development Processes

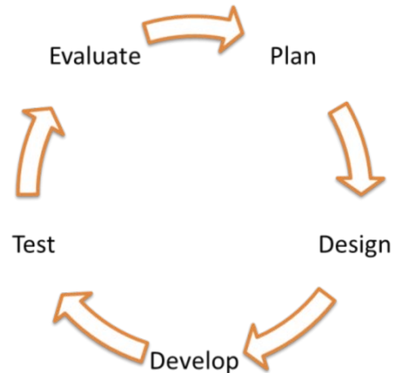
For the software development phase, the ***Agile Development Process*** is a popular method in use today. Agile processes go by various names, but all are iterative and incremental software methodologies. This iterative process of developing software is commonly referred to as ***Iterative Enhancement*****.**

- Scrum – regular meetings, with periodic cycles called sprints
- Crystal - methodology, techniques, and policies
- Dynamic Systems Development Method (DSDM)
- Lean Development
- Feature-Driven Development (FDD)

Agile Software Development Methodologies

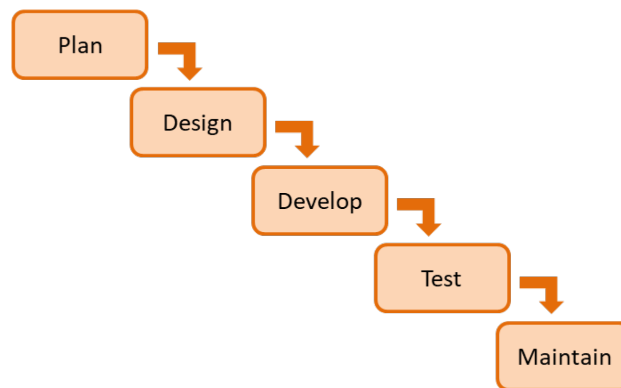
A key component of the Agile Development Process is the ***sprint*** (the development period between status meetings). Sprint status meetings (scrums) are review and planning events that occur regularly (typically weekly). Tasks

completed from the previous sprint plan are reviewed, and completed work is demonstrated to stakeholders for feedback and approval. The tasks that were not completed from the previous sprint plan are reviewed with a course of action (re-plan). The scope of work that will be completed during the next sprint cycle is planned, and engineers are assigned to the tasks.



Agile Development Process Phases

Another popular process for software development is the ***Waterfall model***. The waterfall model uses the same phases, but they are sequential. The phases of the process may overlap to a degree, but are non-repeating and each phase depends on the completion of the previous phase. The Waterfall model was used extensively in the past, but has been replaced for the most part in recent years by the Agile Process.



Waterfall Development Process Phases

Test and Integration

The next phase in the software development life-cycle is integration and testing. In the test phase, the programmer runs the program to ensure that there are no errors in the code, and that it performs correctly (meets the requirements). In large organizations, there is a Test and Integration Team responsible for this phase. Any errors found by the Test Team are relayed back to the developer.

The two types of errors that are looked for during the preliminary test phase are syntax errors and logic errors.

Syntax errors have to do with language specific rules like indentation and punctuation and are found by the compiler or interpreter and the code will not be compiled or executed. These errors would be resolved during development.

Logic errors are errors in the algorithm or the way that the algorithm was written by the programmer. For example, if the requirement is that the program multiply a number by two only if it is greater than ten, and the programmer writes the code so that a number is multiplied by two if it is less than ten, that would be a logic error.

As mentioned above, if the code is part of a larger project, it must be integrated into the overall project and tested again with the complete program. The configuration management system provides this capability as well.

Delivery and Maintenance

The final phase of the software development life-cycle is the delivery and maintenance phase. In this phase, the program is delivered to the client or customer and a period of maintaining the program begins. Maintenance of a program would include updates or patches that fix errors or security issues found after delivery, or upgrades that provide additional functionality or capability. Updates to software programs are commonplace today.

Ergonomics

The set-up or arrangement of computers and furniture to minimize the risk of injury or discomfort from repetitive motion and working in a stationary position from extended periods is a field of ergonomics.

Chapter 1 Review Questions

1. Computers are simply ___________________ devices.
2. The physical parts of the computer are referred to as _____________.
3. The CPU is considered the ___________ of the computer.
4. The CPU performs basic ____________ and controls computer _________.
5. The main memory in a computer is often referred to as _________.
6. Main memory is volatile and is erased when the computer is _____________.
7. __________ Storage device memory is non-volatile and is retained when the power is turned off.
8. A computer keyboard, mouse, and camera are examples of _________ devices.
9. Computer monitors, speakers, and printers are examples of __________devices.
10. Computers follow a 3-step process of _________, _________, and _________.
11. Sets of instructions for the computer are commonly referred to as _________.
12. _____________ and _____________ are the two basic types of software.
13. The language of computers is a _________ language.
14. The smallest information representation in computing is a _____ or binary digit.
15. A binary digit can have a logical state of _____ or _____.
16. A Byte is a combination of ________ bits that are either one or zero.
17. The number represented by 0110 1001 is _______.
18. The binary representation of the number 255 is ___________.
19. The names of the two low-level languages are __________ and __________.
20. A Machine cycle consists of ________, ________, ________, and ________.
21. A 2 GHz (gigahertz) processor can execute __________ instructions per second.
22. High-level languages make programming a computer _____ and more _______.
23. Python is a _______ -level language.
24. A (n) ___________ translates a high-level language into a separate machine language program.
25. A (n) ___________ reads, translates, and executes a program one line at a time.
26. The characteristics and rules that must be followed when writing programs in a high-level language are called ___________ and ___________.

27. Words that are reserved in a programming language are called ____________.
28. The rules for combining symbols, operators, and punctuation in a programming language are referred to as the languages __________.
29. Plan, design, develop, test, and evaluate are the five steps in the ___________ development process.
30. A shorthand version of the steps to complete a task in a computer program is called _______________.
31. A set of logical steps taken to complete a task is called a(n) _____________.
32. The act of discerning in detail from the requirements what the program is to accomplish is called ___________________ ____________________.
33. The four steps in the Software Development Life-cycle are ________, ________, ________, and ________.
34. The two types of programming errors are _________ and _________ errors.

Chapter 1 Exercises

1. Explain the differences between main memory and secondary storage.
2. List at least three (3) input devices.
3. List at least three (3) output devices.
4. List the two (2) types of software.
5. Write the word Python in binary.
6. Write the binary representation for the number 176.
7. List the two low level languages.
8. List the four steps in a machine cycle.
9. List the four steps in the Software Development Life Cycle
10. Write the pseudocode for the steps required to determine the total price for some number of items priced at $9.00 each with a 7% sales tax.
11. Draw a flowchart of the steps in exercise 2 above.
12. What is the purpose of a source code repository?
13. List the five phases of the Agile Development cycle.
14. Explain the difference between logic and syntax errors.

15. Obtain a copy of Python with IDLE. Python is free to download and use, and can be installed and run on any computer. The website URL and steps to obtain and install Python are provided in Appendix B.

Chapter 2

The Python Shell and IDLE

Python is a high-level, general-purpose language. It was created by Guido van Rossum and introduced in 1991, and emphasizes code readability and is similar in many respects to pseudocode. The name Python comes from the famous British comedy Monty Python's Flying Circus. Python is an interpretive language in that it uses an interpreter to translate the code one line at a time and execute it. Interpretive languages tend to be slower than direct native machine code, and can be reverse engineered more easily, so their use should be restricted to areas where this is not an issue.

Python supports procedural programming and object oriented programming. It is a simple yet strong language with many supporting ***libraries*** (code written by others that we can use) as well as a standard library containing extensive capability.

Python Versions

Python is Open Source and is developed under an OSI-approved open source license, which makes it free to use and distribute including for commercial use. The current version in use is Python 3.9.5 which is available from Python.org for Windows and Mac OS X along with additional information and Tutorials. The current version of Python 3 will not run on Windows 7 or earlier versions of the

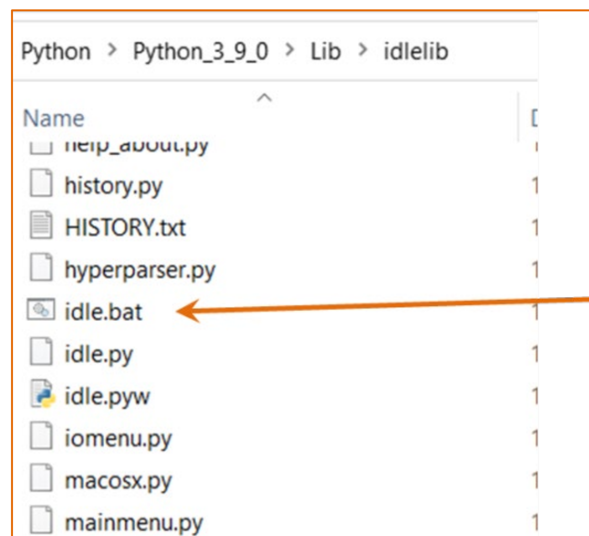
operating system. Python.org also maintains versions of Python 2 which has been used for more than a decade with 2.7.18 currently available. The two versions (2 and 3) are not compatible. In other words, code written in Python 3.x.x will not run in Python 2.x.x. Python 3 (and above) is used exclusively in this text. Additional information is available at Python.org.

Installing and Running Python

Python is free to download and use, and can be installed and run on any computer. The website URL and steps to obtain and install Python are provided in Appendix B. Installing Python should be completed before continuing in this chapter.

The installation for Python includes the interpreter for executing Python code, the IDLE Integrated Development Environment for developing programs, and many libraries that provide functionality without having to write the code.

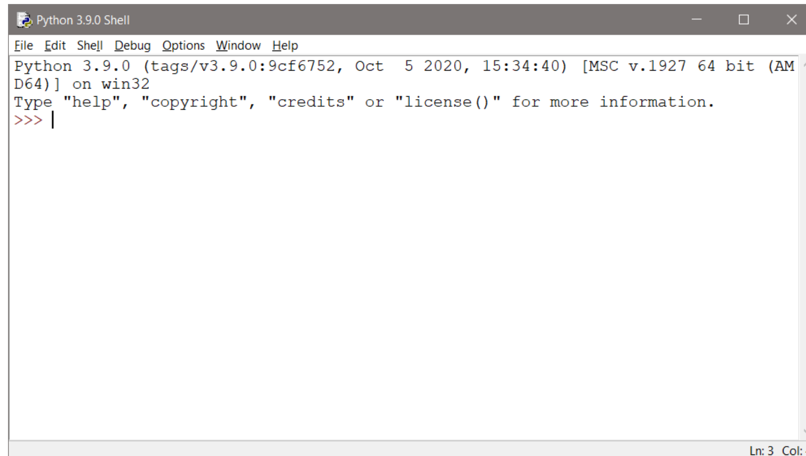
Once Python is installed, IDLE is launched using the idle.bat which is a batch file in the Lib/idlelib directory. Displaying file extensions should be enabled on the computer to ensure the correct file is double-clicked and to become familiar with file extensions. A short-cut can be created at a higher level to eliminate drilling down into the sub-directory each time IDLE is launched.



Python Batch File Location

The Python Shell

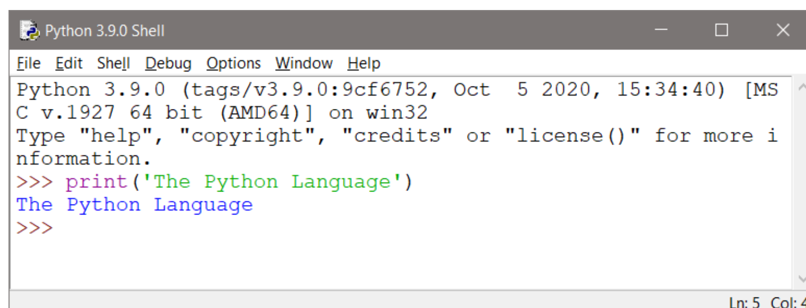
When double-clicked or accessed from the short-cut, idle.bat will open the Python shell shown below. This is the Python interpreter in interactive mode.



The Python Shell

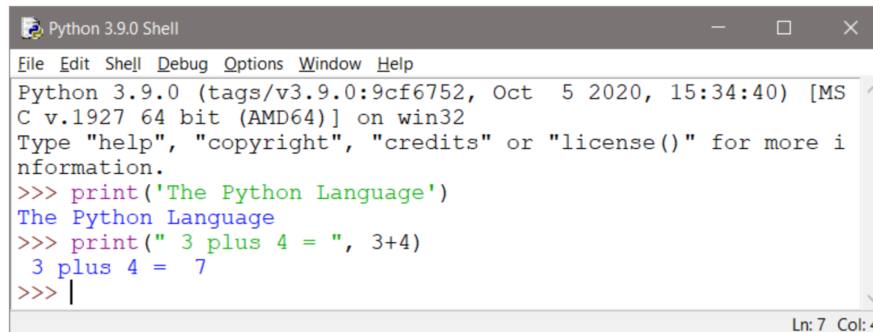
Python statements can be run directly in the shell. Typing a single line in the shell and pressing *Enter* will execute the line using the interpreter. A print statement is an easy way to demonstrate this and to highlight a few things.

Ex. 2.1 - A line of code entered into the shell will execute when *Enter* is pressed. The prompt waiting for input in the shell is '>>>', and the words in the statements are color coded to highlight that they are different items. Also notice that the '>>>' prompt appears again after the output waiting for an additional statement to execute.



Executing Code in the Python Shell

Ex. 2.2 – The added line of code in the next example includes an equation. When *Enter* is pressed, the text and the result of the equation are displayed. Notice that for the first print statement, the text to output is surrounded by single quotes, but in the second print statement they are double quotes. In Python either can be used, but it is important to be consistent. Single quotes are used in this text.



Executing Code in the Python Shell

Programming in the shell and executing one line works well for code snippets or examples, but the goal is to write complete Python programs. The shell simply uses the interpreter to execute the line that was typed when *Enter* is pressed. Files will be used to write and execute more complex programs using the interpreter in ***script mode*** in which the interpreter reads the contents of a file to execute multiple lines of code (a program).

The IDLE Integrated Development Environment

IDLE, which stands for Integrated Development and Learning Environment, is intended to be a simple Integrated Development Environment (IDE) that is cross-platform (runs on multiple operating systems and computers), and is suitable for starting out in Python especially in an educational setting. IDLE provides a multi-window text editor and Python shell with syntax highlighting and smart indent. IDLE features an integrated debugger with breakpoint capability and call stack visibility which will be covered later. IDLE is free to download and use (it is installed with Python), and it does not have the host of features that tend to clutter many IDEs with limited benefit.

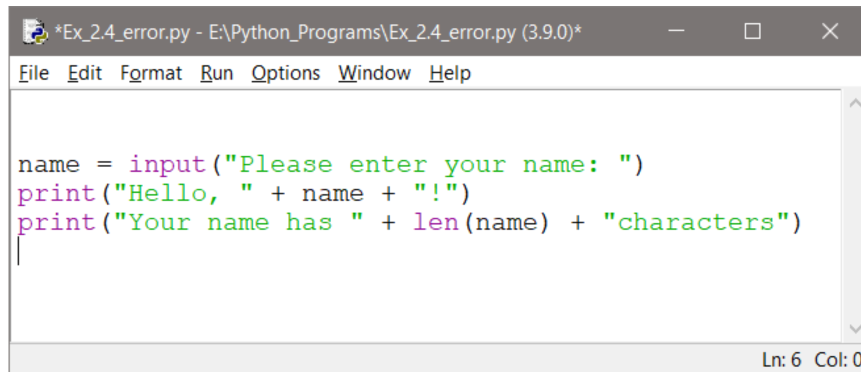
Python Shell with IDLE and Code Files

Starting the IDLE Text Editor

Ex. 2.3 – The IDLE editor is started by choosing File | New File from the shell menu. The new window is the edit window where sequences of Python commands are entered and then executed. Unlike the shell where the lines execute when *Enter* is pressed, the lines in the editor will be executed as a group to form a program. The title of the window below will change from "untitled" when it is saved. The drop-down menus provide basic IDE functionality.

New File Window

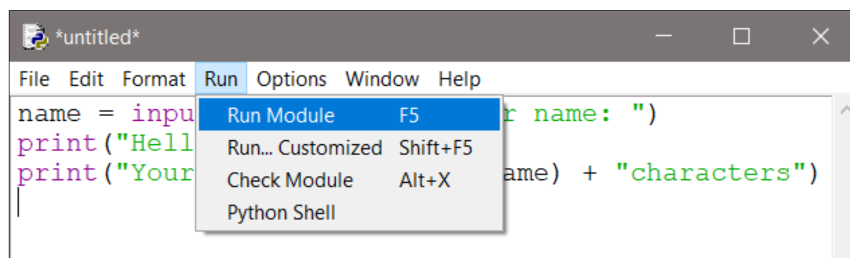
Ex. 2.4 – The lines of code in the edit window below have an intentional error. Errors in programming are called ***bugs*** and are removed by "debugging" the program. Type the lines into the file exactly as shown, and running the program will reveal one way that IDLE indicates errors. Also note that pressing the *Enter* key causes a line feed in the file instead of executing the line of code.



Edit Window Syntax Error

Running Programs in the IDLE Text Editor

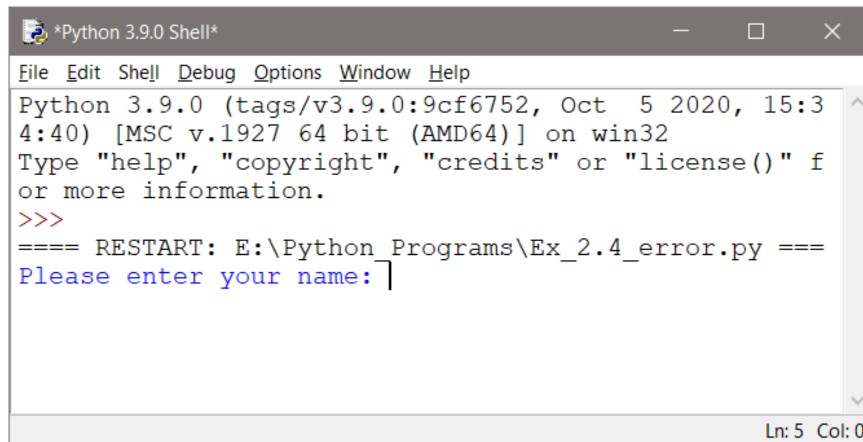
Running the program in the IDLE editor is accomplished by pressing the function key *F5* on the keyboard, or selecting *Run* and then *Run Module* from the menu as shown below.



Running a Program File

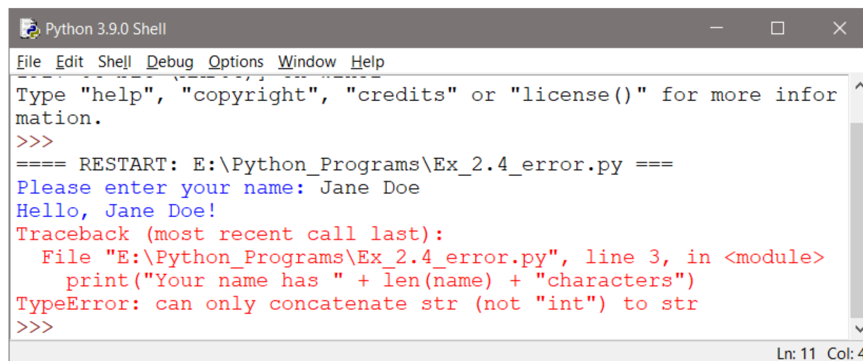
IDLE will force saving the file before it will run the program. Choose an appropriate name and the ".py" file extension will be added. After the file has

been saved, Python will run the program and the prompt to enter your name will appear in the Python shell.



Program Output in the Python Shell

When a response to the prompt is entered and *Enter* is pressed, the intentional error in the code will surface. The red text in the Python shell provides information about the error.



Traceback and Error Information

The error information includes a "***Traceback***" of the function call or calls causing the error, the file name and line number for the error, as well as the line of code itself, and the type of error. The line numbers for the program can be seen at the bottom right of the IDLE editor window, or selected from the Option menu.

The error is the result of the function call *len(*name*)* returning an integer which is the length of the string passed to it in name. Python cannot concatenate (join) an integer onto the string *"Your name has "*, so execution stops and the error message appears. To correct this, the return value from the function call to *len()* can be converted to a string using *str* as shown below. Python can then concatenate (join) the resulting string to the literal string of characters before it and concatenate the literal string after it. These operations will be covered later.

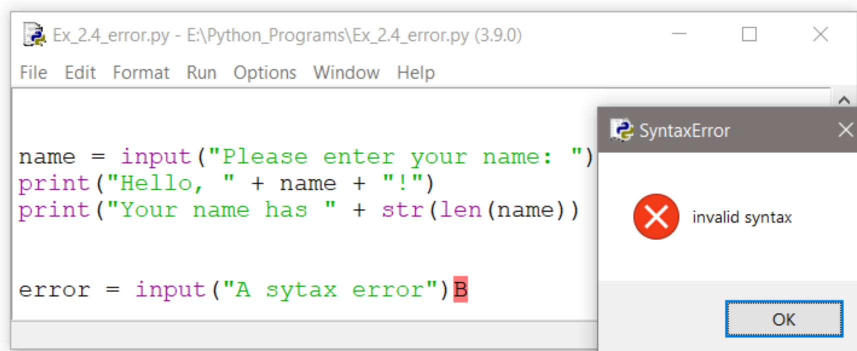
Ex. 2.5 – After correcting the code, saving it, and running it again (*F5*), the program now runs correctly. Notice that the space between Jane and Doe was counted by the *len* function.

```
name = input("Please enter your name: ")
print("Hello, " + name + "!")
print("Your name has " + str(len(name)) + "characters")
```

```
Please enter your name: Jane Doe
Hello, Jane Doe!
Your name has 8 characters.
>>> |
```

Corrected Error Example

Another type is a syntax error. In many cases IDLE will highlight the actual code where the error occurs by boxing it in red and producing an error dialog. In the example below, the character "B" was erroneously typed at the end of the line.



Syntax Error and Dialog

Programming Errors

Syntax errors and Logic errors were described in Chapter 1, and are summarized below with the addition of a third type of error in programming - the Runtime error.

- ***Syntax error*** – incorrect use of language specific rules like indentation and punctuation which will be found by the interpreter and the code will not execute
- ***Logic error*** – the program runs, but does not perform the task it was intended to perform or it produces incorrect results
- ***Runtime error*** – logic error that causes the program to stop executing

The most common errors in programming include misspelling words, forgetting closing quotes, and forgetting closing parentheses. The interpreter helps by highlighting these types of errors, but logic errors must be found by thoroughly testing the program.

Exiting Python and IDLE

To leave IDLE, just close the windows.

Since IDLE insists that files are saved before each execution, it's hard to lose changes when exiting IDLE.

To be really safe, save the program manually before closing the editing window.

Choose "File" on the menu bar and "Save" from the drop-down menu or use *Control-S*.

Exiting Python with IDLE

Chapter 2 Review Questions

1. Python is an ___________________ language.
2. Python uses an interpreter to _______ and _______________ lines of code.
3. The Python interpreter executes _______ of code at a time.
4. In ____________ mode, the interpreter executes a single line at a time from the shell when the Enter key is pressed.
5. The interpreter reads files and executes their contents in _________ mode.
6. Errors in programing are typically referred to as __________.

Chapter 2 Exercises

1. Complete chapter example 2.1 in the Python shell and provide a screen capture of the window with the output.
2. Complete chapter example 2.2 in the Python shell and provide a screen capture of the window with the output.
3. Complete chapter example 2.3 and 2.4 in a separate file (IDLE Editor) and provide a screen capture of the file with the code and the error in the shell.
4. Complete chapter example 2.5 by correcting the program from example 2.4 and provide a screen capture of the file with the code and the output in the shell.
5. Explain the contents of a Traceback when an error occurs.
6. List the three (3) types of programming errors.

Chapter 3

Getting Started in Python

As mentioned previously, five minutes of design time will save hours of programming. This is because a plan has been determined before any code has been written, bugs discovered during the design phase are easier to find and faster to eliminate, and the solution is viewed as a comprehensive program. Very often one part of a program has a direct impact on other parts of the program. A poor design in one area can impact other areas, introduce bugs, require hours of debugging, and force re-writing of code and increasing costs.

Comments

In addition to the tools mentioned in Chapter 1 for designing and developing software, comments within the code can be helpful and are often required. ***Comments*** in programming are lines of code that are not executed, they are provided for human readers, and are used to clarify values or sections, or explain complex operations. This is important because most software is maintained, updated, and expanded. Code is written once, but is read many times, and the person who wrote the code may not be the person making the modifications, or the person who wrote the code may not remember why a section was written a certain way or why a specific value was used. Adding comments to code while it is being written can save hours of reading through the lines later when the code is being changed.

Comments are also used as a development tool. Pseudocode is written in the program as a comment to act as a place-holder or reminder that will be replaced later by actual code.

The source code written in programs is often referred as SLOCs, and includes comments as well as executable lines. The executable lines by themselves are referred to as ELOCs.

SLOC – Source Line of Code (include all text)

ELOC – Executable Line of Code (omits comments)

Comments in Python can be single-line or multi-line. Single-line and end-of-line comments begin with the pound sign (hashtag or octothorp), and multiline comments are surrounded by three single or three double quotes. Comments will be ignored by the interpreter, and most IDEs including IDLE will color code comments to highlight them. Comments add clarity and explain complex areas of the program.

```
# This is a single line comment in Python

print("Hello")    # This is an end of line comment

# This is a multiline comment in Python
# using the pound sign

"""
This is a multiline comment in Python
using three sets of double quotes
"""
```

Comment Styles in Python

Displaying Output

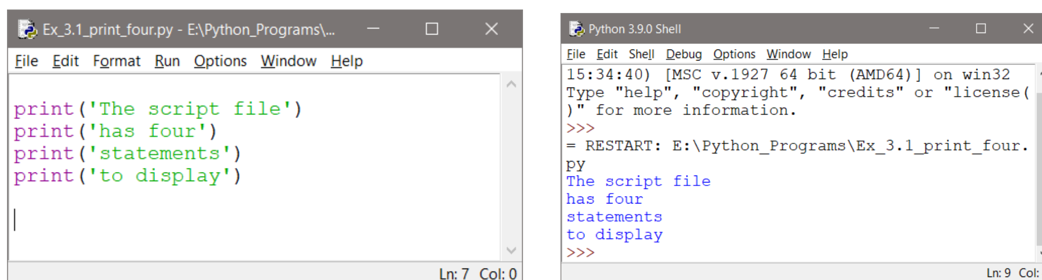
The ***print*** function in Python displays output to the shell. The text inside the parentheses and quotes is a ***string*** (group of characters) and is an ***argument*** passed to the print function (any piece of data passed to a function is referred to as an argument). The argument contains the item or items to display. Single or double quotes can be used with string arguments, and the print function will automatically add a line feed in the output.

A function is code that exists and must be ***called*** in order to execute. In the example, the print function is being called and the argument being passed to the function is "*The Python Language*". The function executes and displays the text.

```
>>> print('The Python Language')
The Python Language
>>>
```

The previous example ran in the shell, but to use the interpreter in ***script mode***, a new file is created using File | New from the shell menu. Example 3.1 displays four lines of output by calling the print function four different times and passing four different strings of characters. The print function adds the line feeds to the output automatically.

Ex. 3.1 – The print function is called four times in this program with different arguments passed to the function each time. The shell window on the right below shows the output when the program in the editor on the left runs.



Program File and Shell Output

Strings

In the previous example, four different phrases (strings) were passed to the print function. A ***string*** is a sequence of characters and is the ***str*** data type in Python. When a string is written in the code as in Example 3.1, it is referred to as a ***string literal***. The example also uses single quotes surrounding the strings, but double quotes can be used. The PEP 8 Style Guide for Python does not provide a recommendation, except to be consistent. There are however cases when one choice is required. When an apostrophe or quotes are part of the literal string to be output, they must be accommodated. This is shown in the next example.

Ex. 3.2 – To display a single quote (apostrophe) in the output, the string is surrounded by double quotes. To include double quotes in output, surround the string by single quotes. To display both, use three sets of quotes.

```
print(" Double quotes for an apostrophe like the word can't ")
print(' Single quotes for double quotes like "Quotes" ')
print(''' I'm displaying "both" ''')
```

```
 Double quotes for an apostrophe like the word can't
 Single quotes for double quotes like "Quotes"
 I'm displaying "both"
>>> |
```

Including an Apostrophe or Quotes in Output

Note: Escape sequences covered later can also be used to include an apostrophe or quotation marks in the output.

Variables

In the example in chapter 2 that was used to display errors, a name was entered into the program and was stored in a ***variable*** called name (repeated below). The input function extracted what was entered on the keyboard when *Enter* was pressed, and ***assigned*** it to name. This is called an assignment statement, and the equal sign is the ***assignment operator***. The right side of an assignment statement is evaluated first, and the result is assigned to the left side.

```
name = input('Please enter your name: ')
```

Variable Assignment from *input*

Variables are used to allocate memory and store information that the program will use. They are called variables because what they store can vary as the program runs. A variable name is a name given by the programmer to a piece of data that is stored in memory, and the assignment operator is used to assign a value to a variable. In the example below, the value 15 is assigned to a variable declared as number. This ***defines*** the variable. The value 15 will be stored by

the computer and it will be accessed (used by the program) by its' name number. The second line prints the value stored in number.

```
number = 15

print(number)
```

Notice that there are no quotes around the word number in the print statement. The variable number is passed to the print function and the value that was stored in memory by the assignment operator will be displayed in the shell as shown below.

```
15
>>>
```

Programs often use multiple variables to store multiple values and each of the variables must have a distinct name. Notice the underscore used in the variable names below. Although the PEP 8 Style Guide lists multiple styles for naming variables in Python, none is recommended or preferred. The following style will be used in this text. If the variable name is one word, it will be all lowercase. If two or more words are used, they will be lowercase and separated by an underscore.

```
number = 15
print(number)

first_variable = 20
print(first_variable)

second_variable = 30
print(second_variable)
```

```
15
20
30
>>>
```

Defining Multiple Variables

Python is case sensitive and case errors as well as misspellings will be caught by the interpreter and a Traceback error will identify the line number for the error.

In the next example, number is defined with all lowercase letters, but an uppercase letter is used in the print statement.

```
number = 15
print(Number)    # not the same as number
```

When the lines of code are interpreted and run, the error displayed in the shell is a NameError and indicates that Number was not defined. The interpreter doesn't know anything about a variable called Number. It only knows about number.

```
NameError: name 'Number' is not defined
>>> |
```

Case Sensitivity - Undefined Variable Error

A similar error occurs when a variable has not been assigned a value. Python does not know what type of data to store without an assignment statement. It determines how to store the data for a variable in memory based on the type of data it is assigned. In addition to the string data type (and others), the Python numeric data types include: int (integer), and float (floating point number). In the code below, the variable is declared without assigning it a value (defining it), and it is then passed to the print function. Since a value was not assigned to the variable, it is undefined to the interpreter.

```
first_variable  # not defined with an assignment

print(first_variable)
```

The interpreter error message indicates that the variable is not defined since it was not assigned a value.

```
NameError: name 'first_variable' is not defined
>>> |
```

Undefined Variable Error

In the code below, three different variables are assigned different types of data.

Ex. 3.3 – Defining variables with different types of data

```
var1 = 15              # an integer
print(var1)

var2 = 12.3            # a foating point number
print(var2)

var3 = 'some text'     # a string
print(var3)
```

```
15
12.3
some text
>>>
```

Defining Different Data Type Variables

In Python, a number can be assigned to a variable and then a string can be assigned to the same variable. Python will handle the change. The program below assigns an integer to number, then a string, then an integer again, and uses the variable in an equation before the final call to the print function.

Ex. 3.4 – Assign the variable number different types of data

```
number = 15              # assign 15 to number
print(number)

number = 'Now a string' # assign a string
print(number)

number = 2               # assign a number again
number = number + 2      # add 2 to number
print(number)
```

The output shows the different data types stored in number at different points in the program, and that the addition was handled by Python without an issue.

```
15
Now a string
4
>>>
```

Re-assigning a Variable a Different Type

Data types categorize values in memory: ***int*** for integers, ***float*** for real numbers, and ***str*** is used to store strings. Floating point numbers are stored with double precision although the data type *double* is not used in Python. In addition the ***bool*** data type can be assigned either True of False. The table below shows the basic data types in Python. Others will be covered later.

Type	Description	Example
int	integers (whole numbers)	3, 18, 56, 1234
float	real numbers (floating point)	7.56, 12.345
str	sequences of characters (strings)	“Python”, “two words”
bool	logical values (Boolean)	True, False

Table 3.1 - Python Basic Data Types

Variable Names

When naming variables, there are a few rules that need to be followed:

- none of Python’s key words can be used as a variable name
- there cannot be any spaces in the names
- the first character must be a letter (or an underscore)
- uppercase and lowercase letters are distinct

The convention for naming variables in Python is to use all lowercase letters for single word names like interest or balance, and for multi-word names to include underscores between words as shown here.

interest_rate or daily_average

In addition, the name of a variable should describe the data that it stores. Most programming standards require descriptive names for variables, and a longer name is usually better. Using names like var or pd are ambiguous and make maintaining the code more difficult. If a comment is needed to describe a variable, then the name of the variable is probably inadequate.

Table 3.2 below lists some good and bad examples of variable names.

Name	Comment
x	does not describe the data being stored
6pack	cannot begin with a digit
sixPack ✓	proper, but not preferred
my_num ✓	proper, but ambiguous
how?many	can only contain letters, digits, and underscores
temperature ✓	proper naming
gross_pay ✓	proper naming

Table 3.2 - Variable Naming Rules

Named Constants

A named constant variable is a value in programing that is not changed by the program. Named constants are defined using all uppercase letters with underscores between words. They are used to eliminate magic numbers, and to ensure that a specific value is used throughout the program.

A ***magic number*** is a literal number in a program without an obvious meaning. When a program is being modified and an expression uses a literal value, it may be difficult to determine what the number means even by the original programmer. As an example, the following line appears in a program and the meaning of 3963.2 is unknown. Since the equation results in a circumference, it appears to be a radius.

```
circumference = 2 * PI * 3963.2        ?
```

By using a named constant in the code, the meaning is clear. It is defined and then used in place of the literal number wherever needed in the program.

```
EARTH_RADIUS = 3963.2
circumference = 2 * PI * EARTH_RADIUS
```

Named constants are also used to ensure that the same value is used throughout the program and by all programmers. As an example, if multiple programmers are working on a program that calls for them to use the radius of the earth in various equations, they can use the named constant instead of typing in different values. The earth is not a sphere and there are multiple values for its radius.

Named constants also prevent typographical errors when the same value is being used multiple times. In addition, when a new value is needed for the constant, the change is made in a single place in the code. As an example, the scientist overseeing the program using EARTH_RADIUS decides that the equatorial radius being used in the program should be changed to the pole radius of 3950.0. It will only need to be changed to the new value in one place in the code. This eliminates the possibility of typographical errors when changing all of the equations that use the value, time spent debugging the program if an equation is overlooked and the output is incorrect.

Printing Multiple Elements

When multiple arguments are passed to the print function, the plus sign or comma is used depending on the data types. A comma is used when a literal string and numeric value are being displayed as shown below.

Ex. 3.5 – the literal string and numeric value are separated by a comma.

```
num = 1000
print('The number is', num)

print('The number is', 123)
```

```
The number is 1000
The number is 123
>>>
```

Printing Strings and Numeric Values

Notice in the print statement above that there is not a space after the word “is”, but it is included in the output. The print function automatically adds a space which is the default separator (when none is provided) for multiple arguments.

Ex. 3.6 – the print function and spacing

In this example an integer variable is surrounded by two string literals. Commas separate the arguments being passed to the print function, and they are automatically separated by spaces in the output. This is the default separator for the print function to use between items when none is provided, and it can be eliminated or changed.

```
tics = 123
print('Today we sold', tics, 'tickets')
```

```
Today we sold 123 tickets
>>>
```

Printing Multiple Arguments and Spacing

Separators

The print function automatically places a space between items in the output. When the space is not needed or another separator is required, the argument ***sep*** can be passed to specify the separator to be used. In the first print statement below, *sep=''* is used to tell the print function to separate elements using *nothing* (there is no space between the two single quotes). In the second print statement, the print function is told to use the semicolon as the separator. Any character or string can be used to separate the elements in the output.

```
tics = 123
print('Today we sold', tics, 'tickets', sep='')

print('Today we sold', tics, 'tickets', sep=';')
```

```
Today we sold123tickets
Today we sold;123;tickets
>>>
```

Assigning Output Separators

String Concatenation

The plus operator in Python performs ***concatenation*** which joins two or more strings together to form a single item. In example Ex. 3.7, note the missing spaces between the words in the output.

Ex. 3.7 – string concatenation in output

```
print('First' + 'second' + 'third')

        Firstsecondthird
        >>>
```

Example 3.7a concatenates a string onto the original string that was stored in the variable word1.

Ex. 3.7a – string concatenation

```
word1 = 'first'
word1 = word1 + 'second'

print(word1)

       firstsecond
       >>>
```

String Concatenation

As a technical note, strings in Python are ***immutable***, that is they cannot be changed. In Example 3.7a, Python created a new string containing the combined words and then pointed the variable name word1 to that new string in memory. This is typically not an issue, and the interpreter will free the previously used memory. In addition, Python provides another solution which is covered later.

Formatted Output

When dollar amounts or other numbers requiring decimal places are part of the output, the ***format*** function provides a solution for output formatting. Two arguments are passed to the format function: the numeric value or variable to be formatted and the format specification. The format function returns a string

holding the formatted number which can then be stored as a string in a variable or passed to the print function directly.

Ex. 3.8 – format function and specifiers for output

The format specifiers are enclosed in quotes and follow the item to be formatted after a comma. The first call to print below formats the value 5 with two decimal places, and the second formats a variable to two decimal places. The ".2" in the specifier calls for two decimal places to be used in the output (this will cause rounding), and the "*f*" indicates formatting for a floating-point number.

```
print(format(5, '.2f'))

num = 1.2345

print(format(num, '.2f'))
```

```
5.00
1.23
>>>
```

Formatting Decimal Output

Ex. 3.8a – the format function rounds numbers

When required to fit within the number of decimal places specified for formatting, the number will be rounded (down from 5 and up from 6).

```
num = 123.98765

print(format(num, '.4f'))
print(format(num, '.3f'))
print(format(num, '.2f'))
print(format(num, '.1f'))
```

```
123.9877
123.988
123.99
124.0
>>>
```

Formatted Output Rounded

Values can also be formatted when assigning them to variables. The formatted variables can then be passed to the print function as shown below.

```
text1 = format(1.2345, '.1f')
text2 = format(9.876, '.1f')
print(text1, text2)


num = 9.876
print(format(num, '.2f'))
```

```
1.2 9.9
9.88
>>>
```

Assigning Formatted Values to Variables

When the number being output requires a comma or commas, the specifier (a comma) is added prior to the decimal in the format specification.

```
num = 12345678.99
print(format(num, ',.2f'))
```

```
12,345,678.99
>>>
```

Adding Commas to Output

To format integers, *"d"* replaces the *"f"* in the specifier. Note that using zero for precision has the same effect when an *"f"* is used.

```
print(format(1234, ',d'))
```

```
1,234
>>>
```

Formatted Integer Output

For scientific notation, *"e"* is the specifier. Preceding it with a decimal point and an integer designates the number of places after the decimal to use in the output.

```
num = 12345678.99
print(format(num, "e"))
print(format(num, ".1e"))
```

```
1.234568e+07
1.2e+07
>>>
```

Scientific Notation

Numbers can also be specified as percentages using "%" and the number will be multiplied by 100 and include the "% sign. In the example, zero is used for the number of decimal places, but it can be any digit.

```
stat = 0.27
print(format(stat, '.0%'))
```

```
27%
>>>
```

Formatting Percentage

For columns or right-aligned output, a minimum width specifier is placed before the decimal point. In this example, the width is designated as nine characters wide. A number with more than nine digits would not be trimmed.

```
first = 1234.56
second = 654.32

print(format(first, "9.2f"))
print(format(second, "9.2f"))
```

```
 1234.56
  654.32
>>>
```

Specified Field Width Output

Recall that after the print function executes, it adds a line feed to the output. To suppress the line feed that is automatically added, pass ***end= ‘‘*** to the function.

Ex. 3.9 – suppressing line feeds.

```
print('No lines', end='')    # suppress the line feed
print(' between', end='')    # suppress the line feed
print(' these.')
```

```
No lines between these.
>>>
```

Suppressing Line Feeds

The automatic line feed added by the print function is suppressed above, but spaces were required before the words *between* and *these*. This is because Python simply concatenates (joins) the strings. Consider the print statements below and their output.

```
print('Do we ' + 'want ' + 'spaces?')
print('Do we' + 'want' + 'spaces?')
```

```
Do we want spaces?
Do wewantspaces?
>>>
```

String Concatenation and Spaces

Escape Characters

Example 3.2 showed ways of including apostrophes and quotes in output. Escape characters (sequences) provide another way with some additions. ***Escape sequences*** in programming are special commands that begin with a backslash and can be embedded in strings for output. When used by themselves, they must be surrounded by quotes. They can be used to format output with a tab or line feed, or to include an apostrophe, quotes, or backslash.

In the example program below, spaces have been added around the escape characters for a line feed, single quote, and double quote for clarity.

```
print("tab" + "\t" + "tab")
print("an extra line feed \n ")
print("single quote  \'  ")
print('double quote  \"  ')
print("backslash  \\")
```

```
tab     tab
an extra line feed

single quote  '
double quote  "
backslash  \
>>>
```

Escape Sequences

Escape sequences are seen by the computer as a single character. Table 3.3 lists commonly used escape sequences.

Escape	Result
\n	output a line feed
\t	output a tab
\’	output a single quote
\”	output a double quote
\\	output a backslash

Table 3.3 - Escape Sequences

Keyboard Input

As shown earlier, Python has a built-in function called ***input*** that reads input from the keyboard. The function returns the value entered on the keyboard as a string which can then be converted to the data type desired.

The first line of code below obtains user input as a string and assigns it to the variable words. The second receives a value from the user and converts it and stores it as an integer in the variable integer. The third line receives the user input and converts it to a floating point number and stores it as a float in the variable decimal. The print function with no arguments (empty parentheses) produces an extra line feed. The third print function is using a plus sign for addition of the numbers, not concatenation since they are numeric values.

Ex. 3.10 – keyboard input and conversion.

Notice below that the second call to the print function displays each of the items separated by a space. The third print function call completes the addition of the integer and floating point number and displays the result. With strings, the result is concatenation but the data types for the addition in the print statement are an integer and floating point number. The result is a floating point number.

```
words = input("Enter a string: ")
integer = int(input("Enter an integer: "))
decimal = float(input("Enter a float: "))

print()
print(words, integer, decimal)
print(integer + decimal)
```

```
Enter a string: Hello
Enter an integer: 20
Enter a float: 1.78

Hello 20 1.78
21.78
>>>
```

Keyboard Input Conversion

Type Conversion (Casting)

Converting an item to a different data type is referred to as ***casting***. Casting is accomplished by surrounding the *item* to be cast with parentheses, and preceding it by the resulting data type needed. Note that casting will only succeed if the *item* being cast is valid for the conversion. For instance trying to convert “Hello” to an integer will fail.

In the next example, the variable first is assigned a floating point number (123.45) which is then cast to an integer. In the second part, the variable second is defined with an integer (5) which is then cast to a float. The output shows the successful conversions.

```
first = 123.45
first_as_int = int(first)        # cast to an integer
print(first_as_int)

second = 5
second_as_float = float(second) # cast to a float
print(second_as_float)
```

```
123
5.0
>>>
```

Casting – Converting Data Types

If an attempt is made to cast an item that is not convertible to a numeric value, the interpreter indicates an error.

```
item = 'word'
print(int(item))
```

```
ValueError: invalid literal for int() with base 10: 'word'
```

Invalid Type Error

Arithmetic Operators

Python arithmetic operators include addition, subtraction, multiplication, two types of division, the modulus operator (modulo divide), and exponentiation. The values on the left and right side of the operator are referred to as ***operands***. As shown in Table 3.4 below, the operators for addition and subtraction are the plus and minus signs, and the multiplication operator is the asterisk. The two division operators include a single forward slash for floating point division and two forward slashes for integer division with positive results being truncated

and negative results being rounded away from zero. The modulus or remainder operator is the percent sign, and the exponentiation operator is two asterisks.

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	72 - 12	60
*	multiplication	4 * 6	24
/	division	7 / 2	3.5
//	integer division	7 // 2	3
%	remainder (mod)	17 % 4	1
**	exponentiation	2 ** 3	8

Table 3.4 – Arithmetic Operators

Precedence for operators in Python follows PEMDAS, parenthetical expressions first, followed by exponentiation, then multiplication, division, modulo division, and then addition and subtraction. Operators with the same precedence are handled left to right. The importance of parentheses is highlighted in the examples below in which the result differs depending on the precedence forced by the parentheses.

4 * 2 + 15 / 5 – 2	the result is 9
4 * (2 + 15) / 5 – 2	the result is 11.6
4 * (2 + 15 / 5 – 2)	the result is 12
(4 * 2 + 15) / 5 – 2	the result is 2.6
4 * 2 + 15 / (5 – 2)	the result is 13
4 * (2 + 15) / (5 – 2)	the result is 22.666
(4 * 2) + (15 / 5) – 2	the result is 9

When writing mathematical expressions, including parentheses even when they align with precedence improves readability. Complex mathematical expressions can be broken into multiple statements to simplify the expression.

Mixed-type expression results depend upon the data types in use.

Two int values	the result is an int
Two float values	the result is a float
An int and float	the int is temporarily converted to a float and the result of the operation is a float.

When a float is added to an integer, the result is a floating point number as shown in the example below.

```
number = 15              # stored as an integer
print(number)


number = number + 1.5    # float added to int
print(number)
```

```
15
16.5
>>>
```

Addition with Integers and Floats

The same results occur with subtraction. When an integer is subtracted from an integer the result is an integer. When a float is subtracted from an integer the result is a float, and when an integer is subtracted from a float the result is a float.

```
result = 10 - 5
print(result)

result = 2 - 1.5
print(result)

result = 5.5 - 2
print(result)
```

```
5
0.5
3.5
>>>
```

Subtraction with Integers and Floats

The two operators for ***division*** are a single or two forward slashes. A single forward slash is used for floating point division and two forward slashes for integer division. The division operators override the mixed-type results that were previously covered. Floating point number division using the integer division operator will produce a floating point result, but it is truncated.

Expression	Value of var1	Comment
var1 = 10 // 5	2	an integer
var1 = 10 / 5	2.0	a float
var1 = 5.5 / 5	1.1	a float
var1 = 2.5 // 5	0.0	fractional part discarded
var1 = 5.5 // 5	1.0	fractional part discarded
var1 = -5.5 // 5	-2.0	rounded away from zero

Table 3.5 – Division Operators

The Round Function

To deliberately round numbers, Python has a ***round*** function that will round numbers to an integer or to a specified number of places. The number of decimal places is passed as the second argument to the function.

number = round(9.4)	number is 9
number = round(9.6)	number is 10
number = round(12.2733, 2)	number is 12.27
number = round(12.2755, 2)	number is 12.28
number = round(-2.7777, 2)	number is -2.78

Round Function

The ***modulus*** or remainder operator produces the remainder after division (sometimes referred to as modulo divide). The operand on the left of the

operator is divided by the operand on the right and the result is the remainder. Remainder division is often used to determine if a number is even or odd, and to extract a digit from a number. Python supports remainder division with floating point and negative numbers as well.

```
result = 5 % 2          result is 1
result = 7 % 2.0        result is 1.0
result = -5 % 2         result is 1
result = -5 % -2        result is -1
result = 8.5 % 2        result is 0.5
```

Remainder Operator

For ***exponentiation*** or raising a number to a power, the operator is two asterisks. The operand to the left of the operator is raised to the power of the operand on the right.

```
result = 2 ** 4         result is 16
result = 2 ** 4.0       result is 16.0
result = 2 ** 1.5       result is 2.8284271247...
result = 2 ** 0         result is 1
result = 2 ** -1        result is 0.5
```

Exponentiation

Programming Algebraic Expressions

When converting mathematical expressions into Python code, the translation may require adding operators and parentheses to ensure the correct result. As an example, the expression 3xy in algebra would not be accepted by the interpreter. It would produce a syntax error. The multiplication operator must be inserted as in 3 * x * y. In addition, when an expression contains fractions, precedence requires careful consideration to ensure that operations occur in the correct order. With extremely complex equations, breaking the expression into parts may be the best course of action.

Conversion examples.

3x	`3 * x`
5xy	`5 * x * y`
$x = \frac{3y}{2a}$	`x = 3 * y / 2 * a`
$x = \frac{y + 5}{z - 3}$	`x = (y + 5) / (z - 3)`
$x = \frac{n(n-1)y^2}{2+n}$	`x = (n * (n - 1) * y ** 2) / (2 + n)`

Converting Algebraic Expressions

Breaking Long Statements

Statements in Python can be broken across lines using the line continuation indicator (backslash) shown in the first example below. However, the backslash is not necessary when a statement is enclosed in parenthesis as in the second and third examples.

```
wind_chill = 35.74 + (0.6215 * tempF) - (35.75 * (wind_speed**0.16)) + \
                0.4275 * tempF * (wind_speed**0.16)

result = (item1 + item2 + item3 +
                item4 + item5)

print (“User 1 gross pay is $“, gross_pay,
                “ and the net pay is “, net_pay)
```

Breaking Long Statements across Lines

The next section follows the development of a complete computer-based solution from requirements decomposition and design through development and testing.

A Complete Example – Theater Program

Requirements:

Write a program for a Theater Manager that computes the total sales receipts and profit for an event based on the number of tickets sold at $29.50 each, and the cost to hold an event which is $1,475.00.

The project begins by determining what the program does, what it needs to complete the task (input), the operations it will perform (process), and what it will produce (output).

Program Requirements Decomposition:

1. The program computes total sales and profit for a Theater Manager.
2. The program input is the number of tickets sold.
3. The program computes total sales based on the number of tickets sold and price for each ticket
4. The program computes the profit for the event based on the total sales from tickets and the cost to hold the event.
5. The program will output the total sales amount and the profit.

Pseudocode or a flowchart can help to determine the correct order of operations.

Step 1 Start and announce the program

Step 2 Prompt for tickets sold and store the value

Step 3 Compute total sales - Tickets sold * $29.50

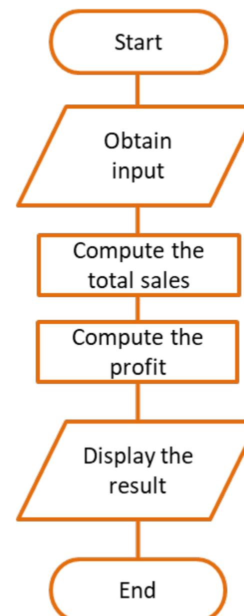
Step 4 Store the total sales in a variable

Step 6 Compute profit – total sales - $1,475.00

Step 7 Store the profit in a variable

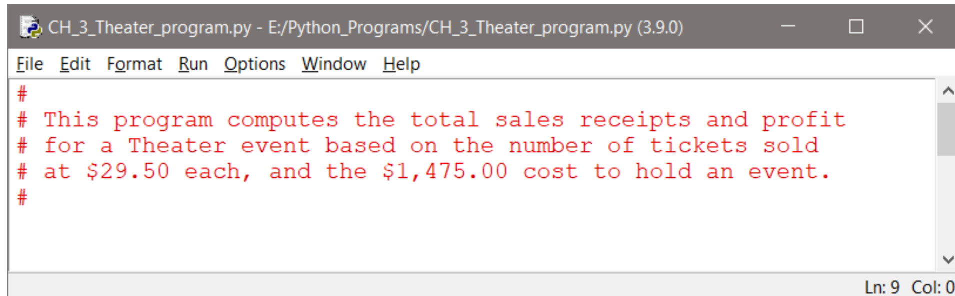
Step 7 Display the total sales and profit

Step 8 End the program



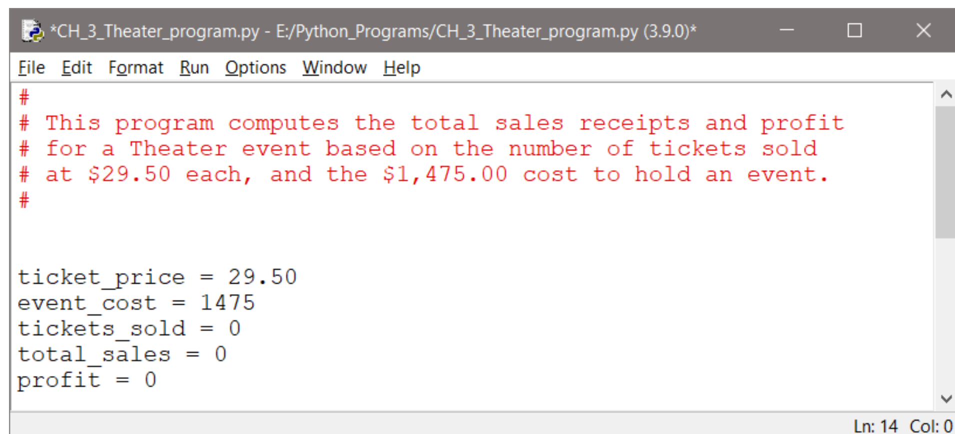
Development

1. Development begins by creating a file, naming it appropriately, and adding a description of the program for future reference.



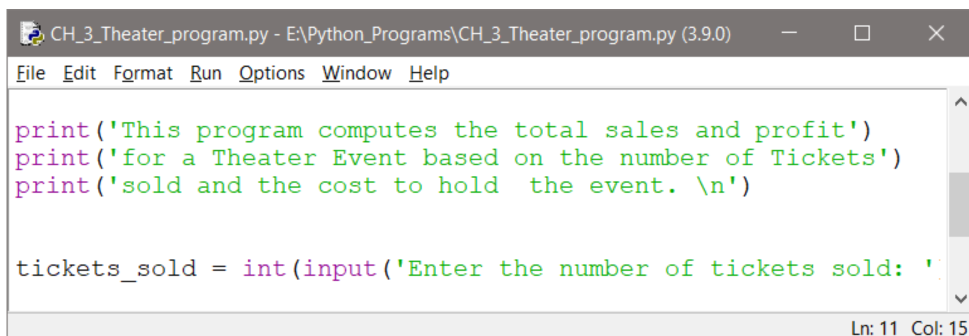
```
#
# This program computes the total sales receipts and profit
# for a Theater event based on the number of tickets sold
# at $29.50 each, and the $1,475.00 cost to hold an event.
#
```

2. The variables needed by the program are defined next. These can be determined by the program input and output, and the processing that will be performed. The ticket price and event cost are provided, the number of tickets sold will be input, the total sales and profit will be computed.



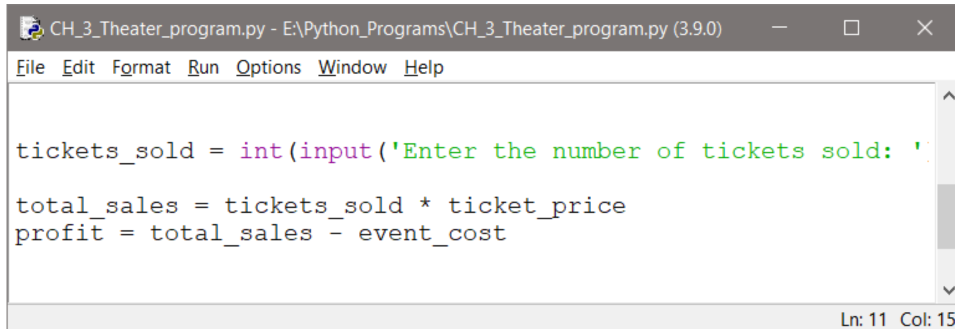
```
#
# This program computes the total sales receipts and profit
# for a Theater event based on the number of tickets sold
# at $29.50 each, and the $1,475.00 cost to hold an event.
#


ticket_price = 29.50
event_cost = 1475
tickets_sold = 0
total_sales = 0
profit = 0
```

3. An announcement to the user of what the program does is added, and the input section which prompts for and obtains the number of tickets sold.

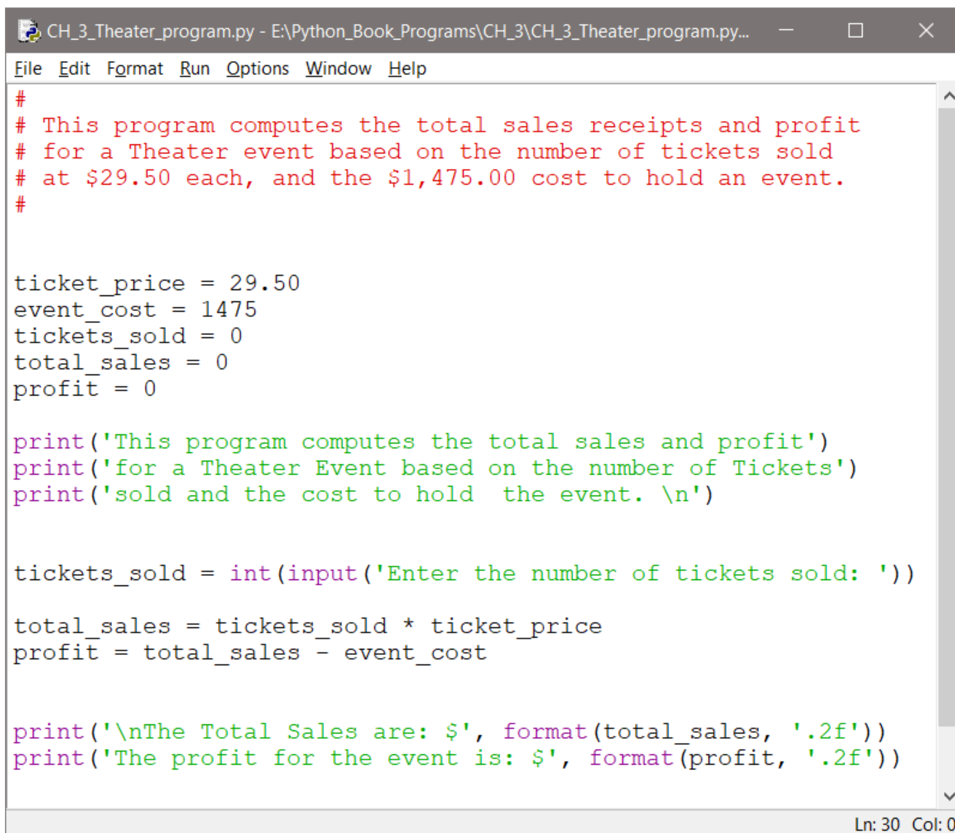


```
print('This program computes the total sales and profit')
print('for a Theater Event based on the number of Tickets')
print('sold and the cost to hold  the event. \n')


tickets_sold = int(input('Enter the number of tickets sold: '
```

4. The processing section computes the total sales from the tickets sold, and the profit once the total sales are computed.



5. Finally, the output section is completed with dollar signs and format specifiers for the dollar amounts (two decimal places). Note the blank lines between sections of code to enhance readability.



6. Testing the program includes initially using input data that is easy to verify. Since 100 * $29.50, and $2950 – $1475 are easily checked, testing begins with 100 tickets. Then other ticket amounts are tested.

```
This program computes the total sales and profit
for a Theater Event based on the number of Tickets
sold and the cost to hold  the event.

Enter the number of tickets sold: 100

The Total Sales are:  2950.00
The profit for the event is: 1475.00
>>>
```

Project Summary:

A step-by-step approach forms good program design and development habits that are critical in developing complex programs. As complexity increases, so does the chance that errors will be introduced. The goal is to minimize errors and debugging time, and deliver a computer-based solution that meets the requirements.

Step 1 Review the program requirements

Ensure an accurate understanding of the task

Step 2 Requirements Decomposition

Break down the task in sub-tasks

Use pseudocode and a flowchart to determine the solution

Step 3 Development – programming the solution

Step 4 Testing and debugging

Chapter 3 Review Questions

1. Words added to programs to explain complex areas or to add clarity and are not executed when the program runs are _______________.
2. The ________ function is used to display output in Python.
3. In Python, literal strings can be surrounded by single or double __________.
4. In order to execute a function, it must be _________.
5. A ___________ is an item passed to a function.
6. A __________ is a sequence of characters in code surrounded with quotes.
7. The equal sign is used to ______________ a value to a variable.
8. The ______ side of the assignment statement is assigned to the ______ side.
9. __________ are used to store values in memory.
10. A variable must be __________ before it can be used by the program.
11. Integer, float, and string are examples of data ________.
12. The ________ data type can be only true or false.
13. Variable names (can or cannot) _______ begin with a number.
14. _______________ should be used in place of magic numbers.
15. Floating point numbers in Python are stored with __________ precision.
16. When multiple arguments are passed to the print function, they are separated by _________ in the output by default.
17. To suppress or replace the default separator for multiple arguments when calling the print function, the __________ argument is used.
18. Concatenation refers to _____________ two or more strings.
19. The term immutable means that strings in Python cannot be ___________.
20. The ___________ function is used to set the number of decimal places in the output of a number.
21. To suppress the automatic line feed after a print function call, the _________ argument is passed.
22. The __________ escape character is used to produce a tab.
23. The _________ function is used to obtain input from the keyboard.
24. Converting an item to a different data type is known as ___________.

Chapter 3 Short Answer Exercises

1. What do the following lines of code output?

```
ounces_per_can = 6
print('Ounces: ', ounces_per_can)
```

2. What is the result when the following lines of code are executed?

```
number = 23
print('The number is ' + number)
```

3. What do the following lines of code output?

```
num = 123.4567
print('Number is ', format(num, '.2f'))
```

4. What do the following lines of code output?

```
num = 12
print('Number is ', format(num, '.2f'))
```

5. What do the following lines of code output?

```
num = 8.367
print('Number is ', format(num, '.1f'))
```

6. In following expression, what does the number "8" specify?

```
print(format(var1, '8.4f'))
```

7. In following expression, what does the letter "d" specify?

```
print(format(var1, ',d'))
```

8. Which of the following variable names follow proper naming conventions?

   a. average
   b. 8pieces
   c. netPay$
   d. gross pay
   e. hourly_rate

9. What type of variable is defined in this expression?

   ```
   INTEREST_RATE = 0.07
   ```

10. What is the output from the following statement?

    ```
    num = 850
    print(‘The number twice is ‘, num, num, sep=”)
    ```

11. What is the output from the following statement?

    ```
    var1 = ‘my’
    var2 = ‘dog is’
    var3 = ‘happy’
    print(var1, var2, var3)
    ```

12. What is the output from the following statement?

    ```
    print(format(24, ‘.3f’))
    ```

13. What is the output from the following statement?

    ```
    print(format(98765.378, ‘,.2f’))
    ```

14. What is the output from the following statements?

    ```
    print(‘Press the Enter key ‘, end=”)
    print(‘when ready.’)
    ```

15. What is the output from the following statements?

    ```
    print(‘She said \”Hello\”. ‘)
    ```

16. In each of the expressions below, what will be the value assigned to the variable result?

    a. result = 5 // 2
    b. result = 7 / 2
    c. result = 4 * 3 / 2
    d. result = 5 % 2
    e. result = 2**3

17. In each of the expressions below, what will be the value assigned to the variable balance?
    a. balance = round(5.4)
    b. balance = round(3.6)
    c. balance = round(6.767, 2)
    d. balance = round(-13.5)

18. In the code below, what data type will be stored in salary? Is it the correct data type for the request?

    salary = int(input('Enter your salary: ')

19. Express the following equations in Python code.
    a. 4xy
    b. z = 2ab
    c. $y = b^2 - 4ac$
    d. $t = \frac{a + b}{x - y}$

Chapter 3 Programming Exercises

1. Write a program that displays the following text.

   Python is an interpretive language.

2. Write a program that displays the following text with Python in quotes.

   The language is "Python".

3. Write a program that displays the following text with the apostrophe.

   That doesn't add up!

4. Write a program that prompts the user to enter their name and then displays 'Hello ', and the name that was entered.

5. Write a program with three variables: first, second, and third. Assign the values 5, 6, and 7 to the variables and display each on a separate line.

6. Write a program that uses the Named Constant below to display 'The interest rate is ', followed by the constant value.

   INTEREST_RATE = 7

7. Write a program that assigns a variable tickets the value 125 and then displays, 'The tickets sold today were ', followed by the variable.

8. Write a program that defines three variables named word1, word2, and word3. Assign 'abc' to word1, 'def' to word2, and then assign word1 and word2 combined to word3 using concatenation. Then display word3.

9. Write a program that uses a format specifier to display the number 12345.678 formatted with commas and two decimal places as shown.

   12,345.68

10. Write a program that displays the numbers below on separate lines, with two (2) decimal places, and in fields that are eight (8) characters wide.

    123.45   1452.56   56.80

11. Write a program that uses three (3) print statements to display the words No, lines, and between all on one line with spaces between the words.

12. Write a program that prompts the user to enter their age and then displays 'Your age is' and the age that was entered

13. Write a program that prompts the user to enter a number, and then a second number. The program will add the numbers, and display 'The sum of the numbers is: ', and the result.

14. Write a program that computes the total cost of a meal based on the meal price entered by the user, plus a 20% tip, and 5% sales tax. The output should be displayed as shown below and include a dollar sign and two (2) decimal places.

```
Enter the price of the meal: $27.56
Total price is $ 34.45
```

15. Expand number 14 to include output of the tip, and tax amounts, before the total price. The output should include a blank line between the input prompt and the output, and include dollar signs and two (2) decimal places for dollar amounts.

```
Enter the price of the meal: $34.98

The tip amount is $ 7.00
The tax amount is $ 1.75
The total price is $ 43.73
```

16. Write a program that prompts the user to enter a Fahrenheit temperature, computes the Celsius temperature and displays the 'The Celsius temperature is: ', and the result. The equation for the conversion is:

    C = (F – 32) / 1.8 Test data: When F = 23, C = -5

17. Write a program that prompts the user to enter the lengths of the two sides of a rectangle. The program will compute the area and perimeter, and assign the values to two variables. Then display the computed values with their titles as shown in the example below.

```
Enter side length one: 3
Enter side length two: 4
The area is:  12.0
The perimeter is:  14.0
```

18. Write a program for a Yogurt vendor that computes the total sales and profit for a day's sales based on the number sold at $6.50 each, and the cost of the Yogurt to the vendor which is $4.25 each. The profit is the total sales minus the total cost. The program will display the output as shown in the example below.

```
Enter the number sold: 10
Units sold:  10
Total sales: $ 65.00
Total cost: $ 42.50
-------------------------
Profit for the day: $ 22.50
```

19. Write a program that prompts the user for two integers (x and y) and computes a result using the equation below. Note the output when y is entered as 1.

$$answer = \frac{x+2}{y-1}$$

20. Part #1: The surface area of a sphere is given by the equation below. Write a program that prompts the user for the radius of a sphere as a float and the units of measure (feet, miles, etc.), computes the surface area, and displays the result with the square units. Use the operator for exponentiation in the solution, and format the output to 3 decimal places. Use 3.14159 as the value for PI.

$$\mathit{Surface\ area} = 4\ \pi\ r^2$$

Part #2: The volume of a sphere is given by the equation below. Write a program that prompts the user for the radius of a sphere as a float and the units of measure (feet, miles, etc.), computes the volume, and displays the result with the units. Use 3.14159 as the value for PI.

$$Volume = 4\pi \frac{r^3}{3}$$

Part #3: Combine parts 1 and 2 into a single program that prompts the user for a radius as a float and the units of measure, and in addition to the surface area and the volume of the sphere, display the circumference. Two examples are provided using 3.13159 as the value for PI, to validate the output.

Earth radius at the equator: 3,963.2 miles

```
Enter the radius of a sphere: 3963.2
Enter the units (feet, miles, etc.): miles

The circumference is 24,901.499 miles
The surface area is 197,379,241.483 square  miles
The volume is 260,751,136,615.626 cubic  miles
```

Volley Ball radius: approx. 4.1 inches

```
Enter the radius of a sphere: 4.1
Enter the units (feet, miles, etc.): inches

The circumference is 25.761 inches
The surface area is 211.241 square  inches
The volume is 288.695 cubic  inches
```

Chapter 3 Programming Challenge

Loan Calculator

Design and develop a program for a car dealer that computes the monthly payment, total payback amount, and total interest paid for a car loan based upon the loan amount, interest rate, and loan duration in months.

The equation for determining the monthly payment for a loan is:

Monthly Loan Payment Formula: $MP = L * (r / (1 - (1 + r)^{-N}))$.

- MP = monthly payment amount
- L = principal, meaning the amount of money borrowed
- r = effective interest rate. Note that this is usually not the annual interest rate (see below).
- N = total number of payments

Calculate the effective interest rate (r) - Most loan terms use the "nominal annual interest rate," but that is an annual rate. Divide the annual interest rate by 100 to put it in decimal form, and then divide it by the number of payments per year (12) to get the effective interest rate.

- Example, if the annual interest rate is 5%, and payments are made monthly (12 times per year), calculate 5/100 to get 0.05, then calculate the rate:

  Effective rate = 0.05 / 12 = **0.004167**.

Sample output:

```
Enter the loan amount: 10000
Enter the interest rate: 4.5
Enter the duration in months: 48

The monthly payment is $ 228.03
The payback amount is $ 10945.67
The total interest is $ 945.67
```

Chapter 4

Decision Structures and Boolean Logic

Decision or control structures determine the statements that execute based upon a condition that is either true or false. A conditional statement is used to determine whether or not a line or lines of code execute. Conditional statements provide multiple paths through a program based on the status of a ***Boolean*** (true or false) condition. If the condition is true, then a statement or statements are executed, otherwise they are not executed. The decision structure is implemented using the ***if*** statement.

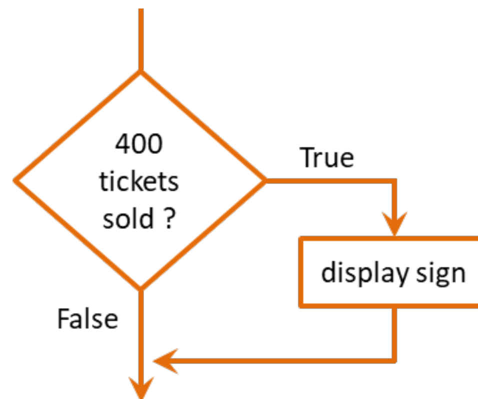
As an example, assume that a Theater has seating for 400 participants. Once the Theater has sold 400 tickets, the show has been sold out. When this occurs, the Theater displays a "Sold Out" sign at the box office. The ***decision*** to display the sign is made based upon whether or not 400 tickets have been sold.

Ex. 4.1 – executing a statement based upon a condition

If 400 tickets have been sold
- Display the "Sold Out" sign

The condition is tested (have 400 tickets been sold), and if it is true, then the "Sold Out" sign is displayed. If the condition is false and 400 tickets have not been sold, then the sign will not be displayed.

Conditional expressions are represented in flowcharts as diamonds. The different paths that the program can take are shown using lines from the corners of the diamond with arrows indicating the direction with text to indicate the result. These paths in the program are often referred to as the ***Flow of Control*** or the ***Order of Operations***. If the condition is true, then the flow of control follows the path to display the sign. Otherwise (if it is false), the program continues.



Decision Structure Flowchart

The if Statement

When writing a conditional statement in Python, the condition begins with the word *if*, followed by the condition, and ends with a colon. The statement that is to be executed based upon the condition is below the condition and indented (one tab space). This is often referred to as the "if clause". The auto-indent feature in the IDE will automatically indent the next line after the semicolon when *Enter* is pressed. The interpreter associates the indented statement with the condition. If the condition is true, the statement will be executed. If the condition is false, then the statement will be skipped. The general format is:

```
if conditon:
    statement
```

Multiple statements can be associated with a condition and form what is commonly referred to as a ***block of code***. A block of code is a group of associated statements. In this case, they are associated with the condition. If the

condition is true, all of the indented statements (the block of code) will be executed. If the condition is false, all of the indented statements will be skipped by the interpreter. The general format is:

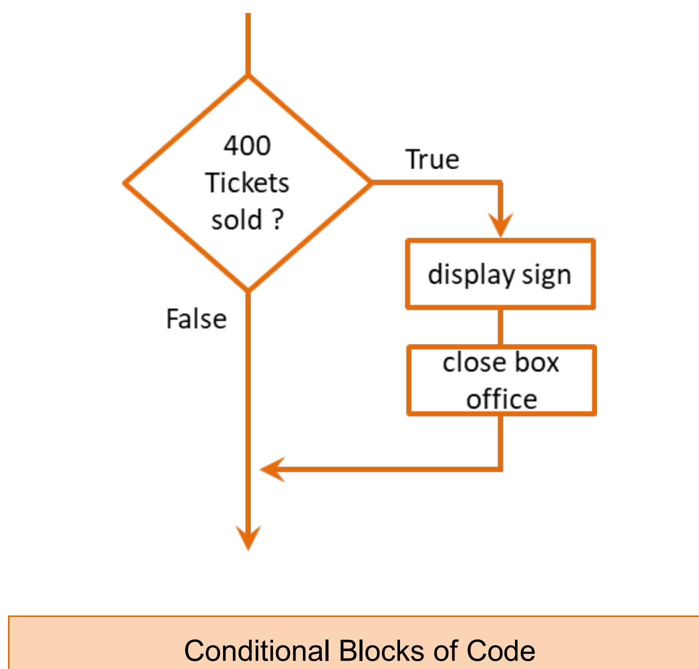
```
if conditon:
    statement1
    statement2
    statement3
    etc.
```

Ex. 4.2 – conditional statements and blocks of code

Continuing the Theater example, assume that when the show is sold out, in addition to the sold out sign being displayed, the box office is closed. A standard practice is to indent in the pseudocode since the lines would be indented in the actual code.

If 400 tickets have been sold
- Display the “Sold Out” sign
- Close the box office

The flowchart for the Theater example has been modified to include closing the box office if the condition is true.



Conditional Blocks of Code

Boolean Expressions

Since conditional statements are either true or false, they are referred to as ***Boolean Expressions*** named after the mathematician George Boole (1815-1864). Boolean expressions are implemented using ***Relational Operators*** that resolve to either true or false. For example, one value can be greater than another, or less than another, or equal to another. One of these three choices must be true. Table 4.1 lists the Boolean operators available in Python.

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
= =	Equal to
! =	Not equal to

Table 4.1 – Relational Operators

Relational operators in conditional statements are used to test the relationship between items. The result of the expression is used to determine the next step for the program. Relational expressions are used extensively in programming.

For the examples below: x = 5, y = 8, z = 5

x > y	5 > 8	False	5 is not greater than 8
x < y	5 < 8	True	5 is less than 8
x >= z	5 >= 5	True	5 is equivalent to 5
x <= z	5 <= 5	True	5 is equivalent to 5
x == y	5 == 8	False	5 is not equivalent to 8
x != y	5 != 8	True	5 is not equivalent to 8
x == z	5 == 5	True	5 is equivalent to 5
x != z	5 != 5	False	5 is equivalent to 5

Relational Expressions

Ex. 4.3 – conditional statements and relational operators

The Theater example conditionally displayed a "Sold Out" sign and closed the box office if 400 tickets had been sold. A Boolean expression with a relational operator would be used in the conditional statement for the code. The number of tickets sold is 400 and the expression is true, or 400 tickets have not been sold and the expression is false.

```
tickets_sold = int(input('Enter the tickets sold: '))

if tickets_sold == 400:
    print('Display the "Sold Out" sign')
    print('Close the box office')
```

The code above obtains the number of tickets sold from the input, and tests for exactly 400 tickets. There could not be more than 400 tickets sold since the box office closes, therefore there are either 400 or less tickets sold.

The if-else Statement

In Example 4.3 the print statements execute when there are exactly 400 tickets sold. Otherwise the program does nothing. To provide for other statements to execute when the condition is false, an ***else*** clause is implemented using the word "else" followed by a colon. An else clause can be thought of as an "*otherwise*" condition for when the relational expression is not true.

When the "*if*" condition is true, the statements in the "*if*" block will be executed and the "*else*" block will be skipped. When the "*if*" condition is false, the "*if*" block will be skipped and the "*else*" block will execute. The general format is:

```
if conditon:
    statement1
    statement2
    etc.

else:
    statement1
    statement2
    etc.
```

Ex. 4.4 – the if-else statement

Continuing the Theater example, if 400 tickets have been sold, the "*if*" block will execute, otherwise the else block will execute. The pseudocode is shown here.

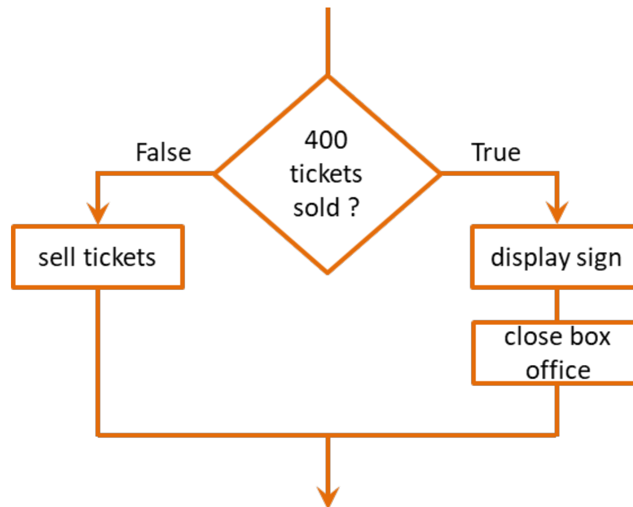
If 400 tickets have been sold:

- Display the "Sold Out" sign
- Close the box office

else:

- Continue to sell tickets

A flowchart highlights the two different paths that can be taken as a result of the conditional expression, and that only one path will be executed. It also shows that the two paths converge afterward and the program continues.



The code below highlights a few points. Indentation is required to form a block of code, and the "if" and "else" are aligned.

```
tickets_sold = int(input('Enter the tickets sold: '))

if tickets_sold == 400:
    print('Display the "Sold Out" sign')
    print('Close the box office')


else:
    print('Keep selling tickets')
```

If-else Condition

Nested if Structures

When two (or more) conditions are being tested, there are several ways to implement the logic. One of these is to use a nested if, which is an "*if*" condition nested inside another "*if*" condition. The implementation is similar to an "*if*", and indentation is again critical. If *condition1* below is true, then *condition2* is tested, and if it is also true, then the statements will be executed. If *condition1* is false, then *condition2* will not be tested and the statements are skipped.

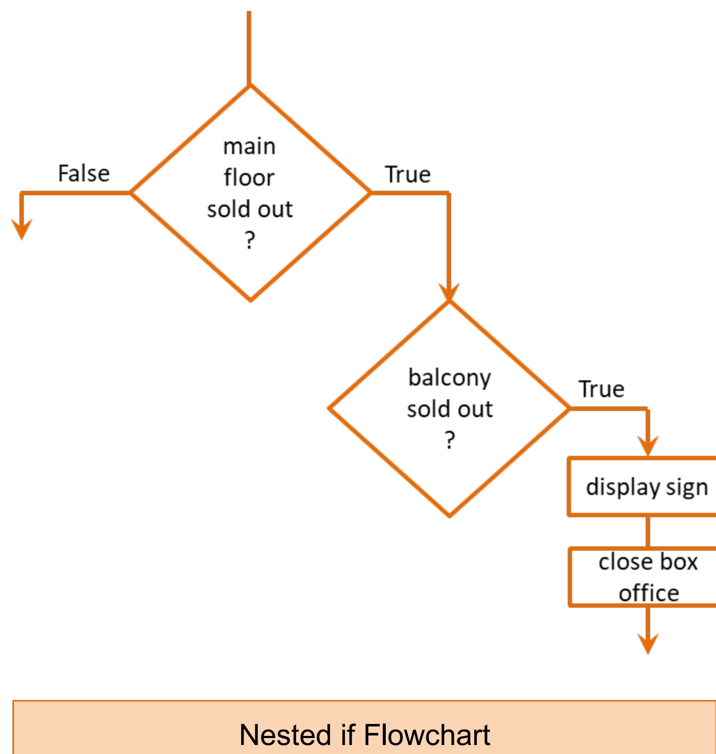
```
if conditon1:
    if condition2:
        statement1
        statement2
        etc.
```

To illustrate this, the Theater example now includes a balcony section with 200 seats. The tickets sales are tracked separately, so to determine if the show is sold out requires testing both seating areas.

If the 400 main floor seats are sold

If the 200 balcony seats are sold

- Display the "Sold Out" sign
- Close the box office



Nested if Flowchart

The conditional expression for the balcony seats above is only tested if the main floor is sold out. Later, Boolean Logic will be covered which will combine expressions and in most cases eliminate the need for a nested if.

The if-elif-else Structure

To handle situations where multiple conditions result in different paths, an "if" and an "else" are inadequate because there are only two paths. Additional "if" statements may be appropriate, but more often the ***if-elif-else*** structure is a better solution. The "***elif***" statement can be thought of as a short version of "else-if".

Ex. 4.5 – the if-elif-else structure

The pseudocode below tests for a proper discount percentage based on the price of an item. The logic only tests the second condition if the first condition is false, and the third condition is only tested if the first and second conditions are false. The final *else* handles the situation when all of the conditions before it are false.

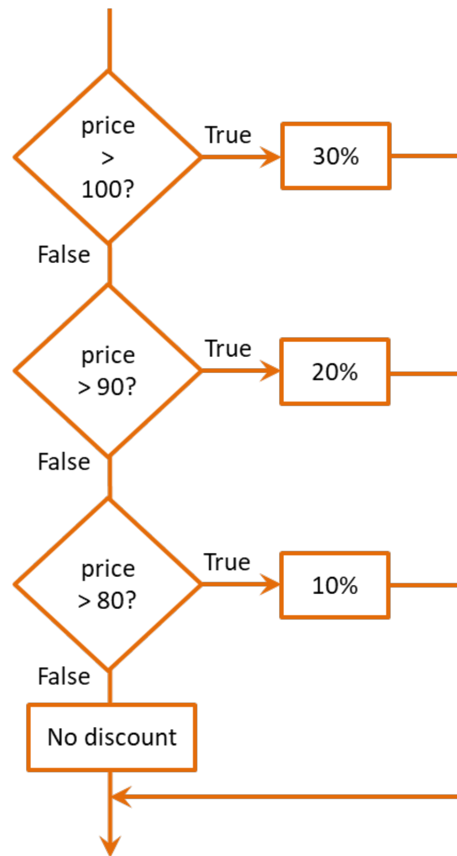
Consider that a price of $120.00 is also greater than $100.00 and greater than $90.00 and greater than $80.00, but after the first condition resolves to true, Discount is 30%, and the other conditions are skipped over. If the price were $95.00, then the first condition would be false and the second condition would be tested. Since it would then be true, the discount would be 20%, and the other conditions after it would be skipped.

Pseudocode

```
If the price > $100.00
    Discount is "30%"
Otherwise-if the price > $90.00
    Discount is "20%"
Otherwise-if the price is > $80.00
    Discount is "10%"
Otherwise
    No discount
```

If-elif-else Pseudocode

The following flowchart highlights the paths based upon the price of the item, and that only one path is taken as a result of the conditions. If all of the conditions are false, then there is no discount which would be the else condition.



If-elif-else Flowchart

The code for the *if-elif-else* structure example is shown below.

```
if price > 100:
    discount = '30%'

elif price > 90:
    discount = '20%'

elif price > 80:
    discount = '10%'

else:
    discount = 'No discount'
```

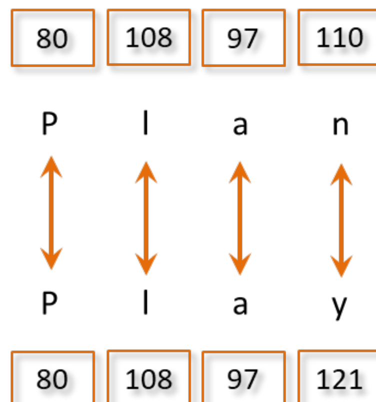
If-elif-else

Comparing Strings

In addition to comparing numbers, very often strings need to be compared. When a password is changed it is typically verified and is entered twice. The two are compared to ensure that they match. In computing, what is actually being compared is the ASCII representation of the letters character by character. Recall from Chapter 1 that each character has a binary representation in the ASCII character set (Appendix A). To compare strings to see if they match, the equivalence operator (two equal signs) is used.

```
word1 = 'Play'
word2 = 'Plan'

if word1 == word2:
    print('The words match')
else:
    print('They don't match')
```

In the conditional statement, each character of each string is compared one at a time using the ASCII representation for the letter. When 'n and 'y' are compared the condition is false. The two strings are not equivalent. The base-10 ASCII values for the letters are 110 for 'n' and 121 for 'y'. Therefore, Play is greater than Plan since 'y' has a higher ASCII value than 'n'.



String Comparison

When a string has more characters than another, they would not be equivalent and the longer string would be greater.

Compound Boolean Expressions (and, or, not)

When a program needs to make decisions based on complex conditions, multiple conditions can be combined using the Logical Operators ***and, or***, and ***not***. In an earlier example, a nested-if was used to test two conditions. The second condition was tested only if the first condition was true. Both conditions need to be true for the statements to execute. The pseudocode would be *"if condition1 is true AND then if condition2 is true, execute the statements"*.

```
if conditon1:
    if condition2:
        statement1
        statement2
        etc.
```

Both of the conditions above can be combined into a single compound expression using the logical "***and***" operator. For the expression to be true, both sides of the expression must be true. If either of them is false, then the expression is false. The general format is:

```
if condition1 and condition2:
    statement1
    statement2
    etc.
```

Combined Expressions with "and"

As shown in Table 4.2 below, the combined expression is only true when both conditions are true.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Table 4.2 - Logical "and" Truth Table

This logic is often used to verify that a number is within a range, especially when validating input. The expression "Garbage in, garbage out" or GIGO is common in programming. The program should validate any input before continuing.

Ex. 4.6 – input validation with "and"

Assume that a program requires the user to enter a number between 1 and 100. A logical **and** can be used to validate the input for the required range. Both conditions must be true, for the expression to be true. The number entered must be greater than 0 and less than or equivalent to 100.

```
num = int(input('Enter a number between 1 and 100: '))

if num > 0 and num <= 100:
    print('That is a good number')
```

Input Validation with "and"

Ex. 4.7 – the Theater example revisited using "and"

The Theater ticket sales scenario from Chapter 3 included a balcony section in addition to the main-floor seats that needed to be sold out for the Theater to display the "Sold Out" sign and close the box office. Both conditions must be true and a nested "*if*" condition was used in the example.

If the 400 main floor seats are sold

If the 200 balcony seats are sold

- Display the "Sold Out" sign
- Close the box office

This can be easily implemented with an "*and*" operator, since both conditions must be true.

```
if main_tickets_sold == 400 and balcony_tickets_sold == 200:
    print('Display the "Sold Out" sign')
    print('Close the box office')
```

Logical "and"

Another way of testing multiple conditions is to reverse the logic. For example, if the main_tickets_sold is less than 400 or balcony_tickets_sold is less than 200 then there are more tickets to sell. The logical ***or*** is used to test an either-or situation. The general format is:

```
if condition1 or condition2:
    statement1
    statement2
    etc.
```

As shown in Table 4.3, when either condition is true, the combined expression is true.

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Table 4.3 - Logical "or" Truth Table

Ex. 4.8 – input validation with "or"

Using the previous example of validating a number between 1 and 100, a logical ***or*** can be used to test both conditions as well. Note the different values used to test outside the range instead of inside, and the change to the print statement. Again, the number must be greater than 0 and less than 101, but the test is reversed. If either condition is true, then the expression is true, and the number is not within the range required.

```
num = int(input('Enter a number between 1 and 100: '))

if num < 1 or num > 100:
    print('That is NOT a good number')
```

Input Validation with "or"

Short-Circuit Evaluation

The logical "*and*" and the logical "*or*" operators both use what is called short-circuit evaluation. With the logical "and", both sides of the compound condition must be true for the expression to be true, so if the left side is false, then the right side is not evaluated. It wouldn't matter if the right side were true since the expression is already false.

The reverse occurs with the logical "or". When either side of a compound expression using "*or*" is true, the expression is true. Therefore, if the left side of the compound condition is true, the right side is not evaluated. It doesn't matter whether the right side is true or false since the expression is already true.

Some Common Logic Errors

Recall that logic errors are those errors that do not halt execution of the program but produce incorrect results. Many of these occur due to confusion associated with logical "*and*" and "*or*" expressions. The wording associated with conditions in requirements can be vague and result in poor translation to code. Pseudocode and flowcharts can help, but careful consideration of the logic is required. Table 4.4 lists some example expressions and the numeric values that would make the expressions true. Notice that there are two situations were any number is valid.

Expression	True when the value of x is:
if x > 0 and x < 100	any number 1 thru 99
if x >= 0 and x <= 100	any number 0 thru 100
if x > 0 or x < 100	any number
if x < 0 or x > 100	negative or above 100
if x <= 0 or x >= 100	negative, zero, or 100 or above

Table 4.4 - Logical Expressions

The last of the logical operators is the ***not*** operator. This operator returns the reverse value of the logical value of a Boolean operand. If the operand is true,

the "*not*" operator returns false. If the operand is false, the "*not*" operator returns true.

A	not A
True	False
False	True

Table 4.5 - Logical "not" Truth Table

Caution should be exercised when using the "*not*" operator because it often introduces bugs and confusion. In pseudocode, the expression below would read "If x is greater than y is true and x is greater than z is true, then return false". It isn't clear what condition is being tested.

```
if not (x > y and x > z):
```

It is often easier to reverse the logic and remove the "*not*" operator. De Morgan's Law provides two forms: one for negation of an "and" expression and one for an "or" expression.

```
! (A and B)        !A or !B
! (A or B)         !A and ! B
```

A logical and methodical approach when creating conditional statements and compound expressions will save a lot of time debugging logic errors.

Boolean Variables

Boolean variables are available in Python as the ***bool*** data type which operates as true or false. They are often used as flags or signals in code when something has occurred or a condition has been met, and can be used in conditional statements. The example below uses the Theater example and sets a Boolean variable to true when enough tickets have been sold. The flag is used to determine if the print statements should execute.

Ex. 4.9 – the Theater revisited using the bool data type

A bool variable sold_out is assigned False, and the condition is tested. If the conditional expression is true, then it is assigned True. It can then be used as a flag in the second "*if*" conditional expression.

```
sold_out = False

if main_sold == 400 and balcony_sold == 200:
    sold_out = True


if sold_out == True:
    print('Display the "Sold Out" sign')
    print('Close the box office')
```

Notice that the second "if" expression above can be written "if sold_out:". Since the variable is Boolean, it is interpreted correctly. The expression also highlights an occasion to use the *not* operator as shown below. The logic is reversed and the expression is clear.

```
if not sold_out:
    print('Keep selling tickets')
```

Common Errors

Some of the most common errors in programming have to do with logical operators and writing conditional expressions. Most are easily found when testing around the conditional value or values being tested.

```
if value < 90          # excludes 90
if value <= 90         # includes 90
```

Other common errors include confusing when to use "*and*", when to use "*or*", and the "*not*" operator. A careful review of the requirements and using tools like pseudocode or a flowchart will save time debugging and testing to find and correct errors.

A surprisingly common error is using a single equal sign when two are required.

```
value = num            # assigns num to value
value == num           # tests for equivalence
```

Since Python relies on indentation to associate lines of code with conditions, forgetting to indent or out-dent will cause issues as well. Note the output statements associated with the conditional expressions below.

```
if value < 90:
    print('I am associated with the IF clause.')
print('I am  not associated with the IF clause.')


if value < 0 and value > 10:
    print('This will never be true.')


if value > 0 or value < 10:
    print('This will always be true.')


if value > 0 and value > 10 or value < 20:
    print('Ambiguous, and always true.')
```

The interpreter will help with errors in syntax and grammar, but logic errors can only be avoided by careful implementation of the code. The best solution may be more deliberate and intentional than the use of complex statements. It is also a fact that code is written once, but read many times, so readability is always a consideration.

Consider that all three of the following solutions accomplish the same thing. The choice of which one to use is up to the programmer.

```
if number > 0:
    if number < 101:
        print('Number is valid.')


if number > 0 and number <= 100:
    print('Number is valid.')


if number <= 0 or number > 100:
    print('Number is not valid.')
else:
    print('Number is valid.')
```

Chapter 4 Review Questions

1. Decision structures determine the statements that execute based upon a _______________.
2. A ____________ result is one that is either true or false.
3. The Flow of Control refers to the _________ in which statements will execute.
4. In a Flowchart, decisions are represented by _____________.
5. Statements that execute when an "*if*" condition is true are below the condition and ___________.
6. When multiple statements are associated with an "if" condition, they are referred to as a ___________ of code.
7. Boolean expressions are implemented using _______________ operators.
8. In an *if-else* statement, when the "*if*" condition is false the _________ clause will execute.
9. An "*if*" conditional statement that directly follows another "*if*" conditional statement and is indented is referred to as a ____________ if.
10. When two strings are compared for equivalence, the _____________ value of each character is compared individually.
11. Compound Boolean expressions are implemented using the ________ operators.
12. For a compound expression that uses a logical "*and*" to be true, _________ of the conditions must be true.
13. For a compound expression that uses a logical "*or*" to be true, _________ of the conditions must be true.
14. The logical operator that is used to negate a Boolean value is the ____ operator.
15. Boolean variables can only be assigned a status of ______ or ______.
16. Boolean variables are often used as __________ in code.

Chapter 4 Short Answer Exercises

1. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if var2 > var1:
        print('var2 is greater')
```

2. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if var1 >= var2:
        print('var2 is greater')
```

3. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if var1 <= var2:
        var3 = var1 + var2
        print('var3 is', var3)
```

4. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if var1 == var2:
        print('They are the same')
print('They are not the same')
```

5. What do the following lines of code output if *first* = 10, and *second* = 10?

```
if first == second:
        print('They are the same')
print('They are not the same')
```

6. What do the following lines of code output if *first* = 10, and *second* = 10?

```
if first > second:
        print('first is greater')
else:
        print('second is greater')
```

7. What do the following lines of code output if *val1* = 3, *val2* = 5, and *val3* = 8

```
if val2 > val1:
        val3 = val2 – val1
elif val3 > val2:
        val3 = val2
else:
        val3 = 99
print('val3 is', val3)
```

8. Write the word Python using the Base-10 digit equivalences for the letters (ref. Appendix A).

9. Are the following string relationships True or False?

   a. Many > Mare
   b. Tent > Tens
   c. dress == Dress
   d. Cod != cod
   e. python >= python

10. Are following expressions true or false if *first* = true and *second* = false?

    a. first and second
    b. first or second
    c. second or first
    d. second and first
    e. !first
    f. !second

11. Write an "*if*" conditional expression using a logical operator that tests for a number that is greater than 32 and less than 120.

12. Write an "*if*" conditional expression using a logical operator that tests for a number between 0 and 50, including 50 but excluding 0.

13. True or false, the following expressions test for the same condition.

    if num > 9 and num < 21   if num >= 10 and num <= 20

14. What range of numbers assigned to num would make this expression true?

    if num > 0 or num < 100

15. What range of numbers assigned to num would make the following expression true?

    if num > 0 and num > 100

16. What numbers are excluded by the following expression?

    if num < 0 or num > 0

17. What value assigned to done would execute the print statement?

```
if done:
        print('That's all')
```

Chapter 4 Programming Exercises

1. Write a program that accepts an integer as input and displays whether or not it is greater than zero.

2. Write a program that accepts an integer as input and displays whether it is positive, negative, or zero.

3. Write a program that accepts an integer as input for the number of hours worked and executes the following algorithm. If the number of hours worked are greater than 40, then output "There is overtime", otherwise output "There is no overtime".

4. Expansion of the program in #3. Draw a flowchart of the algorithm and write a program that implements the following pseudocode. Consider each variable that is needed, the order of operations, and formatting of the output for dollar amounts. Design the solution in terms of input, processing, and then output.

```
Get the number of hours worked
Get the hourly rate of pay
Compute regular pay (hourly rate * hours up to and including 40)
if the number of hours worked > 40
        – compute overtime pay (1.5 * hourly rate for hours > 40)
Output regular pay
Output overtime pay
Output total pay
```

<u>Sample output</u>

```
Enter hours worked 41
Enter hourly rate $10.00

Regular pay is $ 400.00
Overtime pay is $ 15.00
Total pay is $ 415.00
```

5. Modify program #4 to include double time pay (2 * hourly rate) for hours above 50, and add the additional output for the double time pay. The hours from 41 to 50 remain time-and-a-half pay (1.5 * hourly rate). Sample output is shown below.

```
Enter hours worked 51
Enter hourly rate $10.00

Regular pay is $ 400.00
Overtime 1.5x pay is $ 150.00
Overtime at 2x pay is $ 20.00
Total pay is $ 570.00
```

6. Write a program that requests two words from the user and sorts them alphabetically based on relational operators. Display the words in the correct order, or if the same word is entered twice, output "They are the same".

7. Write a program that requests a *username* and *password* from the user, and then requests that they confirm the password. If the passwords match, output "account has been created for" and the *username*, otherwise output "Invalid password confirmation".

8. Write a store program that accepts the price for an item and computes a discounted price based on the criteria below, determines the 7% sales tax amount on the discounted price, and display the original price, discounted price, sales tax, and the total amount for the purchase. Include dollar signs, two decimal places, and right align the dollar amounts as shown in the sample output below.

Discount criteria

Greater than $100.00 – 27%

Greater than $90.00 – 22%

Greater than $80.00 – 16%

Greater than $60.00 – 8%

Sample output

```
Enter the price of the item 102.56

Original price $ 102.56
Discount price $  74.87
Tax            $   5.24
Total amount   $  80.11
```

9. Write a program that prompts the user to enter a temperature and then “F or f” or “C or c” if it is a Fahrenheit or Celsius temperature to convert. Display both of the temperatures or “Cannot convert” if an incorrect letter was entered. The equations for the conversions are:

$$C = (F - 32) / 1.8 \qquad F = (C * 1.8) + 32$$

10. Write a program for a Theater that computes the total sales receipts and profit for an event based on the number of tickets sold and the following criteria:

    - The 200 main floor tickets are sold first, and then the 75 balcony tickets are sold once the main floor is sold out.
    - Main floor tickets are $29.50 each, and Balcony tickets are $19.50 each.
    - The program will request the total number of tickets sold and “M” for Matinee or “E” for evening. The cost to hold an event is $1,200.00 for Matinee and $1,450.00 for evening.
    - The output will include the number of tickets sold and sales for each section, the total sales receipts, the event cost, and the profit for the event. Profit is total sales minus cost.

    #10 (a) Write the pseudocode for the program

    #10 (b) Draw a flowchart of the solution

    #10 (c) Develop the program

<u>Sample output A</u>

```
Enter the number of tickets sold 199
Enter "M: for Matinee or "E" for Evening e

Main Floor tickets sold 199 - sales: $ 5870.50
Balcony tickets sold 0 - sales: $ 0.00
Total sales : $ 5870.50
Event cost: $ 1450.00
Profit from the event: $ 4420.50
```

<u>Sample output B</u>

```
Enter the number of tickets sold 201
Enter "M: for Matinee or "E" for Evening M

Main Floor tickets sold 200 - sales: $ 5900.00
Balcony tickets sold 1 - sales: $ 19.50
Total sales : $ 5919.50
Event cost: $ 1200.00
Profit from the event: $ 4719.50
```

11. The wind chill in North America is computed using temperature in Fahrenheit and wind speed in miles-per-hour, however it is not valid for temperatures above 50 degrees or when the wind speed is 3.0 mph or less. Write a program that requests the temperature and wind speed, and computes the wind chill or displays that it is not valid and the particular reason that it is not valid.

    The equation for approximating the wind chill factor in North America is:

    $$wc = 35.74 + 0.6215\, T_a - 35.75 V^{+0.16} + 0.4275\, T_a\, V^{+0.16}$$

    T_a is the air temperature in Fahrenheit, and

    V is the wind speed in mph (*consider* windSpeed ** 0.16)

Chapter 4 Programming Challenge

Planet-days to Earth-days

Write a program that compares Planet-days to Earth-days.

- Request the name of the planet and a number of Earth-days
- Validate that the input is one of the planets listed below
- Validate that the number of days is greater than zero
- Compute and display the number of planet-days that would pass on the chosen planet based upon the conversion values in hours provided (NASA).
- Display the result formatted to three (3) decimal places.

Planet	Earth Equivalent Hours
Earth	24.0 hours
Mercury	4222.6 hours
Venus	2802.0 hours
Mars	24.7 hours

Sample output A

```
Enter the name of the planet
Mercury, Venus, or Mars Mars
Enter the number of days to compare 5

5.0 Earth-days is equivalent to 4.858 days on Mars
```

Sample output B

```
Enter the name of the planet
Mercury, Venus, or Mars Venus
Enter the number of days to compare 5

5.0 Earth-days is equivalent to 0.043 days on Venus
```

Chapter 5

Repetition Structures - Loops

In a computer program, very often a statement or set of statements need to repeat over and over to accomplish a task or compute a result. An example would be computing compound interest for a bank account over *some* period of time. The program would begin with a balance, compute the interest amount, add it to the balance, compute the interest on the new balance, add it to the balance, and so on. This would continue for as many times as needed.

Start with an account balance
Compute the interest amount
Add it to the balance
Compute the interest on the new balance
Add it to the new balance
Compute the interest on the new balance
And so on…

In addition, it may be desirable to repeat an entire program instead of restarting the program to enter different inputs. ***Repetition Structures*** or ***loops*** provide a way of repeating steps without repeating code. The loop statements continue to execute until a final result has been reached and the loop ends. As an example, the Theater program in Chapter 4 contained a conditional statement for closing the box office if enough tickets had been sold. The alternate path was to

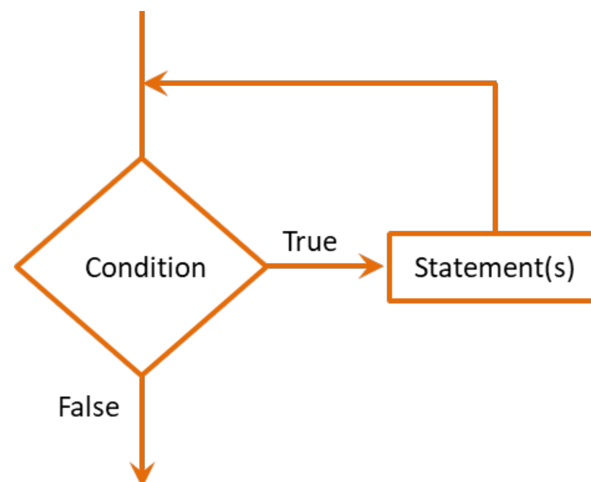
display "Sell Tickets" and end the program. Consider that while there are tickets to sell, they should continue to be sold and that each time tickets are sold, the condition should be tested again until all of the tickets are sold.

The While Loop

The while loop is a ***condition controlled*** loop. The statement or statements within the loop execute while some conditional expression is true. Each execution of a loop is referred to as an ***iteration*** of the loop. The loop structure is similar to conditional structures previously covered with a colon after the condition and indentation for the statements associated with the loop (the body of the loop). The general format is:

```
while condition:
    statment1
    statment2
    etc.
```

In flowcharts, the conditional expression is represented by a diamond, and the arrowed lines representing the order of operations indicate returning to test the conditional expression again when the condition is true. When the condition is false, the loop ends and execution continues.



Since the conditional expression in a while loop is tested prior to the body of the loop executing, it is referred to as a ***pre-test loop***. This means that a while loop may or may not execute depending on the result of the conditional expression.

Ex. 5.1 – Theater ticket sales enhanced with a while loop

A while loop could be used to enhance the Theater program so that it does not end until the Theater is sold out. The pseudocode and code follow.

```
While there are Theater tickets to sell
    Sell another ticket
    Increase the number of tickets sold

Display the sign and close the box office
```

Notice in the actual code below that the final two print statements are out-dented and are not part of the loop. They execute when the conditional expression is false. Also note that tickets_sold increases inside the loop to eventually make the conditional statement false. This is an important point. There must be a change that occurs inside the loop that eventually makes the condition false to end the loop. Otherwise, the loop will continue to run resulting in an ***infinite loop***. When this occurs, the program must be ended to stop the loop.

```
while tickets_sold < 400:
    print('Sell another ticket.')
    tickets_sold = tickets_sold + 1


print('Display "Sold Out" sign')
print('Close the box office')
```

While Loop

Infinite loops occur due to programmer errors. As an example, the following loop is an infinite loop because the variable value never changes (it will always be less than ten) and therefore the conditional expression is always true.

```
value = 3

while value < 10:
    print('Stuck in a loop')
```

Infinite Loop

A while loop can be used to allow the body of an entire program to run multiple times without the user having to restart the program. In this example, the user is prompted for whether or not another temperature conversion is desired. Notice that the variable another is set to '*y*' to start the loop. Since a while loop is a pre-test loop, the condition must be true or the loop will not be entered. The last statement within the loop asks the user if they would like to convert another. Any character entered other than 'y' will end the loop.

Ex. 5.2 – Temperature Conversion – condition-controlled loop

```
another = 'y'

while another == 'y':
    tempF = float(input('\nEnter a Fahrenheit temperature '))
    C = (tempF - 32) * 1.8
    print(tempF, 'Fahrenheit is ', C, 'Celsius\n')
    another = input('Convert another? (enter y for yes) ')


print('Have a nice day.')
```

```
Enter a Fahrenheit temperature 54
54.0 Fahrenheit is  39.6 Celsius

Convert another? (enter y for yes) y

Enter a Fahrenheit temperature 36
36.0 Fahrenheit is  7.2 Celsius

Convert another? (enter y for yes) n
Have a nice day.
```

Condition-controlled Loop

The For Loop

Another type of loop used in programming is the ***count-controlled loop*** where the number of iterations is a specific number of times. The programmer sets the number of times that the loop will execute when designing the loop.

The count-controlled repetition structure in Python is a ***for loop***. It is designed to be used with a sequence or a range of items. One implementation provides a sequence of items in square brackets for use by the loop statements. For each of

the items in the sequence, the loop body will execute. This is the reason that this loop is sometimes referred to as the "for-each" loop. The general format is:

```
for variable in [item1, item2, item3, etc.]:
    statement1
    statement2
    etc.
```

The keyword "*for*" is followed by a variable that will be used to store a copy of the items that are in the sequence one at a time. The sequence of items follow the word "*in*" and are enclosed in square brackets and separated by commas. The expression ends with a colon, and the statements to be executed in the loop body are indented.

One at a time, a copy of each value in the brackets will be placed in the variable and used in the loop statement or statements. In this example, the names are copied into the variable person and the print function is executed three times using each one.

```
for person in ['Abe', 'Beth', 'Jermain']:
    print('Hi', person)
```

```
Hi Abe
Hi Beth
Hi Jermain
```

The next example highlights that a string is a sequence of characters. Each character in the string "*Word*" is placed in the variable letter one at a time and is then passed to the *print* function. Since the print function adds a line feed, the letters are output vertically.

Ex. 5.3 – FOR loop accessing characters in a string

```
for letter in 'Word':
    print(letter)
```

```
W
o
r
d
```

The Range Function and For Loop

The second implementation of the for loop uses the Python ***range()*** function to simplify writing count-controlled loops. The function can accept one, two, or three arguments. When one argument is passed to the function, the range begins at zero and the argument that was passed is used as an ending limit for the range and is not included in the range. In the example, the print statement includes end=' ' to replace the line feed with a space.

Ex. 5.4 – For loop using the range function and a single argument

```
for value in range(10):
    print(value, end=' ')
```

When the code executes, the output begins at 0 and ends at 9 as shown below. Note that the argument passed to *range* is the limit 10 which is not included.

```
0 1 2 3 4 5 6 7 8 9
```

Ex. 5.5 – For loop using the range function and two arguments

When two arguments are passed to *range*, they are used as the starting and ending limits of the series. Again, note that twenty is the limit and is not included in the output.

```
for value in range(10, 20):
    print(value, end=' ')
```

```
10 11 12 13 14 15 16 17 18 19
```

When three arguments are passed to the range function, the third argument is used as the step value for the series. Lines feeds have been replaced by a space.

Ex. 5.6 – For loop using the range function and three arguments

```
for value in range(10, 35, 5):
    print(value, end=' ')
```

```
10 15 20 25 30
```

Each of the integer literals in the *range()* function examples can be replaced with variables. In the next example, all three literals are replaced with variables.

Ex. 5.7 – range function using variables instead of integer literals.

```
start = 24
limit = 49
step = 6
for value in range(start, limit, step):
    print (value, end=' ')
```

```
24 30 36 42 48
```

User Loop Control

For flexibility, a program often allows the user to determine the number of times a loop should iterate. As an example, the temperature conversion program below could convert a range of temperatures entered by the user. The arguments passed to the *range()* function must be integer values.

Ex. 5.8 – Temperature Conversion - user loop control

```
start = int(input('Enter a starting temperature '))
end = int(input('Enter an ending temperature '))

print()
print(' Fahrenheit    Celsius')

for tempF in range(start, end):
    C = (tempF - 32) * 1.8
    print('    ', tempF, '        ', C,)

print()
print('Have a nice day.')
```

```
Enter a starting temperature 27
Enter an ending temperature 32

 Fahrenheit    Celsius
     27          -9.0
     28          -7.2
     29          -5.4
     30          -3.6
     31          -1.8

Have a nice day.
```

Loop Accumulator

When a program needs to compute a running total or accumulate values, it uses what is referred to as an ***accumulator***. The accumulator is a variable that tallies values as a loop iterates and contains the total when the loop finishes. As an example, the program below asks the user how many grades will be entered and totals them as they are input. The variable total is initialized to zero and is used to accumulate the grades each time the loop executes. The print statement displays the average of the grades. Notice that counter is only used to control the loop.

Ex. 5.9 – grade averaging using an accumulator

```
total = 0.0

num_grades = int(input('How many grades are there? '))

for counter in range(num_grades):
    grade = float(input('Enter a grade '))
    total = total + grade

average = total/num_grades
print('The average grade is', format(average, '.1f'))
```

```
How many grades are there? 4
Enter a grade 88
Enter a grade 92
Enter a grade 86
Enter a grade 97
The average grade is 90.8
```

Loop Accumulator

The accumulator inside the loop in Example 5.9 uses an expression common in programming, but impossible in Algebra. In mathematics, a value can never be equal to itself plus some value. In programming, this is an assignment statement and is perfectly acceptable. Assignment statements that have the same variable on both sides of the assignment operator are common. It is important to understand this concept. The right-hand side of an assignment statement is evaluated first by the computer and then the result is assigned to the left-hand

side. In this example, total is assigned the result of the current value of total plus grade.

total = total + grade

What the statement tells the computer is to go to the memory location for total and find out the value that is there. Then go to the memory location for grade and find out what is there. Next, add the two together and place the result in the memory location for total (over-writing the previous value).

Loop Counters

When the number of iterations a loop will execute is undetermined but is needed by the program, a ***counter*** variable is placed inside the loop. For example, a program that computes the average of a set of values needs to know the number of values that were entered in order to compute the average. The next example uses a counter within the loop to count the iterations and then uses the count to compute the average of the values entered. Note that the variables are initialized.

Ex. 5.10 – counting iterations of the loop

```
more = 'y'
counter = 0
total = 0

while more == 'y':
    num = float(input('Enter a sales amount '))
    total = total + num
    counter = counter + 1
    more = input('Enter "y" to continue or "n" to stop.')

average = total / counter
print('The average sale is ', format(average, '.2f'))
```

```
Enter a sales amount 23.76
Enter "y" to continue or "n" to stop.y
Enter a sales amount 19.95
Enter "y" to continue or "n" to stop.n
The average is  21.86
```

Loop Counter

Common Loop Algorithms

Loops are commonly used in programming to implement algorithms including accumulating a total and computing an average of values as shown previously. Others include validating input, finding the minimum or maximum of a set of numbers, or finding a match. This example requests a number within a specific range and the loop continues the request until a valid number is entered.

Ex. 5.11 – input validation

```
num = int(input('Enter a number between 1 and 10 '))

while num < 1 or num > 10:
    print('\tThat is not a valid number.')
    num = int(input('Enter a number between 1 and 10 '))


print('That is a valid number.')
```

```
Enter a number between 1 and 10 12
        That is not a valid number.
Enter a number between 1 and 10 3
That is a valid number.
```

Input Validation Loop

The next example determines the minimum value from a set of inputs. Notice that the first number entered is assigned to the variable smallest. Since it is the only number entered so far, it is the smallest. An explanation follows the code.

Ex. 5.12 – Find the Minimum

```
more = 'y'

num = int(input('Enter an integer '))
smallest = num

while more == 'y' or more == 'Y':
    num = float(input('Enter another integer '))
    if num < smallest:
        smallest = num
    more = input('Enter "y" to continue or "n" to stop.')

print('The smallest number entered was ', smallest)
```

Any number that is input that is smaller than the one currently stored in smallest, is assigned to smallest. When the condition is false, the loop ends and the final print statement executes.

```
Enter an integer 3
Enter another integer 6
Enter "y" to continue or "n" to stop.y
Enter another integer 8
Enter "y" to continue or "n" to stop.y
Enter another integer 2
Enter "y" to continue or "n" to stop.n
The smallest number entered was  2.0
```

Determining the Smallest Value

The algorithm for finding the smallest can be modified to determine the largest. The code below stores the smallest and the largest values in a single loop.

```
num = int(input('Enter an integer '))
smallest = num
largest = num

while more == 'y' or more == 'Y':
    num = float(input('Enter another integer '))
    if num < smallest:
        smallest = num
    if num > largest:
        largest = num
```

Determining the Smallest and Largest Value

Sentinels

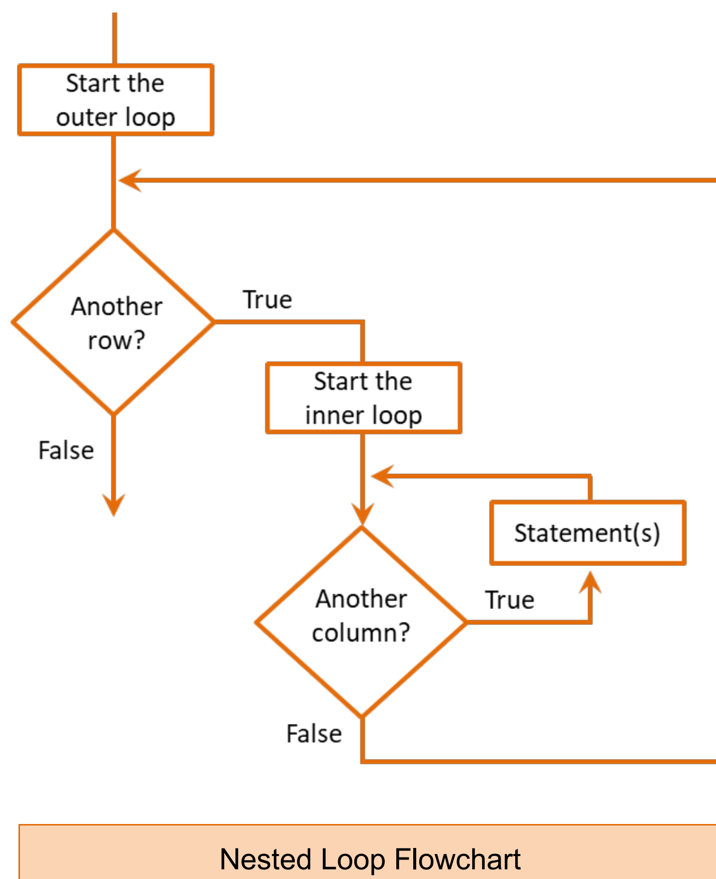
In some of the example programs, the user was asked to enter 'y' to continue. In programming, a ***sentinel*** is often used to indicate that the end of the input has been reached by having the user enter a number that could not be part of the set of values. As an example, if a program is requesting positive integers as input, the user may be prompted to enter -1 when finished. If a program is requesting numeric test grades, the user may be prompted to enter 999 when finished. The point is that a sentinel is a value that is outside the set of values being used by the program. It is intentionally beyond a reasonable input value.

Nested Loops

When a loop is contained inside another loop, it is called a nested loop. An outer loop is entered and an inner loop executes. When the inner loop completes, if there are more outer loop iterations to complete, it again initiates the inner loop. A good example of this operation is a set of rows and columns. The output would display a row of data across (each column), then the next row and all of its columns, and so on.

```
While there is a row of values to print
    While there is a column value to print
        print the value
```

As an example, a program that displays four (4) rows of data having three (3) values (3 columns) would have an outer loop that iterates four times (rows) and an inner loop that iterates three times (columns). For each execution of the outer loop, the inner loop executes three times.

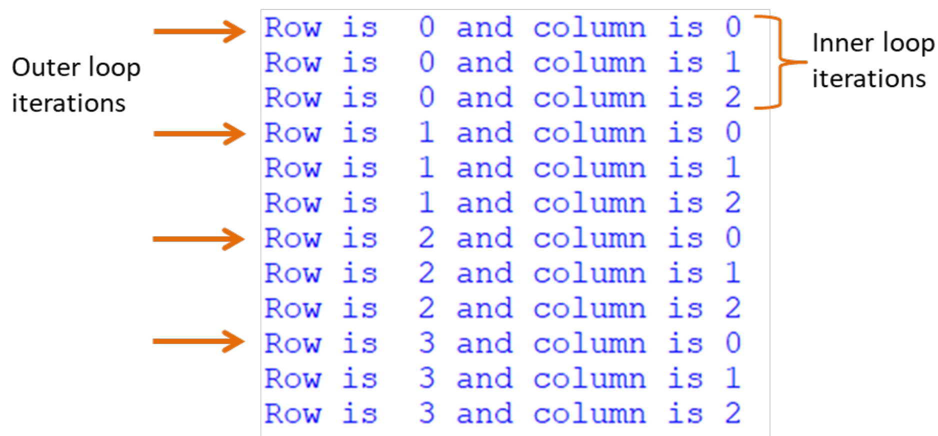


Nested Loop Flowchart

The code below for the example shows the nested loop and a print statement that displays the row and column numbers. For each repetition of the outer loop, the inner loop executes three repetitions.

Ex. 5.13 – Nested Loop

```
for row in range(4):
    for column in range(3):
        print('Row is ', row, 'and column is', column)
```



Nested Loop

Common Loop Errors

Common errors associated with loops include off-by-one errors, where the programmer has written the conditional expression or range incorrectly and the loop is executing one too many or one too few times. This is easily corrected after running and testing the program. Others include confusing what should be inside the loop and what should be outside the loop. When these types of issues occur, adding print statements before, within, and after the loop can help to determine where the problem is located.

A Complete Example – Investment Program

Requirements:

Write a program for a Financial Adviser that computes the number of years to double a $10,000 investment at a given annual interest rate.

Program Pseudocode:

Step 1 Prompt for the interest rate

Step 2 Compute the interest on the balance

Step 3 Add the interest to the balance

Step 4 Increment the number of years

Step 5 Is the balance < $20,000.00

Yes, go back to Step 2

No, got to Step 6

Step 6 Display the number of years

Verbalizing and walking through the steps that the program will take is referred to as ***Storyboarding***, and can often help with determining the sequence and order of operations for the program.

"Set up the program by initializing the balance and years, and obtain the interest rate from the user. While the balance is less than $20,000.00, compute the interest on the balance and add it to the previous balance".

"When the balance is no longer less than $20,000, display the number of years".

Development

The development of the program follows the pseudocode and storyboard. The balance and years are initialized, and the interest rate is obtained from the user. As long as (while) the balance is less than the target amount, compute the interest and add it to the balance, and increment the years. The solution below includes a print statement within the loop for test purposes.

Investment Program Code

```
balance = 10000
years = 0

int_rate = float(input('Enter the interest rate : '))

while balance < 20000:
    interest = balance * (int_rate / 100)
    balance = balance + interest
    years = years + 1
    print('balance is $', format(balance, ',.2f'))


print('The balance doubled in ', years, 'years')
```

```
Enter the interest rate : 4.5
balance is $ 10,450.00
balance is $ 10,920.25
balance is $ 11,411.66
balance is $ 11,925.19
balance is $ 12,461.82
balance is $ 13,022.60
balance is $ 13,608.62
balance is $ 14,221.01
balance is $ 14,860.95
balance is $ 15,529.69
balance is $ 16,228.53
balance is $ 16,958.81
balance is $ 17,721.96
balance is $ 18,519.45
balance is $ 19,352.82
balance is $ 20,223.70
The balance doubled in  16 years
```

The print statement within the loop can be removed after testing.

Testing and Debugging

The development isn't complete until the program is tested and verified for accuracy. Testing can also surface questions about the requirements for the program. The output states that the balance doubled in 16 years, but the balance is actually more than double the initial amount at that point. Since interest is being computed and added annually, the program meets the requirements, but it might be a good idea to ask the Financial Advisor if this is an adequate solution.

Chapter 5 Review Questions

1. A structure that allows repeating steps without repeating code is referred to as a _______________ structure.
2. A loop that repeats while some condition is true is a ________ controlled loop.
3. Each execution of a loop is referred to as an ____________.
4. A while loop is a ___________ loop and may or may not execute depending on the conditional statement.
5. A loop that repeats a specific number of times is a _________ controlled loop.
6. A loop that continues to run without a control or condition to end the loop, it is referred to as a(n) ___________ loop.
7. The range function provides a simplified way to write _______ controlled loops.
8. When one argument is passed to the range function, it is the ________.
9. When two arguments are passed to the range function, the first argument is the _________ and the second is the _________.
10. When three arguments are passed to the range function, the third argument is the ________.
11. A variable within a loop that tallies a running total is an _____________.
12. A variable within a loop that counts the number of iterations of the loop is referred to as a _____________.
13. A value entered by the user that is used by the program to indicate the end of a data set is referred to as a ___________.
14. A loop within a loop is referred to as a __________ loop.
15. Verbalizing and stepping through program operation is known as __________.

Chapter 5 Short Answer Exercises

1. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
while var1 < var2:
        print(var1, end='')
        var1 = var1 + 1
```

2. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
while var1 < var2:
        print(var1, end='')
        var1 = var1 + 2
```

3. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
while var1 <= var2:
        print(var1)
```

4. What do the following lines of code output if *first* = 10, and *second* = 10?

```
while first < second:
        print('Enter a number')
```

5. What do the following lines of code ensure?

```
value = int (input("Enter a positive number')
while value < 1:
        value = int (input("Enter a positive number')
```

6. What do the following lines of code output?

```
for counter in range(3):
        print(counter, end=" ")
```

7. How many times will following loop display "Hello"?

```
for counter in range(3, 7):
        print('Hello')
```

8. How many times will following loop display "Another"?

```
for counter in range(2, 10, 2):
        print('Another')
```

9. In the following code, what term would be used to describe the variable total?

```
for counter in range(3):
        grade = float(input('Enter a grade')
        total = total + grade
```

10. In the following code, what term would be used to describe the variable num?

```
for counter in range(3):
        grade = float(input('Enter a grade')
        total = total + grade
        num = num + 1
```

11. In the following code, what term would be used to describe 999?

```
while grade != 999:
        grade = float(input('Enter a grade or 999 when finished.')
```

Chapter 5 Programming Exercises

1. Write a program that sets a variable num to 0 and uses a while loop to display the numbers 0 thru 9 separated by a space using the variable.

2. Write a program that sets a variable num to 0 and uses a while loop to display the even numbers 0 thru 20 separated by a space using the variable.

3. Write a program with a for-in-range loop that displays the numbers 0 thru 9 separated by a space using a variable.

4. Write a program with a for-in-range loop that displays the even numbers 0 thru 20 on a single line separated by a colon using a variable.

5. Write a program that prompts for a positive integer, and uses a while loop to validate the input. If the number entered is not a positive number, the loop will output the error message "Invalid input", and request another number.

6. Write a program that prompts for a number between 1 and 10 inclusive, and uses a while loop to validate the input. The loop will output the error message "Invalid input", and request another number. When a valid number has been entered, output "Thank you."

7. Write a program that displays a heading and the columns of data shown below containing the number 1 thru 10 and the squares of the numbers. Use the tab escape character and width specifier as needed. Sample output below.

```
Number              Square
--------------------------
   1                   1
   2                   4
   3                   9
   4                  16
   5                  25
   6                  36
   7                  49
   8                  64
   9                  81
  10                 100
```

8. Write a program that uses two variables num1 and num2, and uses a nested loop to display the following output of 3 rows with 5 columns. Consider where the line feed should be located in the loop structures.

```
1       2       3       4       5
1       2       3       4       5
1       2       3       4       5
```

9. Write a program that prompts for a word from the user, and then displays each character in the word separated by a tab. As an example, if the word "*Python*" were entered:

```
Enter a word Python
        P       y       t       h       o       n
```

10. Write a program that prompts for a word from the user, and then displays every other letter with a space between them. As an example, if the user entered "*Exceptional*":

```
Enter a word Exceptional
E c p i n l
```

11. Modify the program from chapter4 that prompts the user for a temperature and then "F or f" if it is a Fahrenheit temperature or "C or c" if it is a Celsius temperature to convert. Display both of the temperatures or "That the letter is invalid" if an incorrect letter was entered. Allow the user to convert another temperature without restarting the program. A sample run is shown below.

$C = (F - 32) / 1.8$ $\qquad$ $F = (C * 1.8) + 32$

Sample program run:

```
Enter a temperature 43
Enter an "F" for Fahrenheit or "C" for Celsius F
43.0 Fahrenheit is  19.80 Celsius

Enter "y" to run again. y

Enter a temperature 22
Enter an "F" for Fahrenheit or "C" for Celsius J
The letter  J  is not valid
A conversion can not be completed.

Enter "y" to run again. y

Enter a temperature 22
Enter an "F" for Fahrenheit or "C" for Celsius C
22.0 Celsius is  71.60 Fahrenheit

Enter "y" to run again.
```

12. Modify the Wind Chill program from chapter 4 to allow the user to compute multiple wind chill factors without restarting the program. Validate the input for temperatures less than or equal to 50 degrees and wind speeds > 3.0 mph. Display an error message and prompt again when invalid data is entered.

    Sample program run:

```
Enter the temperature in Fahrenheit: 56
The wind chill is invalid above 50 dF.

Enter the temperature in Fahrenheit: 33

Enter the wind speed in mph: 2
The wind chill is invalid at wind speeds <= 3.0 mph.

Enter the wind speed in mph: 5.7

The wind chill factor is  27.66

Enter "y" to run again.
```

13. Write a program that requests a cable length (must be positive) from the user and a cable thickness (must be between 0.1 and 2.5) in inches. Validate the input and keep requesting until valid input is received. Write a loop that will

simulate applying one pound of tension and stretching the cable for each repetition of the loop.

The cable will stretch its thickness times 0.28 feet for each pound of tension applied. The cable will break when it is 112% of its original length. Output the pounds of tension on the cable and the length for each pound applied, and announce when the cable has broken.

Sample program run:

```
Enter a cable length in feet: 24
Enter the cable thickness (0.1 - 2.5) inches: 1.75

Tension: 1 lbs.   length: 24.5
Tension: 2 lbs.   length: 25.0
Tension: 3 lbs.   length: 25.5
Tension: 4 lbs.   length: 26.0
Tension: 5 lbs.   length: 26.4
Tension: 6 lbs.   length: 26.9

The cable has broken.
```

Chapter 5 Programming Challenge

Drainage Canal

The canal has a natural flow rate of 40 ft^3 /s at 3.3 feet. Rainfall increases the water level of the canal and a flood gate must be opened to remove the excess water.

Prompt the user for the water level in feet (must be > 3.3) and the number of feet (in 1 foot increments) to open the flood gate (must be >= 1), and compute the time to lower the level to 3.3 feet while displaying the hours passed, and the current level of the canal to two (2) decimal places.

The program will simulate the discharge of water through the flood gate at a rate of 0.03 feet of water per minute for each foot that the food gate is open. This will continue until the water level in the canal has reached 3.3 feet.

The program will validate the input only allowing a water level > 3.3 and a gate opening >= 1, and then start a loop to simulate draining the canal and display the hours passed and current water level each time the loop executes. The program will announce when the canal has reached the natural level of 3.3 feet and end.

Note the output alignment in the sample below.

Sample program run:

```
Enter the water level in feet: 4.6
Enter the gate opening in 1 foot increments: 3

Minutes Passed:    1    Water Level:  4.51  feet
Minutes Passed:    2    Water Level:  4.42  feet
Minutes Passed:    3    Water Level:  4.33  feet
Minutes Passed:    4    Water Level:  4.24  feet
Minutes Passed:    5    Water Level:  4.15  feet
Minutes Passed:    6    Water Level:  4.06  feet
Minutes Passed:    7    Water Level:  3.97  feet
Minutes Passed:    8    Water Level:  3.88  feet
Minutes Passed:    9    Water Level:  3.79  feet
Minutes Passed:   10    Water Level:  3.70  feet
Minutes Passed:   11    Water Level:  3.61  feet
Minutes Passed:   12    Water Level:  3.52  feet
Minutes Passed:   13    Water Level:  3.43  feet
Minutes Passed:   14    Water Level:  3.34  feet
Minutes Passed:   15    Water Level:  3.25  feet

 The Water Level is now at 3.3 feet.
```

Chapter 6

Functions

As programs become longer and execute more tasks, the main function grows and code may be repeated in order to repeat functionality. The design process includes dividing the program into logical sections of distinct functionality which will be developed individually. This is referred to as ***modularization***. Separating the program into distinct parts provides many benefits including the ability to: reuse portions of the code, divide the program development among multiple programmers, and simplify the task. Instead of writing one long program, sections can be developed in ***functions***, and then the functions can be ***called*** when needed, and as many times as needed. Some functions, like *print, input, range,* and *round* have already been used in previous programs. There are many more available, and programmers write their own functions as well.

There are two types of functions, ***void functions*** that just perform a task and ***value-returning functions*** that return a value. The *print* function is a void function, and *input* is an example of a value-returning function. Note the difference in their use below. The *print* function simply displays what is passed to it, but the *input* function returns something that is assigned to a variable.

```
print('A void function')

value = input('Enter a number: ')
```

Void Functions

The code for a function is called the ***function definition*** and it begins with the keyword ***def*** which is followed by a name for the function, a pair of parentheses, and a colon. This first line is of the definition is referred to as the ***function header***. The statements that will execute when the function is called are indented and form a block of code and are referred to as the function body. The general format is:

```
def function_name():
    statement1
    statement2
    etc.
```

The program below prints a statement, then calls a function, and then prints another statement. There are a few differences in this program. First, the *main* function is defined, and the general format for *main* is similar to the format for a void function. Every program has a main function where execution begins when the program is launched. The programs covered previously were run by the IDLE interpreter without a main function, but now is a good time to include the main function. Second, after the definition for the *show_output* function, there is a call to *main*. This is what executes the program. The interpreter reads through the lines of code, and when it reaches the last line it executes the *main* function

Ex. 6.1 – Function definition including "main"

```
def main():                # main function header
    print('In main')

    show_output()          # call to show_output

    print('Back in main')


def show_output():         # function definition
    print('Now in show_output')


main()                     # call the main function
                           # and begin execution
```

Function Definition with Main

As shown in example 6.1, the function call is made at the point in the code when it is to be executed. The function definition that contains the lines that will be executed when the function is called, are located separately. When the function is called, control of the program moves to the function, it executes, and then control returns to the point where it was called. This is shown by the output from the example.

```
In main
Now in show_output
Back in main
```

The program in Example 6.1 is repeated here with line numbers for explanation.

```
 1 # Main function with a call to show_output
 2 def main():                 # main function header
 3     print('In main')
 4 
 5     show_output()           # call to show_output
 6 
 7     print('Back in main')
 8 
 9 
10 def show_output():          # function definition
11     print('Now in show_output')
12 
13 
14 main()                      # call the main function
```

Notice that the function definition starting on line 10 seems to be inside main, but it is below main and out-dented the same as main. Line 14 is the call to main which begins execution of the program. On line 5, the output function is called and executes, and when it completes, control returns to main.

Step 1 the interpreter reads through the program

Step 2 the main function is called (line 14)

Step 3 the print statement in main executes (line 3)

Step 4 the show_output function is called (line 5)

Step 5 the body of the function executes (line 10)

Step 6 the print statement back in main executes (line 7)

Function Program Order of Operations

A reminder about ***indentation*** is warranted. Indentation forms a block of code in Python. The function names, including main begin at the margin, and the function bodies are indented forming a block of code for the function. The IDE highlights items by color-coding the text as shown below. Also note that it is much easier to use the tab key for indentation than to count spaces to be sure they are always the same.

```
def main():

    function1()

    function2()

    function3()

def function1():
    print('F1')


def function2():
    print('F2')


def function3():
    print('F3')

main()
```

Variables and Scope

The part of a program where a variable is accessible is referred to as the variable's ***scope***. When a variable is declared within a function (including main), the scope of the variable is the function. It is referred to as a ***local variable***. A variable defined inside a function is not accessible outside that function, so different functions could have variables with the same name without causing any conflict. Each of the variable's scope would be their particular function, and would not be accessible by another function. If several engineers are working on the same program, but they are working on different functions, they may name a local variable using the same name. Again, there would not be a conflict. To

demonstrate this, the following program has a function and a main function, and each one defines a variable called *word*. The print statements are used to highlight that they are different variables (with different memory locations) although they have the same name.

```
def main():
    word = 'main'

    function1()

    print(word)


def function1():
    word = 'one'
    print(word)


main()
```

```
one
main
```

The output of the program highlights the scope of each of the variables. Even though they have the same name, because of scope, they are different variables with different values.

If the program attempted to access a variable outside of its scope, an error would occur. In addition, attempting to access a variable in a function before it has been defined will cause an error. It is always best to declare all variables needed by a function together and first within the function. This makes readability and maintenance much easier and eliminates the potential for errors.

Global Variables

Defining a variable outside any function in Python makes it ***global*** and accessible to all areas of the program. A function that needs to change the variable precedes it with the keyword global. Other functions can just use it as needed. Global variables should be used sparingly if at all because their use makes debugging very difficult since any part of the program can change a

global variable. In addition, any function that accesses and uses a global variable is dependent on that variable and cannot easily be used in another program.

This example defines a global variable, assigns it the value 2, and main changes the value since it is preceded by the word global. The function displays the value proving that it has access to the global variable and that it has been changed.

Ex. 6.2 – Global variable access

```
num = 2              # global variable

def main():

    global num       # global keyword used

    num = int(input('Enter a number '))

    show_output()


def show_output():
    print(num)


main()

                  Enter a number 24
                  24
```

There are two occasions when having global variables in a program is beneficial and improves efficiency.

1. In a collaborative programming environment when multiple engineers are working on the same project and a consistent value is required. For example, if a project involves multiple engineers working in multiple files and they are writing equations that use the diameter of the Earth, a global variable for the diameter may be declared. The Earth is not a perfect sphere and there are variations in the diameter.
2. A project that uses a value or set of values in many places, and the values tend to change. For example, financial programs need to be changed frequently or at least annually since tax percentages and ranges change. Declaring them as global variables requires only changing them in one place, as opposed to searching through all of the code to ensure that everywhere they are used is updated with the new value.

In these instances, global variables are typically handled as ***global constants***. As noted in Chapter 3, a constant is a value that cannot (should not) be changed by the program. Python does not formally have constants, but they can be implied by following the standard for naming constants with all uppercase letters and underscores as shown here.

GLOBAL_CONSTANT = 2.76

Without using the keyword global, the value cannot be changed by a function, but it will appear that it does even though a new variable has actually been declared. Note the output of the following example that attempts to change a global variable, but is really declaring a new variable with the same name. The global constant is not changed by the function. The function simply has a variable of the same name.

Ex. 6.2A – Global variable incorrect use

```
GLOBAL_CONSTANT = 2.76

def main():

    print(GLOBAL_CONSTANT)
    change()
    print(GLOBAL_CONSTANT)

def change():

    GLOBAL_CONSTANT = 27
    print(GLOBAL_CONSTANT)


main()
```

```
2.76
27
2.76
```

Preceding the global variable with the keyword global allows it to be changed, but this cannot be done in a single step. The variable must be defined as global first and then an assignment can be executed as shown here.

```
global GLOBAL_CONSTANT
GLOBAL_CONSTANT = 27
```

The following example modifies the global variable in the function. Note that two lines of code are required. First, the variable is acknowledged as being global, and then the assignment changes the value.

Ex. 6.2B – Global variable modified by a function

```
GLOBAL_CONSTANT = 2.76

def main():

    print(GLOBAL_CONSTANT)
    change()
    print(GLOBAL_CONSTANT)

def change():

    global GLOBAL_CONSTANT
    GLOBAL_CONSTANT = 27
    print(GLOBAL_CONSTANT)


main()
```

```
2.76
27
27
```

Passing Values to Functions

Functions cannot access variables outside their scope, but often they need to use them. When a function needs to use a variable defined somewhere else in the program, the variable is passed to the function as an ***argument***. To the function receiving the argument, it is referred to as a ***parameter***. Technically speaking, arguments are passed to functions and parameters are received by them. In the following example, main calls a function and passes a variable (argument) to the function that receives it (parameter) into a local variable, and uses it in an equation and print statement. Notice that *argument* is the name of the variable passed as the argument, and *parameter* is the name of the variable receiving the parameter in the function header. This highlights that a value is passed. The computer goes to the memory location for the variable *argument*, finds out what is stored there and passes a copy of it to the function. The function receives the

value and stores it in the memory location for *parameter*. This is referred to as ***pass-by-value***.

It doesn't matter what the receiving function calls the value that it receives, except that it must use that name internally. The parameter variable is actually a local variable to the function and has the function as its scope.

Ex. 6.3 – Passing an Argument to a Function

```
def main():
    argument = 4

    square_it(argument)


def square_it(parameter):
    item = parameter * parameter
    print(item)


main()
```

Passing an Argument to a Function

The argument passed to a function does not need to be a variable. A literal can be passed as well. The function in the example could be called as shown here.

```
square_it(4)
```

Passing Multiple Values to Functions

When needed, multiple arguments can be passed to functions as long as there are parameters to receive them. The arguments can be different data types, and they will be received by the function in the order that they are passed *(see the Technical note below)*. Example 6.4 demonstrates passing three arguments to a function. The first argument is a number, the second is a string, and the third is another number. They are received into p1, p2, and p3 and are used appropriately. If the code tried to multiply p1 and p2, the output would be quite different since p2 is a string.

Ex. 6.4 – Passing multiple Arguments to a Function

```
def main():
    arg1 = 10
    arg2 = 'Product'
    arg3 = 27

    product(arg1, arg2, arg3)


def product(p1, p2, p3):
    value = p1 * p3
    print(p2 + ' is: ', value)


main()
```

```
Product is:  270
```

Passing Multiple Arguments to a Function

Technical note: Python allows keyword arguments which specify the parameter that an argument is assigned to by the receiving function. The item to be received by the parameter is assigned to it in the function call argument list. The function below has two parameters, and when the function is called the arguments are assigned to the receiving parameters in a different order than they are passed. They are assigned correctly. The use of keyword arguments requires specific knowledge of the function's parameter list, and can negatively affect readability and reuse.

```
def main():

    display(num2 = 3, num1 = 4)


def display(num1, num2):
    print(num1)
    print(num2)


main()
```

```
4
3
>>>
```

Value Returning Functions

Value returning functions are different from the void functions covered so far in two ways. They return a value to the calling part of the program, and they have a return statement. The general format is shown below.

```
def function_name():
    statement1
    statement2
    etc.
    return something
```

Notice that the last line of the function is the return statement. This follows the practice of logical programming: input, processing, and output. The variable something would be defined in the function, assigned a value, and its' value would be returned by the function.

Ex. 6.5 – Value-returning function

The function *get_input()* is called from main first. The function obtains user input and assigns it to num. The function returns the value stored in num and it is received back in main and is assigned to the variable usernum. Then the print statement executes.

```
def main():

    usernum = get_input()
    print('User entered ', usernum)



def get_input():
    num = int(input('Enter a number '))
    return(num)


main()
```

```
Enter a number 23
User entered  23
```

Value-returning Function

Functions can return Boolean values (True/False) and strings as well.

Ex. 6.6 – Value-returning function data types

The function *get_password()* is called from main and returns a string. The string is assigned to pw and is then passed to the function *long_enough()* which returns a Boolean value (true or false). The Boolean value is passed to the function *output()* which determines the print statement. Notice that main simply consists of three function calls, and that the function definitions are in the same order that the functions are called. This is a programming practice that enhances readability.

```
def main():

    pw = get_password()
    valid = long_enough(pw)
    output(valid)


def get_password():
    user_pass = input('Enter a password ')
    return(user_pass)

def long_enough(user_name):
    if len(user_name) > 9:
        ok = True
    else:
        ok = False
    return ok

def output(good):
    if good:
        print('That is valid')
    else:
        print('Not valid')


main()
```

The example demonstrates modular programming with most of the code being executed in functions. The main function becomes a series of function calls to execute the program. Most of the functionality that a program executes can be placed in a function. As the requirements are decomposed and the Design Phase begins, areas of the program that lend themselves to being functions will surface. Once, the functionality is determined, the function can be defined.

Technical note: Python allows returning multiple values from functions. The general format is:

```
return value1, value2, etc.
```

The return values must be received in the order that they are returned and are separated by a comma.

```
first, second = get_two_values()
```

Writing a function that returns multiple values could be considered too specific to a particular use or program since the function could not easily be reused elsewhere. Their use can also negatively affect the readability of the program.

Defining and Naming Functions

When creating a function, there are several things to consider and steps that can help in the process. First, determine what the function will do. Each function should accomplish one task, and the name of the function should describe what it does. Function names follow the same rules and conventions for variables with all lowercase letter and words separated by underscores. Since functions perform an action, verbs are usually used to name functions like get_tax_rate, or compute_gross_pay. Once the function task and name have been decided, determine what parameters the function needs in order to accomplish the task. Then, determine if the function will return a value and if so, what data type will it return.

Step 1 name the function what it does

Step 2 determine the parameters that it needs

Step 3 determine if the function will return a value

- If yes, determine the return type

A tool that is helpful with function design as well program design is an Input, Processing, Output document or ***IPO***. An IPO may be in the form of a chart or document, and includes the name of the function, a brief description of what it does, the input needed, the processing it will accomplish, and the output or return value. An IPO can also be used for the overall program. The next example uses functions to obtain a number from the user, compute the square of the number, and display the result. An IPO for the program follows the code.

Ex. 6.7 – IPO (Input, Processing, Output) Documentation

```
def main():

    usernum = get_input()
    square = compute_square(usernum)
    output(square)


def get_input():
    num = int(input('Enter a number to square '))
    return(num)

def compute_square(number):
    sqr = number * number
    return sqr

def output(value):
    print('The number squared is:', value)


main()
```

Program IPO:

Description: the program calls three functions to obtain user input of a number, square the number, and display the square of the number.

Input: number from user

Processing: square the number

Output: display the result

Function IPOs:

get_input()

Description: Obtains user input

Input: number from user

Processing: none

Output: returns the number

compute_square(number)

Description: Computes the square of the number

Input: a number

Processing: square the number

Output: return the value

output(number)

Description: Produces output

Input: the value to display

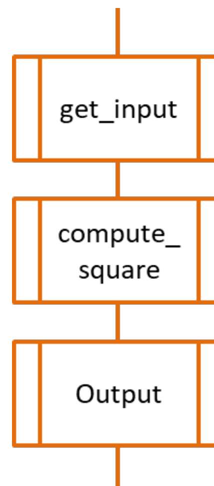
Processing: none

Output: display the result

IPO Documentation

The IPO content is subjective and differs among organizations that use them, but the overall concept is consistent with its name. They are another tool that can be used in the design process to save time and produce modularized programs.

Flowcharts which were covered previously can help with modularization as well and to visualize the order of operations. The flowchart symbol typically used to depict a function is a rectangle with side-bars. A partial flowchart of the example program is shown below.



Function Flowchart Symbol

Functions and Methods - Terminology

Some confusion may arise as a result of different languages using different terminology with respect to functions. For example, Java uses the term method,

C/C++ uses the term function, and Python uses both function and method. For clarification, the Python definitions follow.

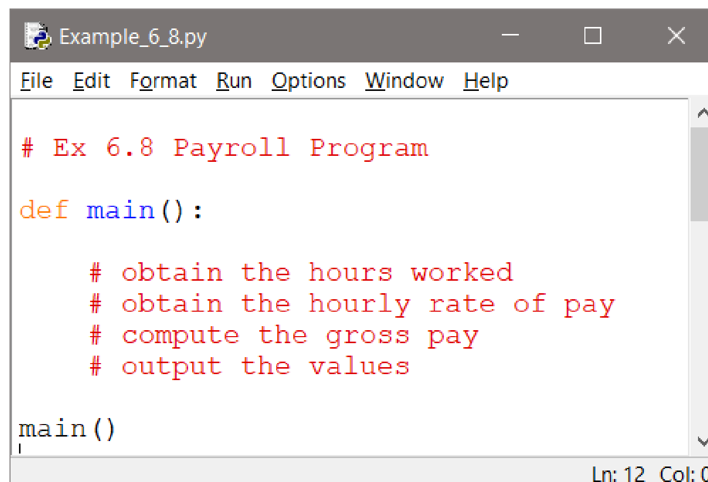
Function – a named block of executable statements

Method - a function that exists inside of an object

Modular Programming with Files

Modularizing programs using functions separates operations into manageable chunks and enhances maintenance, but multiple engineers cannot easily work on the same program because it is in a single program file. Separating the program into files (modules) allows multiple engineers to work on the same program at the same time, permitting ***collaborative development***. Collaborative environments like CMS covered in Chapter 1 facilitate the development and control of multi-file programs. As discussed previously, large and complex program requirements are decomposed during design into manageable sections, and are then further refined into functions. Functions that are related are developed in their own files, and the files are then imported into the main program. Example 6.8 is a payroll program that computes gross pay from the number of hours worked and hourly rate of pay for a user. The functions will be developed in a separate file which will be imported into the main file.

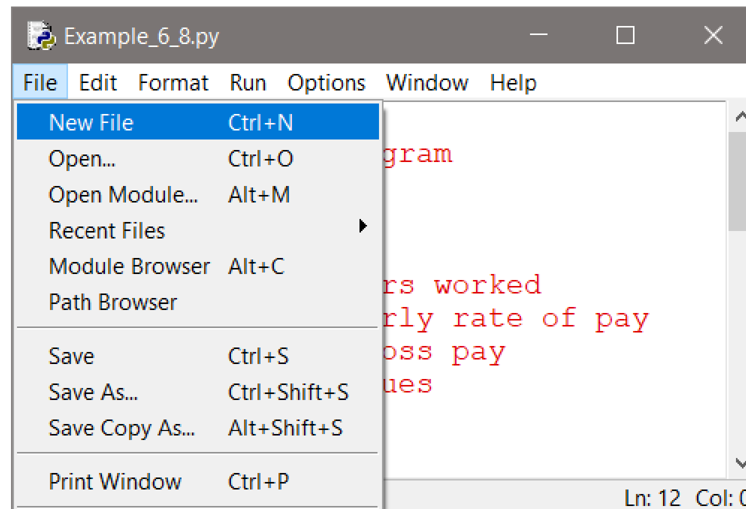
Ex. 6.8 – placing functions in a separate file



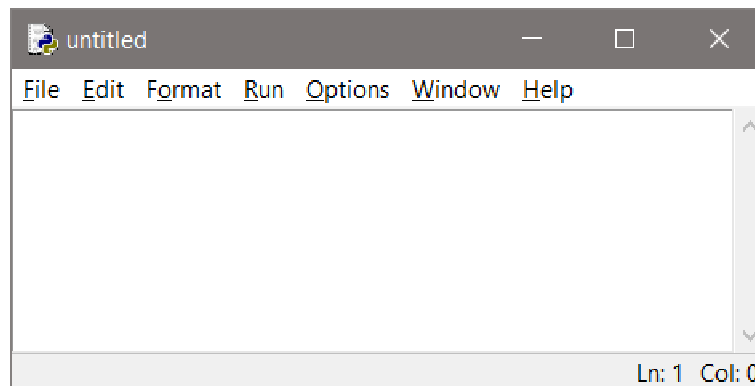
The main file is shown above with pseudocode comments to map out the program. Note that main is defined and the last line in the file calls main.

Ex. 6.8A – creating the accompanying file for the program

The file that will contain the function is created using File | New File from the menu of the main program file. The directory location for the new file defaults to the current directory keeping the files together.



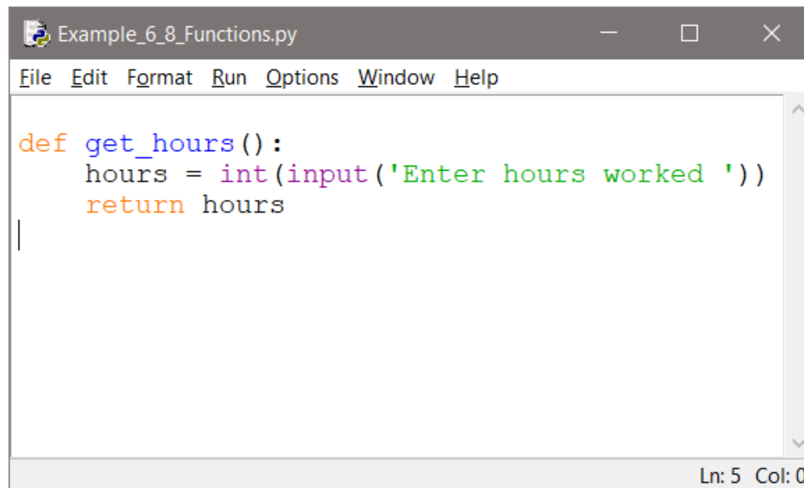
An unnamed (untitled) file is created.



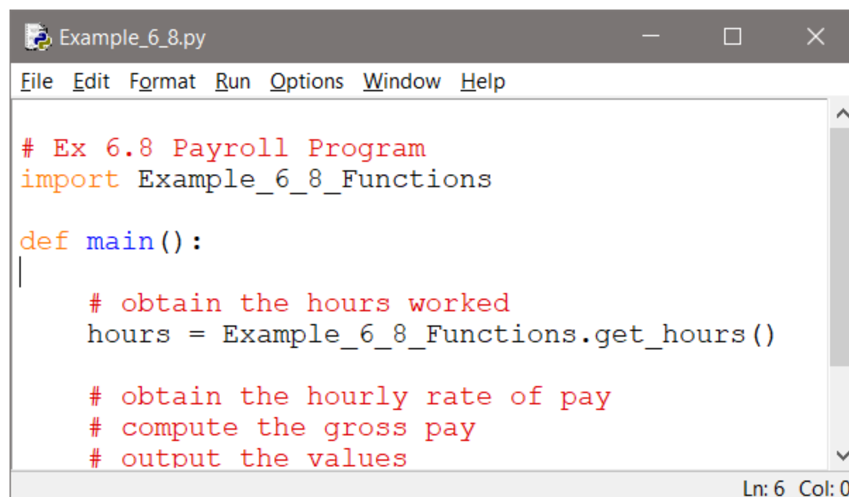
Ex. 6.8B – developing an imported function

It is always best to use the "***build-a-little, test-a-little***" process when developing programs. In other words, develop a small part of the program and test and debug that part until it is working correctly. Then, develop another small part and test and debug the program with the additional part. This process is often referred to as incremental programming or ***iterative enhancement***. The first function for the example program obtains the number of hours worked, and

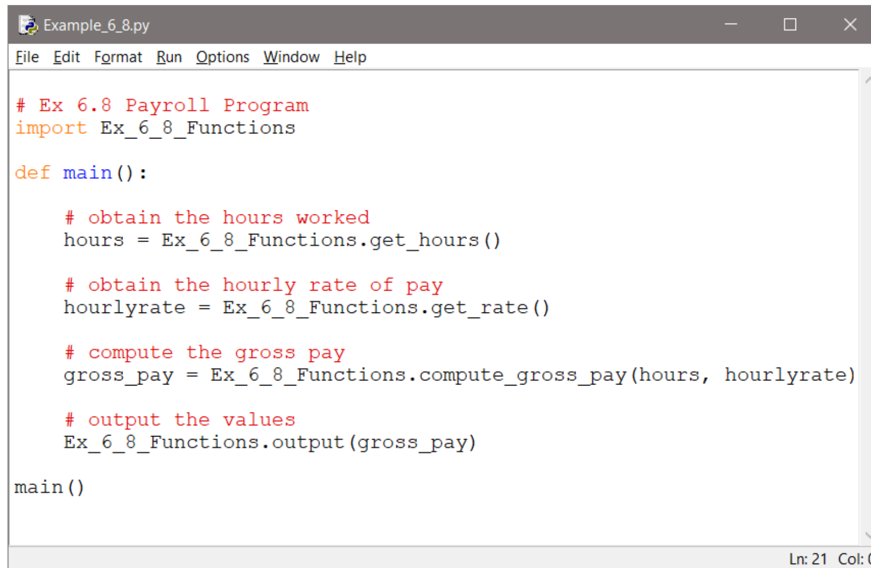
the function definition is written in the new file. The file is saved and given a name that identifies what it contains.



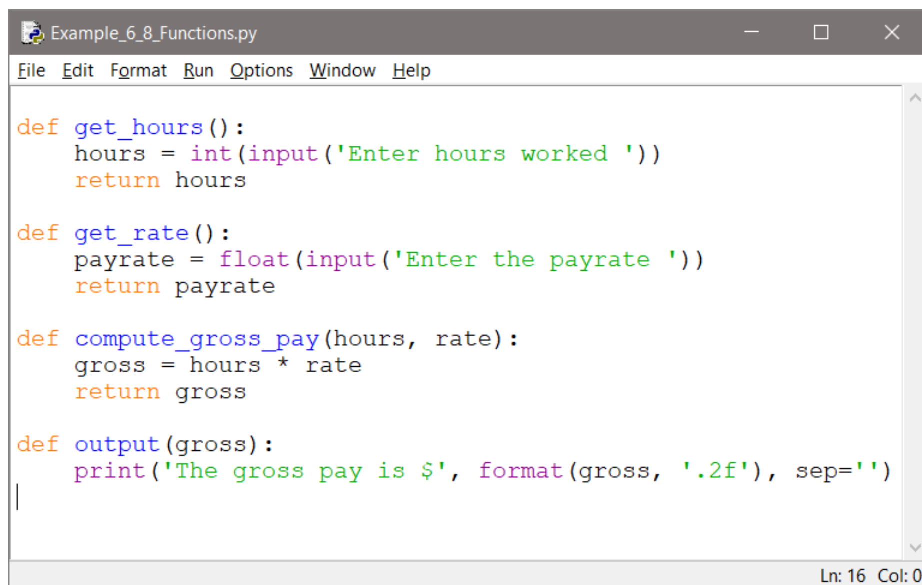
In order to use this function, the file must be imported into the main file and the function is called using the name of the file (omitting the .py file extension), the dot operator, and the name of the function as shown below.



The other functions are added and their calls from main are handled the same way. Notice that because of variable scope it does not matter if the variables in the main function and another function have the same name. They are local to their functions and there is no conflict. The completed definition for the main function is shown below.



The completed file containing the functions is shown here.



Function Module

Modular programming supports the reuse of code that has already been written and a lot of code has been written over the years. To take advantage of this, Python has an extensive set of modules that provide functions for use in programs. They are often referred to as libraries because they contain groups of

related functions. There are currently thousands of Python libraries. A couple of them are used extensively and will be covered next.

The Python Math Module

The Python mathematical functions are contained in the math module. This module is readily available in the Python shell, but must be imported when used in program files. The list of math functions includes: *acos(x), asin(x), atan(x), cos(x), hypot(), log(x), sin(x), sqrt(x),* and *tan(x)*. The module also defines a value for *pi* and *e*, and provides conversions for degrees to radians, *radians(x)*, and radians to degrees, *degrees(x)*. All of these functions return float values. A complete listing of math functions is available at Python.org. When using the math functions, the word math and the dot operator precede the functions.

Ex. 6.9 – math library square root function

```
import math

def main():

    num = 3 * 3

    print('Num is ', num)

    sr_num = math.sqrt(num)

    print('Square root of num is', sr_num)

main()
```

```
Num is  9
Square root of num is 3.0
```

Math Library Square Root Function

The values for *pi* and *e* are used in the same way. They are preceded by the word math and the dot operator. The next example uses the math library value for pi to compute the surface area for a sphere. The equation is:

$$\textit{Surface area} = 4\,\pi\,r^2$$

Ex. 6.10 – math library constant value for pi

```
import math
def main():
    radius = 5
    sa = 4 * math.pi * radius**2
    print('Surface area is ', format(sa, '.2f'))

main()
```

```
Surface area is  314.16
```

Math Library Constant pi

The Random Module

Many programs generate random numbers including simulations and games. They are used to determine random event occurrences, and often for encryption. Python includes random number generation with library functions that require importing the ***random*** module. Arguments are passed to the functions that are used to determine the range of random numbers that could be generated. The functions are preceded by the module name random and the dot operator. The random number functions accept arguments that determine the selection specifics for the random numbers they generate.

The first line below imports the random library and the second line uses ***randint*** to assign num a random integer within the range of 1 to 100 inclusive. Note that the range includes 1 and 100.

```
import random
num = random.randint(1, 100)
```

Similar to the range function covered previously, this line uses ***randrange*** to assign a random integer from 0 thru 9 to num. Note that 10 is excluded.

```
num = random.randrange(10)
```

This line of code assigns a random integer to num between 0 and 100 by passing 101 as the limit for the range.

```
num = random.randrange(0, 101)
```

This line of code assigns a random integer between 0 and 100 to num, stepping by 10s (0, 10, 20, 30, etc.). Note again that 101 is used as the limit.

```
num = random.randrange(0, 101, 10)
```

Most random number generators in other programming languages generate a number between 0.0 and 1.0. Consider that there are actually many values in that range and the returned value can be modified for any purpose. Python provides for floating point random numbers between 0.0 and 1.0 as well. The random function is used as shown here and an example follows.

```
num = random.random()
```

The following code uses random to generate a random number between 0.0 and 1.0 and modifies the number to generate an integer between 1 and 6 inclusive to simulate rolling a die. This process can be used for any range of values.

```
import random

for line in range(10):
    print(int(random.random() * 6 + 1), end=',')
```

```
6,2,4,5,2,1,2,2,6,4,
```

The randint function simplifies simulation of rolling a die as shown here. Recall that the second argument is the limit and is not included.

```
num = random.randint(1, 7)
```

The ***uniform*** function allows setting a range for random floating point numbers as shown here.

```
num = random.uniform(1.0, 7.5)
```

Chapter 6 Review Questions

1. Separating a program into distinct sections is referred to as _______________.
2. A section of code that contains a group of associated statements that perform a specific task is referred to as a ________.
3. To execute a function, it must be ____________.
4. The two types of functions are _________ functions and _____________ functions.
5. The code within a function is called the function ___________.
6. The first line of a function that contains the name and parameter list for the function is called the function __________.
7. Indentation in the program forms _________ of code.
8. The area of a program where a variable is accessible is referred to as the variable's _________.
9. A variable declared inside a function is referred to as a _______ variable.
10. A variable that is declared outside all functions is referred to as a _______ variable.
11. A variable that should not be changed by the program and is named with all uppercase letters with words separated by underscores is referred to as a ______________.
12. Technically speaking, a value passed to a function is called an ___________.
13. Technically speaking, a value received by a function is called a __________.
14. A value-returning function must have a __________ statement.
15. A term used to describe multiple programmers working together on the same program is ______________.
16. An IPO document contains brief descriptions of the _____________, ______________, and ______________ of a program or function.
17. The statement used to access the functions in a module, is the __________ statement.
18. The module available in Python that contains functions for square root and others is the _______ module.
19. The module in Python that provides random number functions is called the ___________ module.

Chapter 6 Short Answer Exercises

1. Write a statement that calls the following function.

```
def show_output( ):
    print('Hello from my function')
```

2. What does the following statement declare?

```
EARTH_DIAMETER = 3963
```

3. Write a statement that calls the following function and passes it the phrase 'Hello World'.

```
def show_output(phrase ):
    print(phrase)
```

4. What do the following lines of code output when the program is executed?

```
def main( ):
    smallest(6, 3)


def smallest(first, second):
    if first < second:
        print('first is smaller)
    else:
        print('second is smaller')

main( )
```

5. Write a statement that calls the following function and stores the return value in a variable named num.

```
def get_value( ):
    val = int(input('Enter an integer '))
    return val
```

6. What data type does the following function return and what will it return if it is called and the number 5 is passed to it?

```
def is_even(num):
    even = False
    if num % 2 == 0:
        even = True
    return even
```

7. What do the following lines of code output?

```
num = math.sqrt(math.sqrt(16))
print(num)
```

8. Write a statement that will assign the variable num a random integer between 1 and 100.

9. Write a statement using randrange that will assign the variable num a random number between 1 and 100 inclusive.

10. Write a statement that will assign the variable num a number between 1 and 100 that is a multiple of 5 (5, 10, 15, 20, etc.).

11. Write a statement that will assign the variable num with a random number between 0.0 and 1.0.

12. What does the following loop produce?

```
for x in range(10):
    print(int(random.random( ) * 10 + 1), end=',')
```

Chapter 6 Programming Exercises

1. Write a function called display_num that obtains a number from the user and prints ‘You entered‘, and the number that was entered.

2. Write a function called get_input that obtains a number from the user and returns the number that was entered.

3. Write a program that calls a function average, passes it three arguments, and the function returns the average of the numbers. Print the average that is returned from the function from main. Write an IPO for the function.

4. Write a program with three (3) functions. The first function will obtain the radius of a circle from the user, the second function will compute and return the circumference of the circle, and the third function will display 'The circumference of the circle is ' with the result. The equation for circumference is shown here.

$$C = 2\pi r$$

5. Modify the circle program in #4 to locate the functions in a separate module, and import the module into the main file.

6. Write a program with four (4) functions located in a separate module. The program will prompt the user for the two side lengths of a rectangle and validate the input (must be > 0). The first function called will compute and return the area, the second function will compute and return the perimeter, the third function will compute and return the diagonal, and the fourth function will display the output as shown in the sample below. Use the Pythagorean Theorem for the diagonal and import math in the module.

```
Enter the length of side 1 3
Enter the length of side 2 4
The area is : 12.0
The perimeter is : 14.0
The diagonal is : 5.0
```

7. Write a sales program with five (5) functions located in a separate module. The first function will obtain and return the price of an item being purchased, the second function will obtain and return the quantity of the items being purchased, the third function will compute and return the total price for the items, the fourth function will compute and return the tax amount at 7% (0.07) for the purchase, and the fifth function will display all of the information as shown (output alignment is not critical).

```
Enter the price of the item: 12.34
Enter the number of items: 2

Price      $12.34
Quantity     2
Subtotal   $24.68
Sales tax   $1.73
----------------
Total Sale $26.41
```

8. Write a program that displays 10 random numbers between 1 and 20 inclusive. No functions are required.

9. Write a program that displays 10 odd random integers between 1 and 100 inclusive separated by colons. Consider how to display only odd values. No functions are required.

10. Write a program that displays random number variation graphically using asterisks. The program will generate 20 random integers between 1 and 20 inclusive, and display that many asterisks as a row. A six row sample is shown below.

```
*******************
*****
***************
*
**************
*******************
```

11. When four random numbers are generated between 1 and 6 inclusive, the probability is high that at least one 6 will be produced. Write a program that calls a function that produces four (4) random numbers and returns true if a six was produced and false if not. If true is returned output “A six” otherwise output “No six”. Call the function 20 times in the program. Are there more sixes than expected?

12. Write a program uses the last random number generated as the upper limit for the next random number. The program will generate a random number between 0 and 100 inclusive, then it will use that number as the upper limit for another random number between 0 and that number, then use that number as the upper limit , and so on... Output the random number each time and end the program when the number generated is zero. Run the program 10 times. On average, how many numbers are displayed?

Chapter 6 Programming Challenge

Meteor Evacuation Status Simulation

Design and develop a program that determines the evacuation status for a city based upon the size and distance of a meteor coming toward the city. The program will accept a meteor size in meters and a distance from the city in miles, and

compute and display the meteor data and evacuation status. Allow the user to enter another set of data without restarting the program.

Required six (6) functions located in a separate module:

- obtain, validate (must be > 0.0), and return the user input of the meteor size in meters
- obtain, validate (must be > 0.0), and return the distance of the meteor in miles
- compute and return the meteor's speed (120 mph * size)
- compute and return the time to impact (distance/speed) in minutes
- determine and return the evacuation status for the city based on the criteria below
- display the data as shown below

Evacuation Status Criteria:

If the time to impact < 45 minutes, then Evacuation CANNOT BE COMPLETED
If time to impact > 45 and <= 90 minutes, then Evacuation is POSSIBLE
If the meteor time to impact is > 90, then Evacuation is PROBABLE

Note that speed is in mph, but time to impact is in minutes.

```
Enter the meteor size in meters: 3
Enter the meteor distance in miles: 400

        Meteor Data:

        Diameter in meters:      3.0
        Distance in miles:       400.00
        Speed in mph:            360.00
        Minutes to impact:       66.67

        Evacuation Status: Evacuation is POSSIBLE

Run again?  enter "Y"
```

Chapter 7

File Operations and Dialogs

The applications and information or data used by computers is stored in files. Recall that the data stored in RAM does not persist between runs of the program, or when the computer is turned off. Files allow information to be stored until it is needed, changed when required, and deleted when no longer needed. All files have what is referred to as a ***file extension***. This is the three or four letters that follow the period in the file name. File extensions are used by most operating systems to associate the file with an application. When a file is double-clicked, the operating system determines the application to launch based upon the file's extension and the application that was used to open that type of file previously. Double-clicking a file named "song.mp3" will launch an audio player because the audio player application has been associated with the mp3 file extension. The example below has a "txt" file extension which is typical for text files which are usually opened with Notepad or Notes by the computer's operating system.

file name

some_data_file.txt

file extension

File Names and Extensions

Different file types are usually opened by different applications, although some applications like Notepad can open a variety of file types. For example, the .py files created for Python programs can be opened with Notepad and viewed as text. Table 7.1 lists some common file extensions.

Extension	Description
.docx	Microsoft Word document file
.exe	executable file
.html	web page file
.java	Java source code file
.jpg	JPEG image file
.mov	movie file
.mp3	audio file
.pdf	Adobe Portable Document File
.py	Python source code file
.zip	ZIP compressed archive

Table 7.1 - Common File Extensions

Files being read from are typically referred to as ***input files***, and files being written to as ***output files***. There are three steps to using a file in a computer program:

- the file is opened
- the file is processed (either written to or read from)
- the file is closed

Opening a File

When a file is opened using Python, it is associated with the program through a ***file object*** that has a variable reference. The variable reference is the name to be

associated with the file in the program. This is not that different from when an integer or float is defined except that the name is associated with a file object. The general format for opening a file is shown here.

```
variable_reference = open(filename, mode)
```

The ***open*** function is passed two arguments. The first is the actual name of the file (in quotes if a literal string), and the second argument is the ***mode*** in which the file will be opened. The mode determines the way that the file will be opened, and what will occur if the file exists or if it does not.

Mode	Description
'r'	Opens a file for reading, produces an error if the file does not exist
'w'	Opens a file for writing. If the file exists, the contents is erased. If the file does not exist, it creates the file.
'a'	Opens a file for appending. Creates the file if it does not exist.

Table 7.2 - File Opening Modes

When the file name is used as the first argument, the program will search the ***default directory*** for the file. The default directory (folder) is where the program is running. The statement below would search the default directory for the file "some_data.txt" and if it finds it, it will open the file in "read" mode, and associate it with the variable reference my_file.

```
my_file = open('some_data.txt', 'r')
```

If the file is in another directory or sub-directory, or it is to be created in another directory, the full path is included and is preceded by the letter "*r*" without quotes. The "*r*" tells Python to treat the backslashes in the path literally as backslashes and not as the escape sequence. A path begins with the drive letter, colon, backslash, and the directories and sub-directories to the file. It is common

today to use a dialog window to obtain file names, or to accomplish saving a file. This is easier for users and eliminates the chance for typographical errors.

```
myFile = open(r 'C:\Users\csimber\some_data.txt', 'r')
```

As mentioned, the variable reference is a name for a file object, and file objects have methods that simplify some file handling processes. Once a file object is associated with a variable, the variable name is used to access the methods. The only time that the actual file name is used is when the file is being opened.

Writing to a File

When writing to a file, the ***write()*** method is used and is passed what is to be written. The variable reference assigned to the file is followed by the dot operator, and the method name. The following example opens a file for writing and assigns the file object to my_file. Then a phrase is written to the file, and the last line closes the file using the ***close()*** method. Note that the close method is not passed any arguments.

Ex. 7.1 – writing a string to a file

```
my_file = open('some_data_file.txt', 'w')
my_file.write('A stitch in time saves nine.')
my_file.close()
```

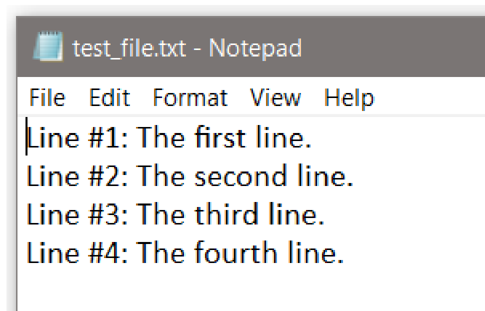
Notice in the example that the only time that the actual file name is used is when it is associated with the variable using the open function. After that, the variable reference for the file object is used. Closing the file ensures that no data is lost. Data being written to a file is queued in a ***buffer*** (a holding area in memory) for efficiency. Closing the file deliberately in the program forces anything being held in the buffer to be written to the file before it is closed. If the program did not close the file, the operating system would eventually close it, but would not check the buffer first.

Writing to a file requires some consideration. The *write()* method will do as it is told, and if the data is to be written on separate lines, then line feeds need to be incorporated into the write statement. The escape sequence '\n' is the ***newline*** character and is used to produce a line feed in the file. The next example opens a

file named "test_file.txt", and associates it with out_file, writes three phrases on separate lines in the file, and then closes the file. The complete program is shown.

Ex. 7.1 – writing on separate lines in a file

```
def main():
    out_file = open('test_file.txt', 'w')

    out_file.write('Line #1: The first line.\n')
    out_file.write('Line #2: The second line.\n')
    out_file.write('Line #3: The third line.\n')
    out_file.write('Line #4: The fourth line.')

    out_file.close()

main( )
```



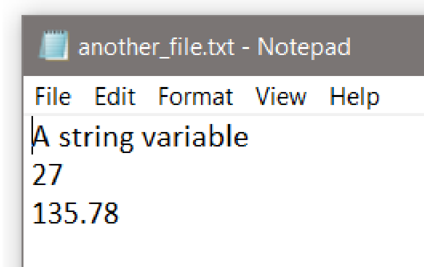
File Writing and Line Feeds

Writing the contents of a variable to a file is handled much like the print function, and to add a line feed, the newline character is concatenated onto a string variable. If the value is not a string, the *str()* function must be used to convert it to a string. Numeric values cannot be written to files as numeric values in Python and must be converted to strings.

The example below defines a string, integer, and float, and then opens a file named "another_file.txt" in write mode and associates it with out_file. The string is then written to the file with a line feed, and the next line writes the integer which is converted to a string that has a line feed concatenated onto it. The next line converts the floating point number to a string for writing, and then the file is closed.

Ex. 7.2 – writing numeric values to a file using str

```
def main():

    my_string = 'A string variable'
    my_int = 27
    my_float = 135.78

    out_file = open('another_file.txt', 'w')
    out_file.write(my_string + '\n')
    out_file.write(str(my_int) + '\n')
    out_file.write(str(my_float))

    out_file.close()

main( )
```



File Writing Numeric Values

If a numeric value is not converted to a string for writing, an error will occur.

```
    out_file.write(intVar)
TypeError: write() argument must be str, not int
>>>
```

Appending to a File

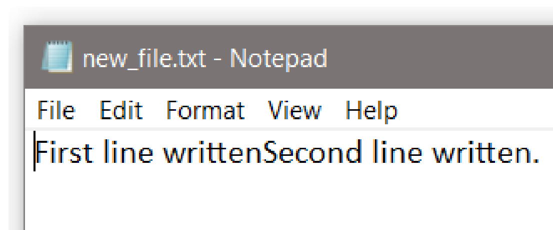
Opening an existing file in write mode erases any data that had been stored in the file. What actually takes place is that the old file is deleted, and a new empty file is created. To ***append*** data to existing data in a file, the file is opened in append mode using 'a' and any existing data in the file is preserved.

The next example opens a file in write mode, writes a string, and closes the file. It then re-opens the file in append mode and writes another string to the file and closes the file. Note that the use of two different modes requires that the file be closed before re-opening in the new mode.

Ex. 7.3 – appending data to a file

```
out_file = open('new_file.txt', 'w')
out_file.write('First line written')
out_file.close()

out_file = open('new_file.txt', 'a')
out_file.write('Second line written.')
out_file.close()
```

In the output file below, a line feed was not written. The first write statement did not include a newline character, and when the file was opened a second time to append information, it was appended to where the last write command ended.



Appending Data to a File

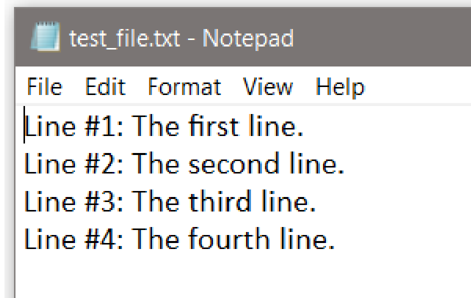
Reading from a File

To read from a file, it is opened using 'r' as the mode. There are file object methods for reading from a file including ***read()*** which returns the entire file contents as a string, and ***readline()*** which will read one line from the file (until '\n' is encountered). Since the *read()* method also reads the newline characters, the information read in will include any line feeds.

As an example, the file that was written in Ex. 7.1 (repeated below) can be read into a single string and displayed. The text and line feeds are read from the file and stored in file_data. When file_data is passed to the print function, the line

feeds are included in the output. Later it will be shown that these lines can be split or parsed from the single string.

Ex. 7.4 – reading data from a file into a single string




```
in_file = open('test_file.txt', 'r')
file_data = in_file.read()
in_file.close()
print(file_data)
```

```
Line #1: The first line.
Line #2: The second line.
Line #3: The third line.
Line #4: The fourth line.
```

Reading from a File with read()

The following program reads a single line from the file in Ex. 7.4.

Ex. 7.4A – reading a single line from a file into a string

```
def main():
    in_file = open('test_file.txt', 'r')

    one_line = in_file.readline()

    in_file.close()

    print(one_line)

main( )
```

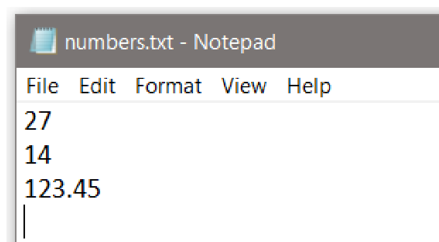
```
Line #1: The first line.
```

Reading Numeric Data

When reading numeric values from a file, they are returned as strings and must be converted to a numeric data type in order to use them as a numeric value. Chapter 3 introduced casting for type conversion and it is used when reading from a file. There are a few circumstances that may arise when doing this depending on how the data was written to the file. Several examples follow.

For the first example, three numbers have been written to a text file called numbers.txt. When they were written, they were converted to strings, and a line feed was concatenated so they are on separate lines in the file. The *readline()* method will be used to read each value, and the values will be cast to numeric values. This is straight forward because readline stops at the line feed.

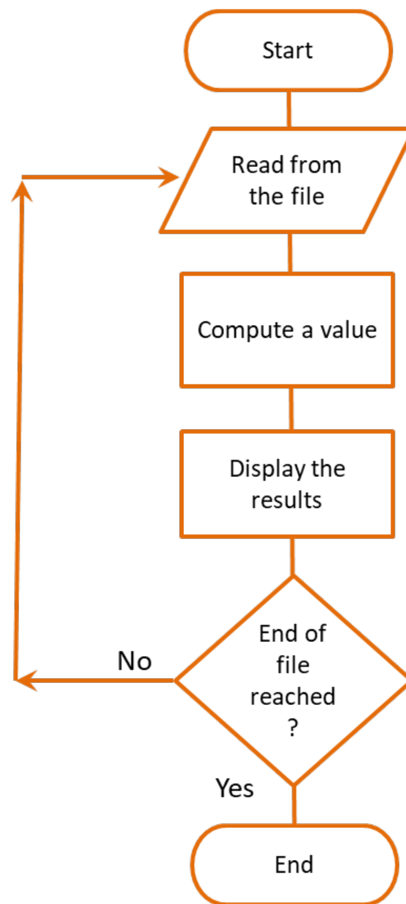
Ex. 7.5 – reading and casting numeric data from a file




```
in_file = open('numbers.txt', 'r')

num1 = int(in_file.readline())
num2 = int(in_file.readline())
num3 = float(in_file.readline())

num4 = num1 + num2 + num3
print(num4)
```

```
164.45
>>>
```

In Ex. 7.5, the lines were read one at a time and the returned data was stored into three separate variables. Typically, a loop would be used to read a line or value, process the data, and output some result. The file is read until there are no more values. Every file contains an end of file (EOF) marker that indicates where the file ends. When it is reached, a value cannot be read by the Python method being used. This ends the loop that is reading from the file. A flowchart follows.



File Reading Flowchart

The code for a loop that reads and prints data from a file is show here.

```
input_file = open('dataFile.txt', 'r')

for line in input_file:
    print(line)
```

Next, consider a file that was written with columnar data with tabs between the values. This is referred to as tab-delimited data. A ***delimiter*** is a character used to mark the beginning or end of an item of data. When a delimiter is present, using *read()* would include the delimiter (tabs in this case) in the returned string. Using *readline()* would also include any delimiter within a line including spaces between items. It is common to read files one line at a time in a loop and process

the data, and there are several methods in Python that help to convert the data to a useful format.

Removing Newline Characters

When writing data to a file, a tab '\t' or newline '\n' is often added to separate data or data sets. When Python reads from a file, the data is returned as a string and may include tabs, line feeds, and spaces. To remove them, there are several methods including ***rstrip*** which will remove white space (\n, \t, and space) from the right side of the string. Table 7.3 lists common string modification methods.

Method	Description
lower()	returns a lower case copy of the string
lstrip()	returns a copy of the string with leading white space removed
lstrip(*char*)	returns a copy of the string with leading instances of *char* removed
rstrip()	returns a copy of the string with trailing white space removed
rstrip(*char*)	returns a copy of the string with trailing instances of *char* removed
strip()	returns a copy of the string with all leading and trailing white space characters removed
strip(*char*)	returns a copy of the string with all leading and trailing instances of *char* removed
upper()	returns an upper case copy of the string

Table 7.3 - String Modification Methods

The string modifications are used to convert what has been read into a usable format and ensure that white space characters are not part of any data being converted to a numeric value.

In addition, there is a ***split()*** method that can split (parse) a line of data using a delimiter. The default delimiter is any white space, but another could be used. As an example, a data file contains the phrase "She sells sea shells by the

seashore" on two lines. The entire phrase (both lines) is read into a string and the *split()* method is used to extract each word from the phrase in a loop.

Ex. 7.6 – reading text from a file into a string and parsing the data

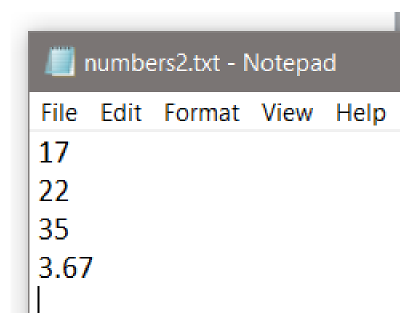
```
def main():

    inFile = open('Sea_shells.txt', 'r')
    phrase = inFile.read()

    for word in phrase.split():
        print(word)

main( )
```

```
She
sells
sea
shells
by
the
seashore
```

String Parsing

When the data is numeric, the algorithm (solution) is similar except that each item is read individually using a for-in loop and it must be cast before it can be used as a number. It could also be read all at once using *read()*, parsed using *split()*, and cast as needed. Both examples will be shown.

This next example reads a data file named numbers2.txt containing the numbers shown on separate lines, and sums the values. When the data is read using a for-in loop, it is read as a string and must be cast in order to be used as a number. The file contents are shown below.



Ex. 7.7 – reading numbers from a file in a loop

The for-in loop reads each item in the file one at a time, assigns the item to num, and num is cast to a float for use with addition.

```
total = 0.0

inFile = open('numbers2.txt', 'r')

for num in inFile:
    total = total + float(num)
print(total)
```

```
77.67
>>>
```

Example 7.7A reads the entire file into a string and parses the individual items using *split()* and casts them.

Ex. 7.7A – reading numbers from a file with read and parsing them

```
def main():

    total = 0.0

    inFile = open('numbers2.txt', 'r')
    numbers = inFile.read()

    for num in numbers.split():
        total = total + float(num)
    print(total)

main( )
```

The technique used for reading and handling data from a file is often dependent upon the task required. The data can be read one item or line at a time, or the entire contents can be read at once. Loops are typically used when reading one item or line at a time into a variable, and it is not always necessary to read the items into a variable first before handling them.

The next example reads numeric values directly from a file and processes them. A data file has been created called numbers3.txt containing 222, 444, 555, and 1221. The numbers will be read within the for-in loop expression, processed, and displayed until the end of the file is reached.

Ex. 7.8 – reading numbers from a file while processing them

```
def main():

    inFile = open('numbers3.txt', 'r')

    for number in inFile:
        print(int(number) * 3.5)

    inFile.close()

main( )
```

```
777.0
1554.0
1942.5
4273.5
```

This is a simpler algorithm and does not require additional methods. Notice in the output that the numbers are displayed as floating point numbers. The ***promotion*** occurs since they are being multiplied by 3.5 (a float).

Exceptions

An exception is a type of error that occurs when a program is running. Exceptions are raised or thrown and must be handled or the program will terminate. When a file cannot be created or cannot be opened, or when there is a data type mismatch, an exception will occur. The format for an exception handler in Python is the ***try/except*** statement, and there are several variations that will be covered. The general format is shown below.

```
try:
    statement1
    statement2
    etc.

except ExceptionName:
    statement1
    statement2
    etc.
```

The ***try*** block is entered and if a statement raises an exception, the handler immediately following the ***except*** clause that matches the type of exception raised executes and the program continues.

Ex. 7.9 – file-not-found exception handler

This program could be read as "try to open the file, and if an error occurs print 'No file exists'. The exception name is ***IOError*** which is the type of exception that would be raised if the file did not exist or could not be opened. Once an exception is raised, the try block is exited and any statements following the one that raised the exception will not be executed.

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')
        print('If an exception is raised, ', end='')
        print('this line will not be displayed')

    except IOError:
        print('No file exists.')

    inFile.close()

main()
```

Each type of exception that could be raised should have an exception handler for that specific exception. An exception that is not handled will halt the program, but an exception clause that does not list a specific exception, will handle any exception that is raised in the try suite. This could be considered a default handler as shown below.

Ex. 7.10 – handling multiple exception types

The two anticipated exceptions are the file error and a type error. Any other exceptions would be handled by the exception handler with no exception name.

```
try:
    input_file = open('missingFile.txt', 'r')
    for line in input_file:
        val = int(line)
        sum = sum + val

except IOError:
    print('No file exists.')

except ValueError:
    print('A bad value was read')

except:
    print('Other Error in program.')
```

An exception raised is actually an object and contains information about the error. If it were displayed, it would show the same message that would be seen in the Traceback error message. The contents can be accessed by assigning the exception to a variable.

```
except ValueError as e:
    print(e)
```

The try-except expressions can include an ***else*** clause (or else suite) which will execute only if no exceptions were raised. If an exception is raised, then the else clause is skipped. It can be thought of as "try to execute these, and if an exception is raised, execute the exception handler, *otherwise* execute these".

There is a ***finally*** clause (or finally suite) which executes regardless of whether an exception was raised or not to perform cleanup. If a try suite opens a file and then executes other statements, one of those other statements may throw an exception. But the file is still open. A finally suite allows closing the file, or any other cleanup needed whether an exception was raised or not.

Ex. 7.11 – the finally suite and closing an input file

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')
        for line in input_file:
            val = int(line)

    except IOError:
        print('No File exists.')

    except ValueError:
        print('A bad value was read.')

    finally:
        inFile.close()

main()
```

This section covered common exceptions that could be thrown when handling files. Other exceptions can be reviewed at Python.org. The Traceback error message will describe the error during testing so that a handler can be written, or a generic handler could be used that displays

the contents. The goal is to anticipate and handle as many exceptions as possible. Exceptions that are not handled will terminate the program.

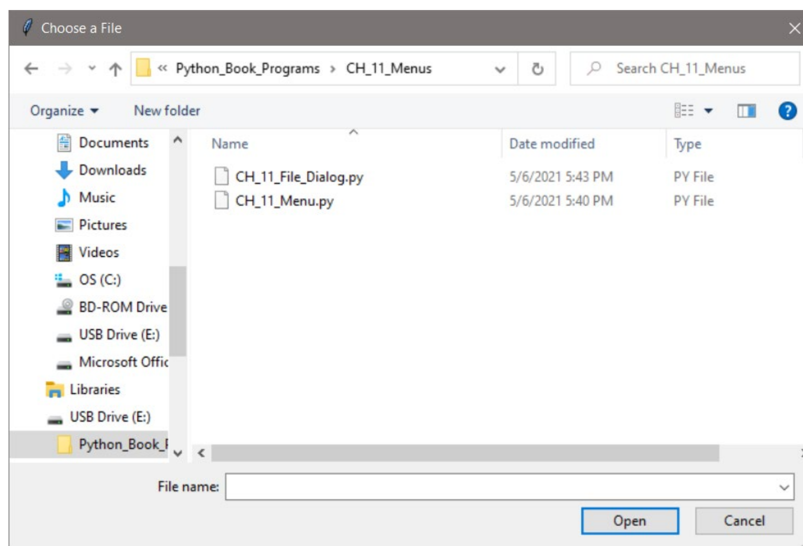
File Open Dialog

The tkinter module provides dialogs for handling files. Using them requires the specific import statement shown below.

```
import tkinter
from tkinter import filedialog
```

Ex. 7.12 – Open File Dialog – Python 3.0+

When the dialog appears, the default directory is the directory where the program is running.



Open File Dialog

The dialog returns a string containing the full path to the file and the name of the file. The string is used to open the file as shown below.

```
filename = filedialog.askopenfilename(title='Choose a file.')
infile = open(filename,'r')
```

Chapter 7 Review Questions

1. The characters that follow the name of a file are referred to as the file ______________.
2. A file opened for reading is referred to as an _________ file.
3. A file opened for writing is referred to as an __________ file.
4. In order to use a file in a program, the file must be __________.
5. When a file is opened in a program, it is associated with a ____________ that has a variable reference.
6. A file opened by a program is opened in a specific _________ such reading, writing, or appending.
7. For file handling, the directory where the program is running is referred to as the __________ directory.
8. The _______ method is used to output data to a file.
9. An area in memory where data to be written is temporarily stored is referred to as a __________.
10. When a program is finished using a file, it should _________ the file.
11. The escape sequence or newline character is ________.
12. To write a numeric value to a file, it must first be converted to a __________ using the _______ function.
13. When a data file is opened for writing using ‘w’, any data existing in the file will be __________.
14. Adding to a string or combining two strings is referred to as ___________.
15. Adding to the end of a file’s contents is referred to as ______________ to the file.
16. When numeric data is read from a file, it must be ______ before using it in an equation.
17. A ____________ is a character used to mark the beginning or end of an item of data.
18. The ___________ method is used to strip off trailing white space characters.
19. The ___________ method is used to parse items of data from a string.
20. When an error occurs because a file cannot be found for reading or created for writing, it is referred to as raising an ______________.

Chapter 7 Short Answer Exercises

1. Write the statements required to open a file named “test_file.txt” for writing and associate it with the variable reference my_file, write “This is a test.” to the file, and close the file.

2. Write the statements required to open a file named “test_file.txt” for writing and associate it with the variable reference my_file, write “This is a test.” to the file, and close the file. Then reopen the file for reading, read a line of text, print the line of text, and close the file.

3. Write the statements required to open a file named “numbers.txt”, write the numbers 1, 2, and 3 to the file, and close the file.

4. Write the statement to open a file named “numbers.txt” for writing that does not erase the existing data in the file. Associate the file with the name my_file.

5. Write the statements required to open a file named “data.txt” for reading in a try block, and the exception handler for an IOError that displays “The file could not be opened”.

6. Write the statements required for a try block to read all of the data from a file named “data.txt” and print the contents, and handle an exception.

Chapter 7 Programming Exercises

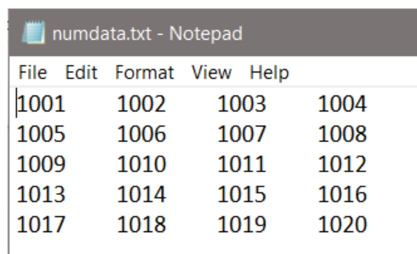
1. Write a program that creates a file for writing called “data.txt” and write the following lines to the file on separate lines and close the file. Open the file for reading and display the contents of the file.

    The first line
    The second line
    The third line
    The fourth line
    The fifth line
    The sixth line

2. Write a program that reads the “data.txt” file created from #1 above and display the contents of the file with a line number and colon as shown below.

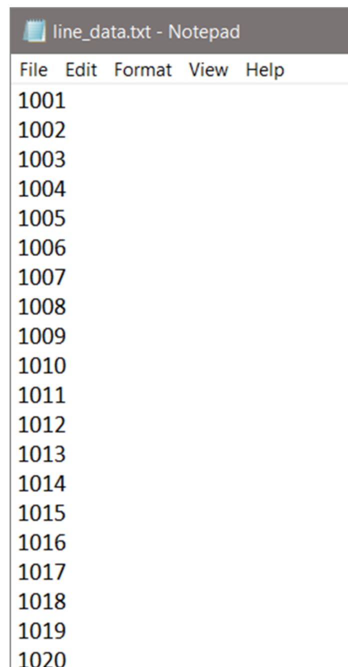
```
1: The first line
2: The second line
3: The third line
4: The fourth line
5: The fifth line
6: The sixth line
```

3. Write a program that creates a text file named “num_data.txt” and writes the numbers 1001 thru 1020, separated by a tab with four numbers per line.



4. Write a program that creates a text file named “line_data.txt” and writes the numbers 1001 thru 1020, on separate lines. Then open the file and display the numbers in two columns separated by a tab as shown below. Hint: use strip() to remove the line feed.

Data file example:



Display output example:

```
1001	1002
1003	1004
1005	1006
1007	1008
1009	1010
1011	1012
1013	1014
1015	1016
1017	1018
1019	1020
```

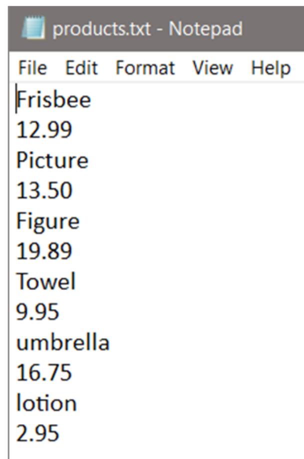
5. Create a text file named "sales_data.txt" with the sales data listed below each on a separate line. Write a program that defines main which reads one value from the file at a time and calls a function to compute and return the discount price (20% off), and display the original and discount prices in two (2) columns separated by a tab as shown.

Sales data: 19.64, 3.56, 9.87, 16.33, 12.95, 6.50

```
$ 19.64 $ 15.71
$  3.56 $  2.85
$  9.87 $  7.90
$ 16.33 $ 13.06
$ 12.95 $ 10.36
$  6.50 $  5.20
```

6. Create a text file named "products.txt" with the product names and prices listed below them on separate lines. Write a program that reads the data from the file and calls a function to compute and return the discount price for the item (20% off), and display the item name, original price, and sale price in columns with column headers as shown.

Data file example: Display output example:


products.txt - Notepad
File Edit Format View Help
Frisbee
12.99
Picture
13.50
Figure
19.89
Towel
9.95
umbrella
16.75
lotion
2.95

```
Product        Price        Sale Price
--------------------------------------
Frisbee      $ 12.99        $ 10.39
Picture      $ 13.50        $ 10.80
Figure       $ 19.89        $ 15.91
Towel        $  9.95        $  7.96
umbrella     $ 16.75        $ 13.40
lotion       $  2.95        $  2.36
```

7. Create a text file named "some_data.txt" with the following single line in the file.

1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, sales

Write a program that opens and reads the file, displays the numbers vertically, totals the numbers, and displays the total. Include a try block and exception handling for an IOError and a ValueError that display appropriate messages. Include a finally clause that closes the file.

Chapter 7 Programming Challenges

#1 Employee Data File

Design and develop a program for a local company payroll that uses the employee data file information shown below. Create the data file shown below left and write a program that will read the file and display the name and ID for the employee, and the gross pay for each employee based upon the input file data. The format for the output is shown below. Include two (2) exception handlers in the solution.

The date format for the input file is: name, ID number, hourly rate, and hours worked.

Data set for the "employee_data.txt" file is:

```
Erica
#824
11.35
32
Tamar
#926
12.80
18
Simone
#765
14.55
12
Darius
#960
16.75
26
Sheila
#923
27.25
22
```

Output Format

```
Employee     ID#    Gross Pay
-----------------------------
Erica        #824    $363.20

Tamar        #926    $230.40

Simone       #765    $174.60

Darius       #960    $435.50

Sheila       #923    $599.50
```

#2 Employee Data File - Dialog

Modify the program in Programming Challenge #1 to use a File Open dialog to obtain the name of the file.

Chapter 8

Strings, Lists, Dictionaries, and Sets

Strings are used extensively in computer programs and Python provides many ways to examine and manipulate strings including the ability to examine the individual characters in a string. Consider a program that validates a password to ensure that it contains specific characters. Each character of the password needs to be visited and checked to determine if it meets one of the requirements. To access the individual characters of a string, the ***for-in*** loop walks the string one character at a time while placing a copy of the character in a variable that can be used in statements within the loop. This example assigns a string to temp and the loop prints each character in the variable.

```
temp = 'something'

for char in temp:
    print(char)
```

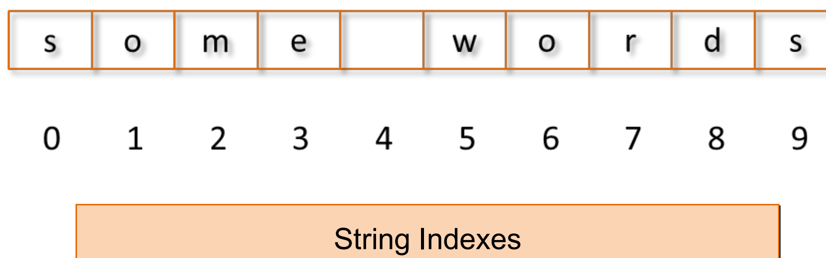
```
s
o
m
e
t
h
i
n
g
```

As each letter is copied into the variable, it can be examined or manipulated, and since the letter in the variable is a copy, any changes made to it do not affect the original string. This example substitutes the letter "s" for "t" in the output without changing the original string.

```
temp = 'the'

for char in temp:
    if char == 't':
        char = 's'
        print(char, end='')
    else:
        print(char, end='')

print('\n' + temp)
```

```
she
the
```

String characters can also be accessed using the ***index*** of the character. The index is the position in the string beginning at zero. Note that the string is ten (10) characters long (including the space), yet the indexes are 0 through 9.



To access the character using the index, the index is placed in square brackets. The format is shown below.

```
my_string[index]
```

Ex. 8.1 – indexing strings

```
a_string = 'something'
print('Index zero is ', a_string[0])
```

```
Index zero is  s
```

```
a_string = 'something'
print (a_string[0], a_string[3], a_string[7])
```

```
s e n
```

In Python, negative indexes can be used to access character positions relative to the last character in the string. The index -1 is the last character in the string, and negative numbers work backward from there.

Ex. 8.2 – negative string indexes

```
b_string = 'negative'
print (b_string[-1], b_string[-4], b_string[-6])
```

```
e t g
```

The index can also be used to obtain a copy of a single character from a string.

Ex. 8.3 – copying a character from a string

```
c_string = 'copy'
ch = c_string[2]
print('character is ', ch)
```

```
character is  p
```

If an index is used that is out of range, an ***IndexError*** exception will be thrown. The ***len()*** function, which returns the length of the string, can be used as a way of controlling loops when accessing string characters to prevent errors. Note in the example below that index is initialized to zero and incremented in the loop. The length of the string controls the loop, and the space between the two words is included in the output.

```
temp = 'theater tickets'
index = 0

while index < len(temp):
    print(temp[index], end='')
    index = index + 1
```

```
theater tickets
```

Recall that strings in Python are immutable, and cannot be changed once created. The '+' operator is used to concatenate strings which actually creates a new string and assigns it to the variable name for the original string. The original string can no longer be used because there is no longer a variable referencing it. Eventually, the Python interpreter will remove the original string from memory.

Ex. 8.4 – concatenating strings

```
city_string = 'New'
city_string = city_string + ' York'
```

When the second statement in Ex. 8.4 executes, a new string is created and Python assigns the variable name to the new string. This modification can be accomplished with multiple string variables as well.

```
part1 = 'New'
part2 = ' York'

part1 = part1 + part2
print(part1)
```

```
New York
```

A third string could also be created by concatenating two others.

```
part1 = 'San'
part2 = ' Diego'

part3 = part1 + part2
print(part3)
```

```
San Diego
```

String Slicing

String ***slicing*** is used to select a portion of a string using optional start, end, and step specifiers. The general format allows one, two, or three specifiers. When the first specifier is omitted, Python uses zero as the start and the specifier as the end which is not included in the slice.

```
my_string[:end]
```

When two specifiers are used, the first is the start index and the second specifier indexes the end of the slice and is not included in the slice.

```
my_string[start:end]
```

When three specifiers are used, the third is the step in the sequence.

```
my_string[start:end:step]
```

Ex. 8.5 – slice expressions with strings.

```
sequence = '123456789'

first_four = sequence[:4]
print(first_four, end='')
print()

second_four = sequence[5:9]
print(second_four, end='')
print()

every_other = sequence[0:9:2]
print(every_other, end='')
```

```
1234
6789
13579
```

In and Not In

Searching for content in strings can be handled using the ***in*** and ***not in*** operators. The example would search for and find the word “time” in the string.

Ex. 8.6 – searching for content in strings.

```
phrase = 'A stitch in time saves nine.'
search_word = 'time'

if search_word in phrase:
    print('Found it.')
else:
    print('Not found.')
```

```
Found it.
```

The logic also works in reverse using "not in".

```
phrase = 'A stitch in time saves nine.'
search_word = 'time'

if search_word not in phrase:
    print('Not found.')
else:
    print('Found it.')
```

```
Found it.
```

The next example searches for the character '6' in the string. Note that a single character can be a string. It is just a string that is one character long.

```
sequence = '123456789'

if '6' in sequence:
    print('Found a six.')
```

```
Found a six.
```

String Testing and Modification

The string testing methods return true or false, and test each character in the string. If the string is not at least one character long, the result is false.

Method	Description
isalnum()	True if the string contains only alphabetic letters or digits
isalpha()	True if the string contains only alphabetic letters
isdigit()	True if the string contains only numeric digits
islower()	True if the string contains only lower case alphabetic letters
isspace()	True if the string contains only white space characters
isupper()	True if the string contains only uppercase alphabetic letters

Table 8.1 - String Testing Methods

The string modification methods (ref Table 7.3) include conversion to upper and lower case, and various strip methods: lower(), upper(), lstrip(), rstrip(), and strip(*char*).

The ***search*** and ***replace*** methods include: endswith(*substring*), find(*substring*), replace(*old, new*), and startswith(*substring*), and return true or false.

Lists

Lists in Python are sequences of data that are ***mutable***, dynamic, and can be indexed and sliced. They can hold different types of data, and can be accessed in the same way as strings. Initializing a list with values is accomplished using the assignment operator and enclosing the members of the list in brackets.

```
numbers = [5, 15, 25, 35]          # numbers
words = ['the', 'and', 'why']      # strings
mixed = ['first', 105, 15.6]       # strings and numbers
```

There are several ways to access the elements in a list. As an example, the first statement in the example below assigns a list of numbers to num_list. Notice in the output that the first print statement displays the list surrounded by square brackets. The second set of statements use a for-in loop to access each element in the list, and the last statement accesses the list element using an index. There are a variety of ways to access list elements.

Ex. 8.7 – numeric lists

```
num_list = [5, 15, 25, 35]
print(num_list)

for n in num_list:
    print(n, end=' ')

print()
print(num_list[2])
```

```
[5, 15, 25, 35]
5 15 25 35
25
```

The len() function works with lists, and can be used to control a loop. In the next example, the loop counter index is incremented to control the loop, and is also used as the index for accessing the list elements.

Ex. 8.8 – string lists and the len() function

```
word_list = ['one', 'two', 'three']
index = 0
while index < len(word_list):
    print(word_list[index])
    index = index + 1
```

```
one
two
three
```

There are also built in functions for lists to add elements, insert elements, remove elements, change the order of the list, and to find the minimum and maximum values in a list.

To ***append*** an item to the end of a list, the statement includes the name of the list, the dot operator, the append function, and the element to be added in parentheses.

```
num_list = [5, 15, 25, 35]
num_list.append(45)
print(num_list)
```

```
[5, 15, 25, 35, 45]
```

To ***insert*** an item into a list, the statement includes the name of the list, the dot operator, the insert function, the index where the element is to be inserted, and the element to be inserted.

```
num_list = [5, 15, 25, 35]
num_list.insert(2, 45)
print(num_list)
```

```
[5, 15, 45, 25, 35]
```

To ***remove*** an item from a list, it must be in the list or an exception is raised. The format includes the name of the list, the dot operator, the remove function, and

the element to be removed in parentheses. If there are elements in the list beyond the element being removed, they are shifted toward the front of the list.

```
num_list = [5, 15, 25, 35]
num_list.remove(25)
print(num_list)
```

```
[5, 15, 35]
```

To ***reverse*** the order of a list, the statement includes the name of the list, the dot operator, and the reverse function.

```
num_list = [5, 15, 25, 35]
num_list.reverse()
print(num_list)
```

```
[35, 25, 15, 5]
```

To ***sort*** a list, the statement includes the name of the list, the dot operator and the sort function.

```
cities = ['Boston', 'Caldon', 'Albany']
cities.sort()
print(cities)
```

```
['Albany', 'Boston', 'Caldon']
```

To find the minimum or maximum value in a list, the list is passed to the ***min*** and ***max*** functions.

```
numbers = [15, 3, 106, 27]
print(min(numbers))
print(max(numbers))
```

```
3
106
```

Elements in a list can be changed using the index of the element. There is an ***index()*** function that can be used to determine the index for a specific element, but it will raise an exception if the element is not in the list. To determine first if the item is in the list, the '*in*' operator can be used. Once it has been determined

that the element is in the list, the index function can be used to obtain the position in the list for changing the element.

```
numbers = [1, 2, 3, 4, 5]

if 3 in numbers:
    pos = numbers.index(3)
    numbers[pos] = 99

print(numbers)
```

```
[1, 2, 99, 4, 5]
```

Lists can be ***concatenated*** using the '+' operator to combine two lists. In this example two lists are combined and then sorted to maintain the order.

```
list1 = ['a', 'c', 'e', 'g']
list2 = ['b', 'd', 'f', 'h']

list1 = list1 + list2
print(list1)
list1.sort()
print(list1)
```

```
['a', 'c', 'e', 'g', 'b', 'd', 'f', 'h']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Lists can be copied, but not using the assignment operator. Assigning one list to another would simply have both list names reference the same list.

```
new_list = old_list     # referencing the same list
```

To ***copy*** a list, an empty list is defined and each element in the first list is appended to the new list. It could also be accomplished by concatenating the old list onto the new empty list.

```
old_list = [12, 22, 32]
new_list = []

for element in old_list:
    new_list.append(element)
```

Split

The ***Split*** method by default uses the space as a separator and returns a list of items in the string. A different separator can be specified including "/" when a date is being parsed. In this example, a string is split into a list.

Ex. 8.9 – split a string without a specified separator

```
time_string = 'hour minute second'
time_list = my_string.split()
print(time_list[1])
```

```
minute
```

Ex. 8.9A– split a string with a specified separator

```
time_string = '10:23:59'
time_list = time_string.split(':')
print(time_list[1])
```

```
23
```

Lists can be passed to functions and functions can return lists. In the next example, the list *num_list* is passed to *get_sum* which returns a sum of the numbers in the list.

Ex. 8.10 – passing a list to a function

```
def main():
    num_list = [5, 15, 25, 35]
    print('The sum is :', get_sum(num_list))


def get_sum(in_list):
    vals = 0
    for num in in_list:
        vals = vals + num

    return vals

main()
```

```
The sum is : 80
```

Ex. 8.11 – returning a list from a function

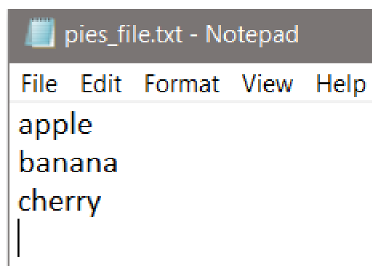
```
def main():
    my_list = get_list()

    print('The list is :', my_list)


def get_list():
    new_list = [1, 2, 3, 4, 5]

    return new_list

main()
```

```
The list is : [1, 2, 3, 4, 5]
```

Lists can be written to files with ***writelines(list_name)***, but there are no line feeds with this method. To include line feeds, a loop is needed and the newline character needs to be added. A tab or a space could be added the same way and used as a delimiter when reading.

Ex. 8.12 – writing a list to a file

```
def main():
    pies = ['apple', 'banana', 'cherry']

    out_file = open('pies_file.txt', 'w')

    for pie_type in pies:
        out_file.write(pie_type + '\n')

    out_file.close()

main()
```



When reading from a file to populate a list, the newline character '\n' will be included when each line is read. The newline character can be removed in a loop by visiting each index in the list and stripping it off. The example below prints the list before and after the newline character has been removed.

Ex. 8.12 – reading into a list from a file and removing '\n'

```
def main():
    input_file = open('pies_file.txt', 'r')

    pie_list = input_file.readlines()

    input_file.close()

    print(pie_list)

    count = 0
    while count < len(pie_list):
        pie_list[count] = pie_list[count].rstrip('\n')
        count = count + 1

    print(pie_list)

main()
```

```
['apple\n', 'banana\n', 'cherry\n']
['apple', 'banana', 'cherry']
```

Ex. 8.12A – reading into a list from a file and removing '\n' with rstrip

```
def main():
    input_file = open('pies_file.txt', 'r')

    pie_list = []

    for line in input_file:
        pie_list.append(line.rstrip('\n'))

    input_file.close()

    print(pie_list)

main()
```

```
['apple', 'banana', 'cherry']
```

Two-dimensional Lists

A list can have lists as elements and are considered two-dimensional as in having rows and columns. When a two-dimensional list is indexed there are two dimensions to consider and both begin at zero.

values[0][0]	values[0][1]	values[0][2]
values[1][0]	values[1][1]	values[1][2]
values[2][0]	values[2][1]	values[2][2]
values[3][0]	values[3][1]	values[3][2]

For a two-dimensional list, a nested loop is used to access the entire list. As an example, consider a list of names, ID numbers, and hourly pay rates. The cell indexes are shown for clarity.

'Amir' [0][0]	'Conner' [0][1]	'Darla' [0][2]
'ID 112' [1][0]	'ID 204' [1][1]	'ID 157' [1][2]
'15.75' [2][0]	'18.50' [2][1]	'28.30' [2][2]

The list is initialized using sets of square brackets with a comma between sets, and a set of square brackets surround the lists. The nested (inner) loop accesses each row for the column designated in the outer loop. Once each row has been accessed, the outer loop accesses the next column, and so on.

```
ROWS = 3
COLS = 3

emp_list = [['Amir', 'Conner', 'Darla'],
            ['ID 112', 'ID 204', 'ID 157'],
            ['15.75', '18.50', '28.30']]


for c in range(COLS):
    for r in range(ROWS):
        print(emp_list[r][c], end='\t')

    print()
```

```
Amir    ID 112  15.75
Conner  ID 204  18.50
Darla   ID 157  28.30
```

Tuples

A tuple is simply a list that is immutable and cannot be changed. Tuples process faster and since the data cannot be changed, a tuple protects the data. Tuples also support all list operations and built-in functions except those that modify lists including: append, remove, insert, reverse and sort. To modify a tuple, it can be converted to a list, and then back to a tuple.

```
my_tuple = tuple(my_list)        # convert list to tuple

my_list2 = list(my_tuple)        # convert tuple to list
```

Plotting List Data (matplotlib)

There is a Python package for plotting called ***matplotlib*** that enables plotting line, bar, histogram, scatterplots, pie charts, and more using list data in an auto-scaling resizable window. The package is not part of the Python standard library, and must be installed separately using the Python ***pip*** installer. This is covered in Appendix C. Once matplotlib is installed, the module ***pyplot*** from the package is imported similar to the way that the math package is imported, but note the module name, dot, and package. Typically the module is imported "as" a shortened name to lessen the amount of typing each time it is accessed. Here it is imported as "plt".

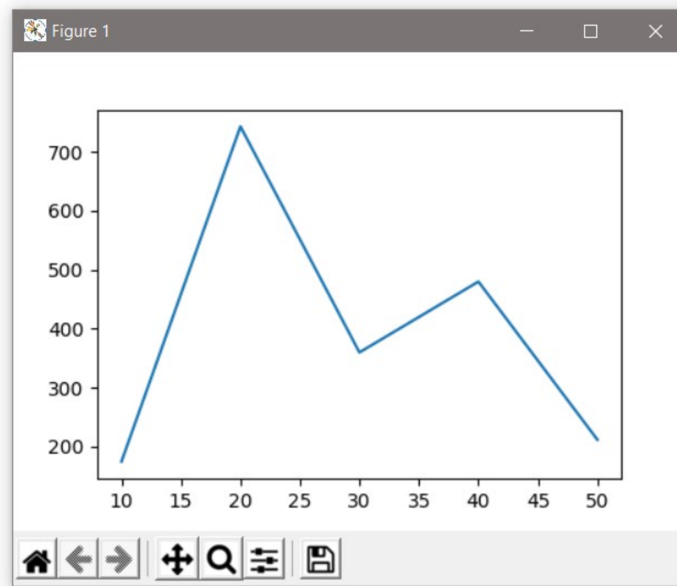
```
import matplotlib.pyplot as plt
```

There are many features and functions available in pyplot beyond those explained in this section. There are also other packages and modules that have been developed for use with Python that provide additional features and capabilities. The goal of this section is to introduce basic plotting functionality using the pyplot module in matplotlib.

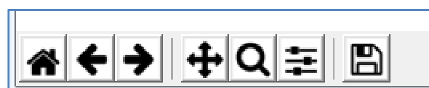
The example shown below first establishes the number of data points for each axis using the *x_coords* and *y_coords* lists. The number of x-axis tick marks is established from the number of coordinates in the list and is augmented with interim five values. The number of y-axis tick marks and their values are established by the values in the *y_coords* list. The call to ***plt.plot*** actually builds the graph in memory, and it is then displayed when ***plt.show()*** is called.

Ex. 8.13 – simple plot example

```
import matplotlib.pyplot as plt

x_coords = [10, 20, 30, 40, 50]

y_coords = [175, 743, 360, 480, 212]

plt.plot(x_coords, y_coords)
plt.show()
```



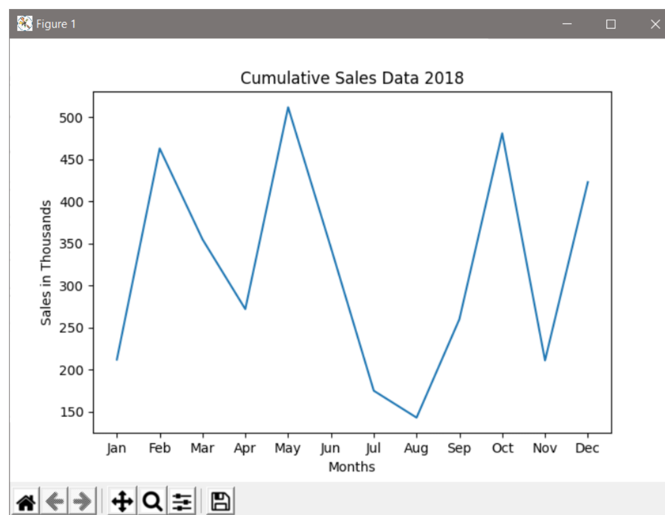
The data is plotted and the window has features that are automatically added in the lower left-hand corner including zooming in a rectangular shape, saving the image, and others.



There are many options available for customizing the charts including: axis labels, tick marks, data markers, the width of bars for bar charts, and slice labels for pie charts. The next example uses a function which is passed a list of sales data, and customizes the chart with x-tic mark labels, axis labels, and a title for the chart. Note that the x coordinates use a list that is also needed to position the x-tic labels. The y coordinates are the sales amounts from the sales list. The result is a more informative plot.

Ex. 8.14 – line graph of sales data with customization

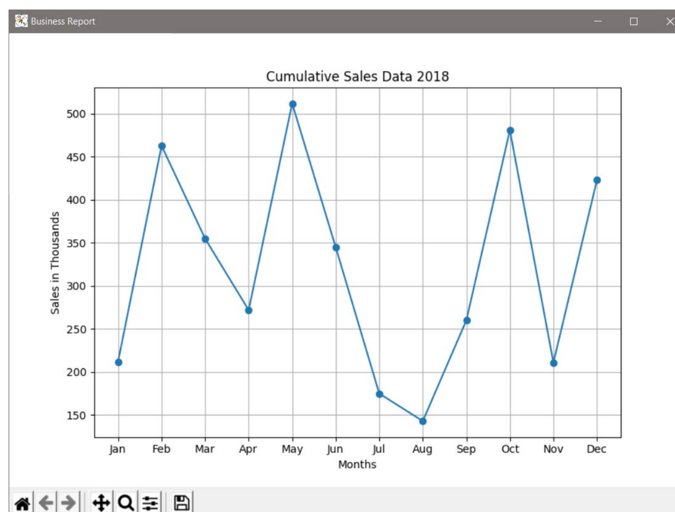
```
def main():

    sales = [212, 463, 355, 272, 512, 345, \
                175, 143, 260, 481, 211, 423]

    plot_list(sales)


def plot_list(sales_list):

    x_coords = [0,1,2,3,4,5,6,7,8,9,10,11]
    y_coords = sales_list
    plt.xticks([0,1,2,3,4,5,6,7,8,9,10,11],
               ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])

    plt.title('Cumulative Sales Data 2018')
    plt.ylabel('Sales in Thousands')
    plt.xlabel('Months')

    plt.plot(x_coords, y_coords)

    plt.show()

main()
```



Some additional enhancements include changing the window title on the title bar of the window using *gcf()*, which is "get current figure", and then "canvas set window title". Shapes can be added to the data points using the marker option for the plot function, and adding a background grid to the chart is implemented by setting it to true. These enhancements are included in the next example.

Ex. 8.14 – line graph of sales data enhanced

```
def plot_list(sales_list):
        x_coords = [0,1,2,3,4,5,6,7,8,9,10,11]
        y_coords = sales_list

        plt.xticks([0,1,2,3,4,5,6,7,8,9,10,11],
                   ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                    'Jul',  'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])

        fig = plt.gcf()
        fig.canvas.set_window_title('Business Report')

        plt.title('Cumulative Sales Data 2018')
        plt.ylabel('Sales in Thousands')
        plt.xlabel('Months')

        plt.grid(True)
        plt.plot(x_coords, y_coords,marker='o')
        plt.show()
```



Plotting multiple lines requires two plot functions, and there is a legend option with labeling, and linestyles can be assigned.

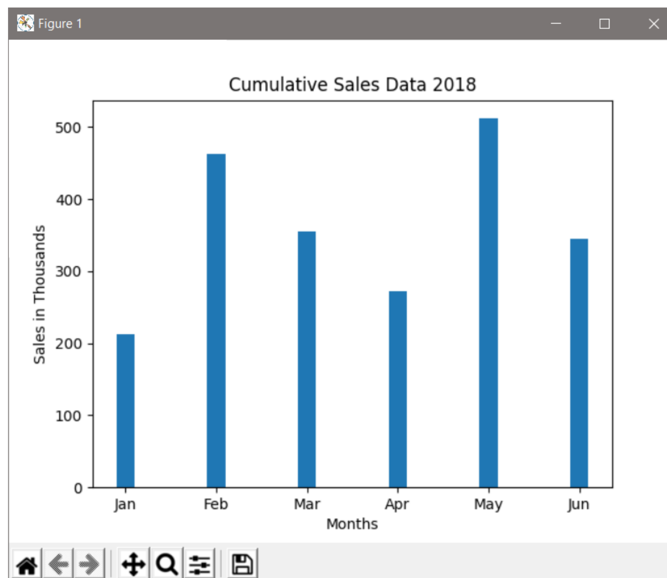
```
line1 = [10, 40, 30, 20, 50]
line2 = [20, 70, 45, 100, 80]

x_coords = [0,1,2,3,4]
plt.xticks([0,1,2,3,4],['One','Two','Three','Four','Five'])

plt.plot(x_coords, line1, label ='line 1', linestyle='-')
plt.plot(x_coords, line2, label = 'line 2', linestyle='--')
plt.legend()
plt.show()
```

There are additional line markers that can be used including 's' for squares, '*' for asterisks, '^' for triangles, and 'D' for diamonds.

Ex. 8.15 – bar chart of sales data

This example uses six months of sales data to create a bar chart. The new items to consider include using the x_coords as the bar's left edge, the y_coords (sales data) as the height for the bar, and including a width for the bars. The x_coords and xticks need to accommodate the width of the bars, so they are increased.

```
def chart_list(sales_list):
        x_coords = [0,10,20,30,40,50]
        y_coords = sales_list
        plt.xticks([0,10,20,30,40,50],
                   ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'])

        bar_width = 2
        plt.bar(x_coords, y_coords, bar_width)


        plt.title('Cumulative Sales Data 2018')
        plt.ylabel('Sales in Thousands')
        plt.xlabel('Months')

        plt.show()

main()
```

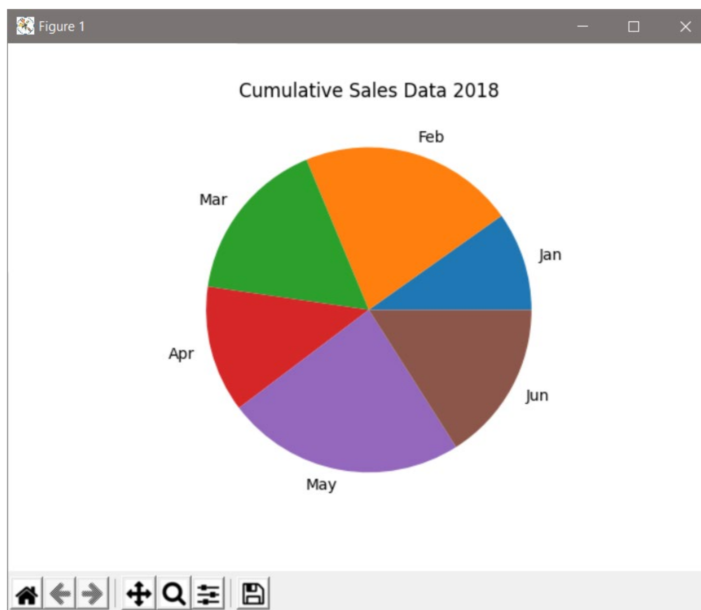


Additional features and customizations are available for bar charts including changing the color for the bars, and can be found at various websites.

Ex. 8.16 – pie chart of sales data

This example uses six months of sales data to create a pie chart. The new items to consider include the slice labels, but there are fewer items needed to draw a simple pie chart.

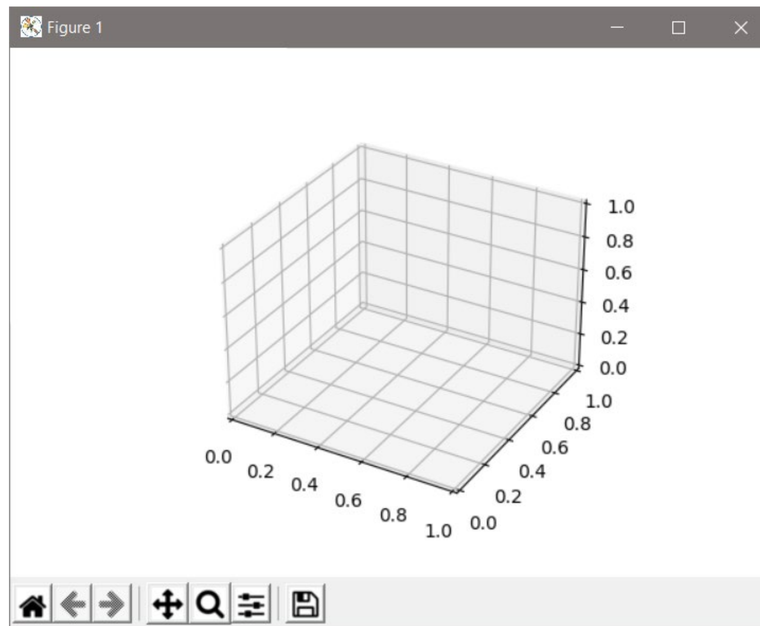
```
def main():

    sales = [212, 463, 355, 272, 512, 345]

    pie_list(sales)


def pie_list(sales_list):

        slice_labels = ['Jan', 'Feb', 'Mar',
                        'Apr', 'May', 'Jun']

        plt.pie(sales_list, labels = slice_labels)

        plt.title('Cumulative Sales Data 2018')

        plt.show()

main()
```



A pie chart can also be customized with different colors, 3D affects, segmented slices, and other features.

Pyplot also provides for creating 3D plots which can also be customized. A simple example is provided below.

```
fig = plt.figure(figsize=(4,4))
ax = fig.add_subplot(111, projection='3d')

plt.show()
```



Modules like pyplot are easy to use and provide extensive functionality. A tutorial and additional information for pyplot is available at the matplotlib.org website.

Dictionaries

Organizing and storing data is often required when implementing solutions. Containers that store and manage data are referred to as data structures which can be used to implement ***collections***. Collections are objects that store other objects as elements. A list is an example of a collection. Others include a dictionary which stores elements as key/value pairs, and sets which contain no duplicates. There are benefits and limitations with each collection type that should be considered when using them in a solution.

A ***dictionary*** is an associative array container with a key and a value associated with the key. Consider a data set in a file consisting of student ID numbers and student names (partial set shown below). A program that needed to access and manage the data would need to read the data and store it. Earlier in the chapter, examples using a string or list were used for similar data, but assume that names need to be found quickly by searching for the ID number. A dictionary could store the ID number as the key, and the name would be the associated value for the key. Python provides many dictionary operations that simplify the process.

Key (Student ID)	Value (Student Name)
10310	Allison Knox
11298	Amir Cumber
10452	Cody Garfield
12034	Layna Camron

A dictionary can be created by assigning key/value pairs to a dictionary name as shown below, but collections are more often used for large amounts of data.

```
students = {10310:'Alison Knox', 11298:'Amir Cumber',...}
```

Typically, a dictionary is created by declaring an empty dictionary and then adding key/value pairs.

```
dictionary_name = {}

dictionary_name[key] = value
```

The first statement below declares a dictionary named students, and the second adds the first student to the dictionary. The key in brackets is the student ID and the value is the name of the student.

```
students = {}

students[10310] = 'Allison Knox'
```

To access a value from a dictionary requires the use of a key. If the key searched for does not exist in the dictionary, an error will occur. Testing beforehand is

required to ensure that the key exists. The next examples will use the student dictionary populated here.

```
students = {}

students[10310] = 'Allison Knox'
students[11298] = 'Amir Cumber'
students[10452] = 'Cody Garfield'
students[12034] = 'Layna Camron'
```

Ex. 8.17 – testing for a key in a dictionary

```
ID = int(input('Enter the student ID: '))

if ID in students:
    print(students[ID])
else:
    print('That ID is not valid')
```

```
Enter the student ID: 11298
Amir Cumber
```

When assigning a value to a key in a dictionary, if the key exists, the value will be changed. There cannot be duplicate keys in a dictionary. If the key does not exist, the key/value pair will be added to the dictionary.

To display the contents of a dictionary, the name of the dictionary can be passed to the print function, but note the output format.

```
print(students)
```

```
{10310: 'Allison Knox', 11298: 'Amir Cumber', 10452: 'Cody Garfield',
12034: 'Layna Camron'}
```

To delete a key/value pair, the ***del*** statement is used with the dictionary name and the key. Again, if the key does not exist, there will be an error.

```
if ID in students:
    del students[ID]
else:
    print('That ID is not valid')
```

To determine the number of key/value pairs that are contained in a dictionary, the *len()* function can be used, and to iterate over the keys a dictionary, a *for* loop can be used.

```
print('There are ' + str(len(students)) + ' students')

for stukey in students:
    print(stukey)
```

```
There are 4 students
10310
11298
10452
12034
```

The ***get()*** function is another way to determine if a key exists in a dictionary. It provides for a default value if the key does not exist and does not cause an error. If the key does exist, it returns the value associated with the key.

```
stu_name = students.get(10310, 'Not found')
```

There are some additional dictionary functions that provide additional features and access to keys, values, and both.

Sets

Recall that a ***set*** is a collection that cannot contain duplicates. They provide some set operations that may be familiar like union, intersection, difference, and symmetric difference. Sets are optimized in memory for fast searching, so they may provide a solution when speed is an issue. A set can be declared and populated later or initialized when declared. Note the use of parentheses when declaring an empty set.

```
set_name = set()

set_name = ([element, element, ...])
```

When adding to a set, the ***add*** method is used with the element to be added in parentheses.

```
numset = set([1,2,3,])

numset.add(4)
```

A *for* loop can be used to access the elements in a set.

```
numset = set([1,2,3,])

numset.add(4)

for num in numset:
    print(num, end=':')
```

```
1:2:3:4:
```

There are two ways to remove an element from a set: ***remove*** and ***discard***. The remove method causes an error if the element is not in the set, the discard method does not.

```
numset = set([1,2,3,4])

numset.remove(3)
numset.discard(2)

for num in numset:
    print(num, end=':')
```

```
1:4:
```

To determine if an element exists in a set, the in and not in operators can be used, and the len function returns the number of elements in the set.

```
numset = set([1,2,3,4,5])

print(str(len(numset)))

search_value = 3

if search_value in numset:
    print('Found it')
```

```
5
Found it
```

The methods for set comparisons and relationships provide for determining union, intersection, differences, and symmetric differences, as well as subsets and supersets. The format for each of these and a brief explanation follows.

The ***union*** method returns a set of elements that is the union of both sets - all of the elements that appear in the sets without duplicates. The "|" operator (referred to as a pipe) can also be used.

```
set1.union(set2)
set1 | set2
```

The ***intersection*** method returns a set of elements that appear in both sets. The "&" operator (ampersand) can also be used.

```
set1.intersection(set2)
set1 & set2
```

The ***difference*** method returns a set of elements that appear in set1 but do not appear in set2. The subtraction operator "-" can also be used.

```
set1.difference(set2)
set1 - set2
```

The ***symmetric difference*** method returns a set of elements that do not appear in both sets. The "^" operator (caret symbol) can also be used.

```
set1.symetric_difference(set2)
set1 ^ set2
```

The ***issubset*** method returns a Boolean value - True if set2 is a subset of set1, and False otherwise. The relational operators provide the same functionality.

```
set2.issubsset(set1)
set2 <= set1
```

The ***issuperset*** method returns a Boolean value – True if set1 is a superset of set2, and False otherwise. The relational operators provide the same functionality.

```
set1.issuperset(set2)
set1 >= set2
```

Chapter 8 Review Questions

1. The _________ in a string can be accessed individually using an index.
2. The first position or index of a character in a string is _________.
3. The index [-1] accesses the __________ character in the string.
4. If an index is used that is out of range, a _________ exception will be raised.
5. The term ___________ indicates that strings cannot be changed.
6. Adding one string to another is referred to as ________________.
7. Lists are data sequences that are mutable, meaning that they can be ________.
8. The _______ function is used to determine the size of a list.
9. The append function is used to add an item to the _______ of a list.
10. The ________ of an element in a list is used to access the element.
11. A tuple is a list that cannot be _____________.
12. A dictionary stores elements as ____________ pairs.
13. A set is a collection that cannot contain ____________.

Chapter 8 Short Answer Exercises

1. What do the following lines of code output?

```
a_string = 'some words'
print(a_string[1], end='')
print(a_string[-1])
```

2. What do the following lines of code output?

```
a_string = 'nothing here'
for ch in a_string:
    if ch == 'e':
        print('Found one.')
```

3. What do the following lines of code output?

```
a_string = 'Thank you.'
print(a_string[11])
```

4. What will the string part1 contain after the following lines of code execute?

```
part1 = ‘a good ’
part2 = ‘thing’
part1 = part1 + part2
```

5. What will the string part3 contain after the following lines of code execute?

```
part1 = ‘a good ’
part2 = ‘thing’
part3 = part1 + part2
```

6. What do the following lines of code output?

```
letters = ‘ABCDEFGHIJ’
afew = letters[:3]
print(afew)
```

7. What do the following lines of code output?

```
letters = ‘ABCDEFGHIJ’
afew = letters[1:6:3]
print(afew)
```

8. What do the following lines of code output?

```
find_it = ‘Once’
phrase = ‘once upon a time’
if find_it in phrase:
        print(‘Found.’)
```

9. What do the following lines of code output?

```
phrase = ‘some words’
phrase = phrase.strip(‘s’)
print(phrase)
```

10. What do the following lines of code output?

```
names = [‘Amy’, ‘Beth’, ‘Darrus’]
print(names[1])
```

11. What will be the output after the following lines of code execute?

```
search_num = 2
values = [1, 2, 3, 4]
if search_num in values:
        pos = values.index(search_num)
        values[pos] = 0

print(values)
```

12. What do the following lines of code output?

```
date_string = ‘10/17/22’
date_list = date_string.split(‘/’)
print(date_list[1])
```

13. What do the following lines of code output?

```
numbers = [ 5, 10, 20, 40, 50]
numbers.append(60)
numbers.insert(3, 30)
numbers.remove(5)
print(numbers)
```

14. What do the following lines of code output?

```
pies = {1:‘cherry’, 2:’apple’, 3:’pumpkin’}
print(pies[2])
```

15. What do the following lines of code output?

```
pies = {1:‘cherry’, 2:’apple’, 3:’pumpkin’}
pies[2] = ‘peach’
print(pies[2])
```

16. What elements will the set contain after the following statement?

```
set1 = set(‘Python’)
```

17. What elements will the set contain after the following statement?

```
word = set(‘radar’)
```

18. What elements will be output after the following statements?

```
word1 = set('abc')
word2 = set('def')
print(word1.union(word2)
```

Chapter 8 Programming Exercises

1. Write a program that creates the list numbers below, then adds 7 to the list, sorts the list, and prints the list.

```
numbers = [8, 6, 12, 9, 11, 10]
```

2. Write a program that creates the list cities below and removes 'Albany' after making sure that it is in the list, then print the list.

```
cities = ['Dodge', 'York', 'Albany', 'Moor']
```

3. Write a program that requests five integers from the user and creates a list of the numbers. Then sorts the list, and prints the lowest number, the highest number and the sum of the numbers.

4. Write a program that creates the list shown below. Then insert the number 15 in the proper index to maintain the order of the list (do not use sort). Then remove the number 11, and reverse the order of the list and print the list.

```
numbers = [11, 12, 13, 14, 16, 17, 18]
```

5. Write a program that creates a string called date and assign it "10,10,22". Then split the string into a list and print the list as 10/10/22.

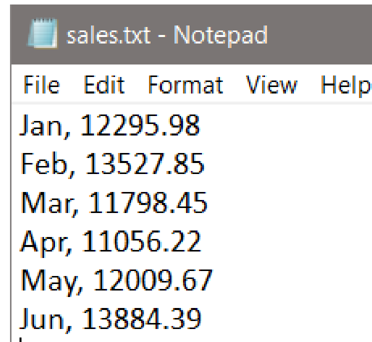
6. Create the list shown below and write the list on separate lines to a file called "pets.txt".

```
pet_list = ['cat', 'dog', 'hamster', 'iguana']
```

7. Write a program that creates the list below in main, and passes it to a function called get_squares that squares the numbers in the list and returns the modified list. Then output the resulting list from main.

```
values = [2, 4, 6, 8, 10]
```

8. Create a file with the data below called “sales.txt”, and write a program that reads the file and create two lists. One list will contain the month names and one list will contain the sales numbers (floats). Plot the data with the month on the X axis and the amount on the Y axis using pyplot.



```
Jan, 12295.98
Feb, 13527.85
Mar, 11798.45
Apr, 11056.22
May, 12009.67
Jun, 13884.39
```

9. Modify program #8 to include “Company Sales Data’ as the chart title, “Thousands” as the y-label, “Months” as the x-label, and diamonds as the marker symbols, and add a grid to the background.

10. Complete Program #8 plotting a pie chart instead of a line chart.

11. Write a program that creates a dictionary called “values” with the key/value pairs below. Then add the pair 6:’six’, replace ‘number’ with ‘three, and print just the values (not the keys).

    1: ‘one’, 2:‘two’, 3:’number’, 4:’four’, 5:’five’

12. Write a program that creates an empty set named “myset” and adds the even numbers from 2 through 20 using a loop, and displays the set.

13. Write a program that creates three empty sets named “even”, “odd”, and “both”. Using a single loop, populate “even” with the even numbers 2 through 20, and populate “odd” with the odd numbers 1 through 19. Assign the union of “even” and “odd” to the set named “both”, and display “both”.

14. Write a program that creates the two sets below, and creates and displays a third set that is the union of the two.

```
set1 = set([1, 2, 3, 4, 5, 6, 7])
set2 = set([5, 6, 7, 8, 9, 10, 11])
```

15. Write a program that creates the two sets below, and creates and displays a third set that is the intersection of the two.

set1 = set([1, 2, 3, 4, 5, 6, 7])

set2 = set([5, 6, 7, 8, 9, 10, 11])

Chapter 8 Programming Challenges

#1 – Decryption

Type or copy/paste this single line of text into a file named "message.txt" and write a program that reads the file and decrypts the text using the key below.

thg:tgllt:tga:thgllt:by:9Thg:tga:thorg:and:makgt:9a:fgw:dollart:To:fggd:hgr:9tga:TurTlgt:from:Thg:thgllt:ThaT:9thg:hat:told.

Key: 't' is 's', ':' is space, 'T' is 't', 'g' is 'e', and '9' is newline

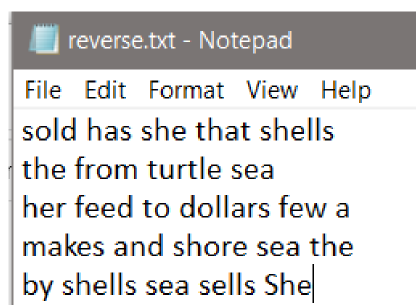
#2 - Reverse lines and Text

Create a file named "reverse.txt" with the lines exactly as shown in the sample file below. Do not include spaces at the ends of the lines.

Write a program that reads the file and uses lists and strings to reverse the order of the lines and to reverse the words in each line. Then display the resulting five (5) lines of text.

Hint: consider a function to split, reverse and print.

reverse.txt:



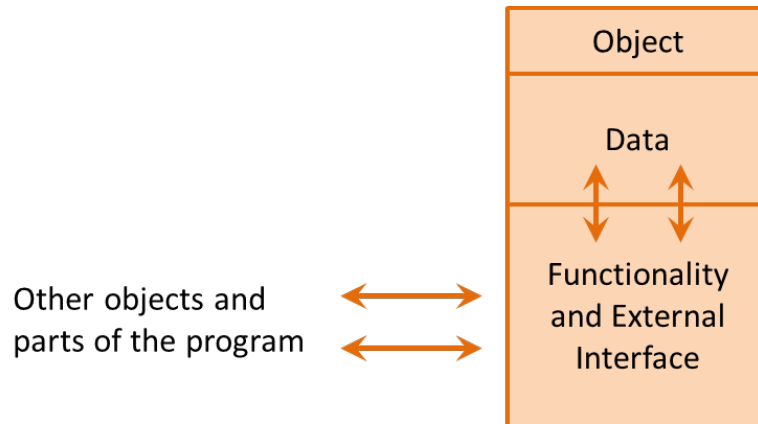
Chapter 9

Classes and Objects

The two approaches to programming in use today include procedural and object-oriented programming. The programs written so far in this text have been procedural, and the order of operations follows a flow of control where a specific task is completed, and then another, and then another, and so on. The functions have been called and in some cases passed data to complete a specific task. The data and the functionality have been separate. In ***Object-oriented Programming*** (OOP), data and functionality are combined in an object and are *hidden* from the rest of the program. This is referred to as ***encapsulation***. The data items stored by an object are referred to as ***attributes*** or members (sometimes member variables), and the functionality within an object is referred to as ***methods*** or behaviors. Object oriented terminology has changed over the decades and different terms are used depending on the programming language. This is unfortunate, but essentially objects have data elements (attributes) and methods (behaviors) that operate on the data elements, and provide an interface for other objects and other parts of the program to operate on them.

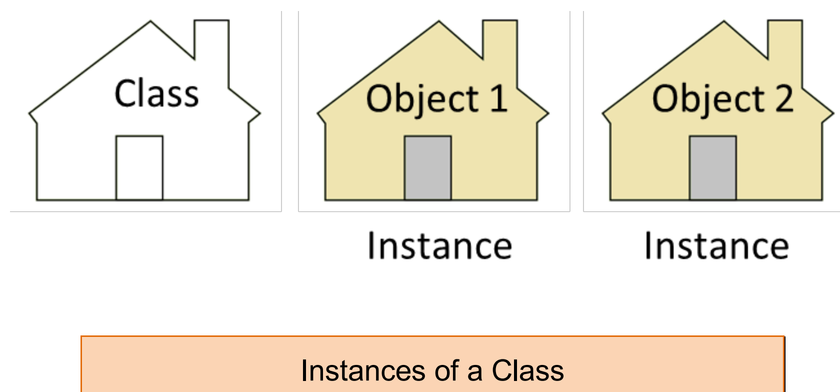
The purpose of hiding the data from outside the object is to protect the data from being corrupted or changed arbitrarily. Parts of a program and other objects can access the object's attributes through a ***public interface*** which provides protection for the data while allowing access to data elements when necessary. This is sometimes referred to as information hiding since programs can use an

object without knowing the inner workings. They interact with an object through the public interface which only requires knowledge of the interface. Any future changes to an object internally do not necessarily require changes to a program that uses that object. Unless the public interface has changed, there is typically no need to modify programs that use the object.



Classes

To create an object, there must be a class. A ***class*** is a framework or blueprint of what the object will contain when it is created. An architect can provide a detailed drawing of a building that shows a door, but the door cannot be opened. A building must be built from the drawing, and then the door of the building can be opened. The building would be an instance of the drawing the same way that an object is an ***instance*** of a class. In addition, multiple buildings could be built from the same drawing and they would all be identical, and they would each have their own door. Multiple objects can be ***instantiated*** from a single class, and they would each have their own set of attributes.



Instances of a Class

The attributes for a class are defined in the ***class definition*** which outlines the data attributes and methods for the class.

The general format for a class definition in Python is shown here.

```
class ClassName:
    def __init__(self, parameter, parameter, ...):

        statement
        statement
        ...
```

Class Definition

The class definition begins with the *class* key word and the name of the class. The naming convention for classes is to begin each word with an uppercase letter. The next line defines the ***constructor*** which looks like a function or method header but includes the word __init__ (short for ***initializer***) which is preceded and followed by two underscores. The constructor executes when a new object of the class is created. Any parameters used by the constructor would be included on this line along with the parameter self which is a reference to the object being created. Any word can be used, but self is typical and considered the standard although root and others are common.

As an example, consider a *Reservation* class with attributes for name, day, time, and number of guests. (The examples use the 24-hour clock)

Ex. 9.1 – Reservation class definition example

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests
```

The class definition begins with the class key word and the name of the class. The next line is the constructor which includes the parameters for self, name, day, time, and guests. When an object of this class is created, the four parameters

for name, day, time, and guests must be passed to the constructor. Recall that the parameter self refers to the object that is being created and is not passed as an argument to the constructor. The next four lines declare and ***initialize*** the instance attributes using the values of the parameters that were passed to the constructor. Example 9.2 creates a *Reservation* object (instance of the class) and passes 'Lake', 'Monday', 1830, and 4 to the constructor. A *Reservation* object is created in main, and assigned to "r1". The print statement accesses the attributes using the name of the object, the dot operator, and the attribute names.

Ex. 9.2 – creating a Reservation object

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests


def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)
    print(r1.name, r1.day, r1.time, r1.guests)

main()
```

```
Lake Monday 1800 4
```

Each *Reservation* object created will have its own set of attributes with its own set of values. The values stored in the instance attributes are referred to as the object's ***state***.

Reservation r1	
name	Lake
day	Monday
time	1830
guests	4

State of an Object

To highlight this, the next example creates two *Reservation* objects (r1 and r2) with different attribute values (states).

Ex. 9.3 – multiple Reservation objects

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests


def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)
    r2 = Reservation('Bethea', 'Tuesday', 1900, 6)
    print(r1.name, r1.day, r1.time, r1.guests)
    print(r2.name, r2.day, r2.time, r2.guests)

main()
```

```
Lake Monday 1830 4
Bethea Tuesday 1900 6
```

The *Reservation* example had a constructor that received parameters to initialize the attributes of the object. Some classes do not receive parameters when an object is created and there are methods within the class for assigning values to the attributes. Also, consider that a *Reservation* may need to be changed. To provide for this capability, a method would be added to the class definition which would allow a change to the state of the object.

Methods

The behavior of an object is specified by writing methods in the class definition. A ***method*** is like a function, but it is inside an object and interacts with the data elements of the object. As an example, the *Reservation* class has been modified below to include a method that allows changing the time attribute of an object. The method is included in the class (note the indentation alignment), and the first parameter is the object on which the method is being called, and it is received as self by the method.

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests

    def change_time(self, time):
        self.time = time
```

After an object of the class is created, the method can be called using the object name, the dot operator, and the method name. Note that only one argument is passed by main to the method in the example below, but that two are received by the method in the class above. The object reference is passed automatically.

Ex. 9.4 – object method call

```
def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)
    r2 = Reservation('Bethea', 'Tuesday', 1900, 6)

    r1.change_time(1800)
    print(r1.name, r1.day, r1.time, r1.guests)

main()
```

```
Lake Monday 1800 4
```

Methods could also be added to enable changing each of the *Reservation* attributes. Or, consider if the constructor for a class did not have parameters and the attributes for the class were all set through methods. Different requirements call for different solutions.

Access Specifiers

Different parts of an object are designated for access using what are referred to as access specifiers. Python does not have an effective way of specifying or enforcing ***public***, protected, and private access like other languages, but a convention uses a single preceding underscore to *indicate* ***protected***, and two

underscores preceding an item to *indicate* that it is ***private***. Since Python uses name mangling (beyond the scope of this text), some protection is afforded private elements (although they are still accessible).

> Public Access Modifier: The members declared as public (which is the default) are deemed accessible from outside an object of the class.
>
> Protected Access Modifier: The members designated as protected (preceded by a single underscore) are deemed accessible only from a class derived from it (in a subclass).
>
> Private Access Modifier: These members are designated (preceded by two underscores) as only accessible from within the class. Access from outside the class is deemed inappropriate.

In the above example, the name, day, time, and guests attributes of a *Reservation* object could be accessed by the main program or other objects. To specify them as private and that they should only be accessed through the methods, two underscores are placed in front of the attribute names. As mentioned before, they are still accessible, but this indicates that they are private and should not be directly accessed. The same two-underscore convention is used with private methods as well. The following example specifies the attributes as private.

Ex. 9.5 – instance attributes as private

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.__name = name
        self.__day = day
        self.__time = time
        self.__guests = guests
```

The *Reservation* class examples included a method to change the time attribute. Methods that change the state (attributes) of an object are referred to as ***mutator*** methods. Methods that access an object's attributes without changing them are referred to as ***accessor*** methods. Mutator methods should have names that indicate what they change and typically begin with the word *set*. Accessor methods should have names that indicate what they obtain and typically begin with the word *get*. For this reason these methods are often referred to as ***setters*** and ***getters***. A complete example is shown later in the chapter.

```
get_some_value()     # accessor

set_some_value()     # mutator
```

Class Attributes

An attribute that is declared outside the __init__ function is a Class attribute and is shared by all objects of the class. They are useful for class constants, tracking data across all instances of the class (similar to static variables), and for defining defaults values. The class attribute can be accessed using the class name or the objects name. As an example, a business program that has an invoice class might have a class attribute for a state tax. The attribute would be shared by all instances (objects) of the class. Another example would be creating a list of the invoices processed. When a new invoice object is created, the constructor adds it the list.

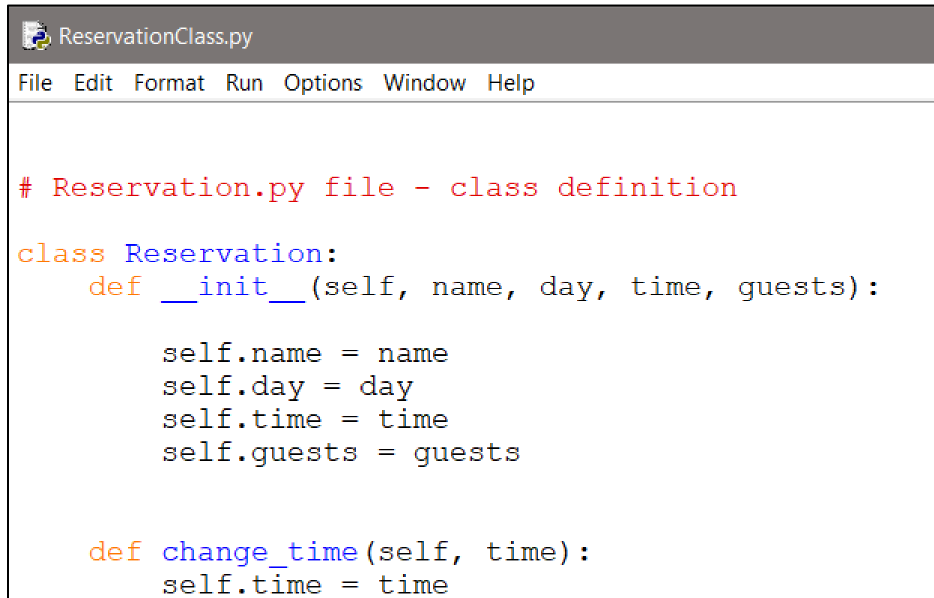
```
class Invoice:

    state_tax = 0.05
    invoice_list = []

    def __init__(self, customer, address):

        self.__customer = customer
        self.__address = address

        self.invoice_list.append(self)
                .
                .
```

Modularization and Class Files

The examples above included a main function and the class definition together in the same file, but class definitions are typically located in separate files. This aligns with the process of ***modularization***, or separating a program into distinct parts. Recall that modularizing a program provides many benefits including the ability to: reuse portions of the code, divide the program development among multiple programmers, and simplify the overall project. Classes should be in separate files as well, and then imported into a program for use.

As an example, the Reservation class program has been modified to place the class in a separate file.

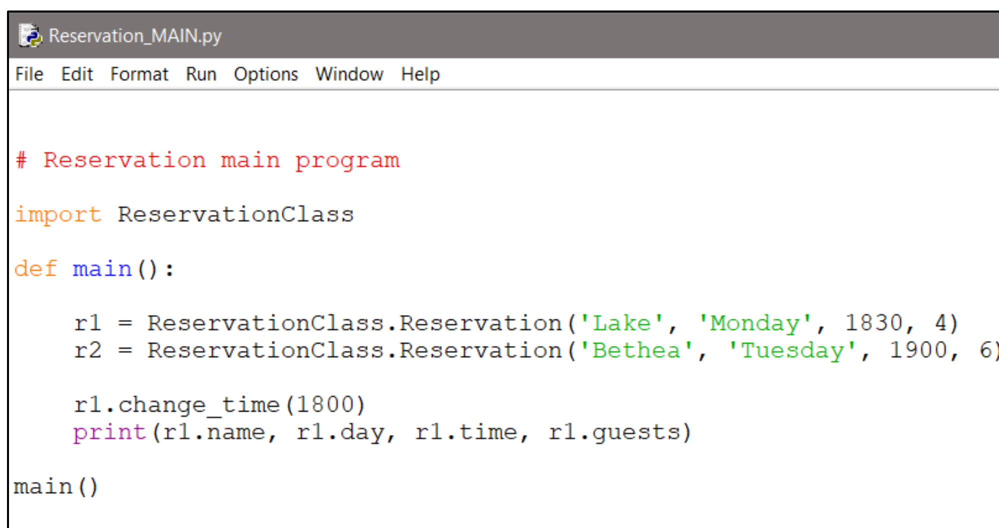
Ex. 9.6 – Reservation program class file example


ReservationClass.py
File Edit Format Run Options Window Help

```
# Reservation.py file - class definition

class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests


    def change_time(self, time):
        self.time = time
```

The main program imports the class file, and the file name precedes the call to the constructor. Note that the call to change the time is preceded only by the object reference.

Ex. 9.6A – Reservation program main program file


Reservation_MAIN.py
File Edit Format Run Options Window Help

```
# Reservation main program

import ReservationClass

def main():

    r1 = ReservationClass.Reservation('Lake', 'Monday', 1830, 4)
    r2 = ReservationClass.Reservation('Bethea', 'Tuesday', 1900, 6)

    r1.change_time(1800)
    print(r1.name, r1.day, r1.time, r1.guests)

main()
```

Displaying the State of an Object

Adding the print statement "*print(r1)*" to the Reservation program might imply that the contents of "r1" would be displayed. But the output would not be the objects state, but a default Python message with the value's type and address in memory where it is stored. The output from the statement is shown below.

```
<ReservationClass.Reservation object at 0x0000022955940040>
```

In order to display an object's state, Python provides the __str__ method (two underscores before and after) which accepts only one parameter (self) and returns a programmer defined string representing the state of the object. The *Reservation* class has been modified to include this method below. Note the use of self in the string that is returned to access the instance attributes.

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests


    def __str__(self):
        return 'The data  is ' + self.name + '-' + \
               str(self.day) + '-' + str(self.time) + \
               '-' + str(self.guests)
```

The __str__ method is not called directly, but is called automatically when the object is passed to the print function.

```
print(r1)
```

```
The data  is Lake-Monday-1830-4
```

Since the string returned by the __str__ method is programmer defined, it can contain as much information about the object's state as needed and in any format. It can be a valuable tool for testing and debugging the program. Another version is shown below.

```
def __str__(self):
    return f'Res: ({self.name),{self.day},{self.time},{self.guests})'
```

Objects as Arguments

When passing objects as arguments to functions and methods, the parameter is a reference to the object. This provides access to the object's methods and attributes by using the name of the object. The example below creates a *Reservation* object and passes it to a function. The function receives the object as a reference in the parameter obj which allows access to the attributes of the object. Note that the name used by the function to receive the object reference can be anything, and is used to access the name attribute of the object by the function.

Ex. 9.7 – objects as arguments (class definition omitted)

```
def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)

    display_res_name(r1)


def display_res_name(obj):
    print('The name is ' + obj.name)

main()
```

```
The name is Lake
```

In Ex. 9.7 above, a separate function is used to show an example of passing an object to a function. The name attribute is accessed directly which is not consistent with OOP. When designing and developing classes, methods should be included that provide the functionality and operations on objects of the class. A program that uses the class should not have to include functionality pertaining to the object. The class should have a method to get the name and return it.

Designing Classes

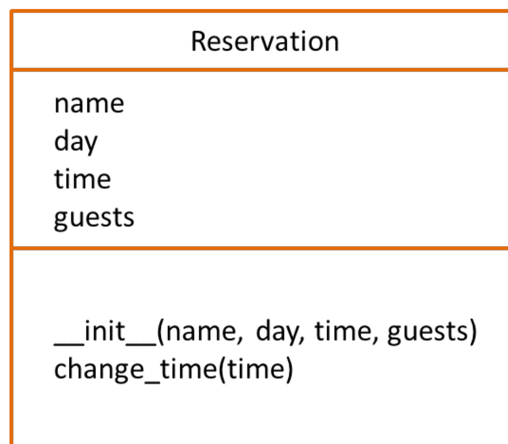
Before writing a class definition, time should be spent designing the class and considering the data attributes and the public interface (methods) needed to access and change the data when appropriate. Most classes model real-world objects and should represent a clear and single ***abstraction***. That is, the class should not include functionality that is outside its responsibilities like getting user input, or anything that is specific to a particular program. One of the goals

of OOP is reuse of the class. Another goal is ***cohesion*** which refers to degree to which a class represents a single abstraction without external dependencies. The degree to which a class depends on another is referred to as ***coupling***. A class should have cohesion (represent a stand-alone entity), and loose coupling (no external dependencies).

One of the tools used in the design of classes is ***CRC cards*** (Class Responsibility and Collaborators) cards. Using an index card, teams brain-storm ideas and note the nouns and verbs used when describing a class. The nouns represent instance variables (data elements) and the verbs represent the methods that will act on the data including the public interface. The goal is to capture all possible data elements and methods, and then refine the lists as the design evolves.

Another tool that is used to design and document classes are ***Unified Modeling Language (UML)*** diagrams which describe a class, the attributes, and methods. The top section contains the name of the class, the next section describes the data attributes, and the bottom section lists the methods for the class. A generalized example for the Reservation class is shown here.

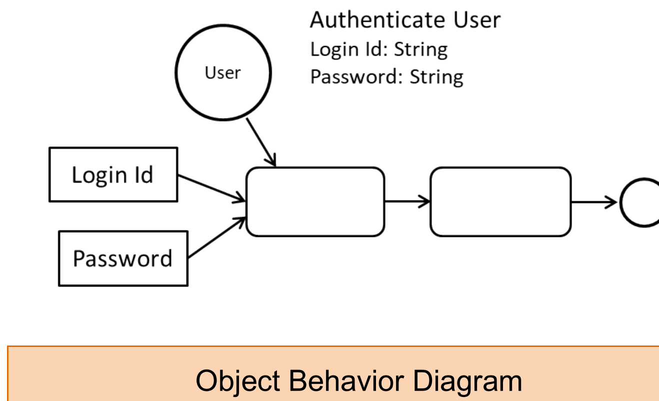
Ex. 9.8 – UML Diagram



UML Diagram

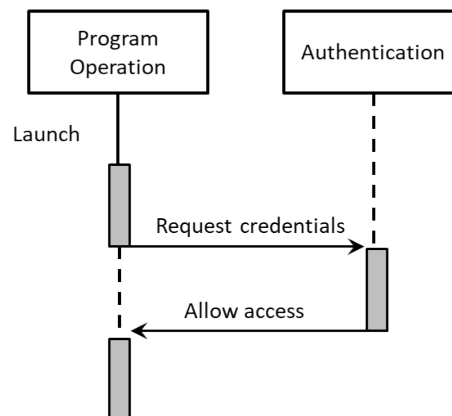
The UML format may differ across industries or organizations, but all capture the data attributes and methods and can also be used in the design process. UML behavior diagrams (often referred to as object activity diagrams) are used to show the flow of control, data, and transactions. UML Superstructure Specifications provide a standard for object interaction depiction. The Behavior

Diagram below illustrates the authentication of user activity with Login Id and Password.



Object Behavior Diagram

The Object Sequence Diagram adds the chronological aspect.



Object Sequence Diagram

The Reservation Class Revisited

The Reservation class in the previous examples had limitations to highlight specific concepts. A more cohesive and complete design would include private data attributes and additional capability, and is worth revisiting. A CRC card could be used to list nouns (things) and verbs (actions).

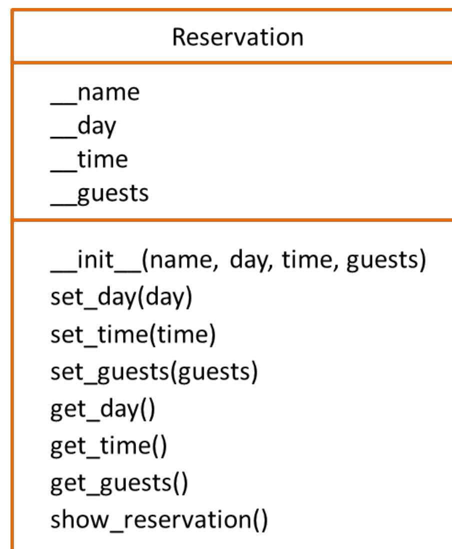
> Nouns (data attributes): name, day, time, and number of guests
>
> Verbs (methods): create a reservation, set/return day, set/return the time, set/return the number of guests, and display the reservation (state)

The Reservation class has been modified to specify the data attributes as private, and to provide mutator and accessor methods for each.

Ex. 9.9 – Reservation Class

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.__name = name
        self.__day = day
        self.__time = time
        self.__guests = guests

    def set_day(self, day):
        self.__day = day

    def set_time(self, time):
        self.__time = time

    def set_guests(self, guests):
        self.__guests = guests

    def get_day(self):
        return self.__day

    def get_time(self):
        return self.__time

    def get_guests(self):
        return self.__guests

    def show_reservation(self):
        print('Name:' + self.__name + '\nDay: ' + str(self.__day) +
              '\nTime: ' + str(self.__time) +
              '\nGuest: ' + str(self.__guests))
```

The UML Diagram for the Reservation Class would be updated as follows:



Earlier an example incorrectly accessed an attibute of an object directly. Now that the Reservation Class is more complete, proper access can be shown using the set and get methods. The example creates an instance of the class, prints the day using the accessor method, changes the time using the mutator method, and displays the state of the object.

Ex. 9.10 – Reservation Class in use

```
def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)

    print(r1.get_day())

    r1.set_time(1900)

    r1.show_reservation()

main()
```

```
Monday
Name:Lake
Day: Monday
Time: 1900
Guest: 4
```

Pickling

The Reservation class examples covered designing and creating the class with access to and changes to attributes, and creating objects. A program that uses the class would create a complete reservation list for the business, and provide additional functionality. The business would want to review the reservations for the day, add new reservations that it can accommodate, and remove reservations when there are cancellations. These operations would be carried out by the program, not the class. When the business closed and the program was ended, the objects would be saved in a file either as text or they could be ***serialized***. Python provides a way to serialize objects called ***pickling*** which converts objects into byte streams (0s and 1s). The *pickle* module's ***dump*** function serializes an object and writes it to a file, and the ***load*** function retrieves an object from a file and de-serializes it. This allows preserving an object's state, and for streaming objects across networks from one server to another.

Using the *Reservation* class, the following program creates a *Reservation* object and serializes it to a file using pickle.dump. It then de-serializes it and calls the display method. Note the file modes used when opening the files. The mode for writing binary is 'wb' and reading binary mode is 'rb'. To highlight the object reference, "r1" and "res" are used.

Ex. 9.11 – Business Class

```
import pickle
import ResClass


def main():

     r1 = ResClass.Reservation('Lake', 'Monday', 1830, 4)

     outfile = open('resData.txt', 'wb')

     pickle.dump(r1, outfile)

     outfile.close()

     infile = open('resData.txt', 'rb')

     res = pickle.load(infile)

     infile.close()

     res.show_reservation()

main()
```

```
Name:Lake
Day: Monday
Time: 1830
Guest: 4
```

A portion of the pickled file is shown below.

resData.txt - Notepad

File Edit Format View Help

€□•Ž   Œ□ResClass”Œ□Reservation”“”)”}”(Œ□_Reservation__name”Œ□Lake”Œ

Ln 1, Col 1 | 100% | Windows (CRLF) | ANSI

Inheritance

Objects are often specialized versions of a general class. For instance, a restaurant is a business, a clothing store is a business, and so is a theater. They have many things in common like employees, sales, and expenses, but some differences or special characteristics as well. The common characteristics could be implemented in a Business class, and then each specific business type could be ***derived*** from that class. The derived classes would inherit the common characteristics from the Business class, and would implement those that are specific to them. In other words, they would ***extend*** the Business Class, and would have what is called an "is a" relationship. This relationship is established through ***inheritance***. The specialized classes (derived classes or subclasses) inherit the characteristics of the general class (base or super class).

In the diagram, the Business Class is the base class, and the specific businesses are derived classes or subclasses. As an example, suppose that a program is needed that can be used by a company with different businesses. The common characteristics for all businesses would be implemented in the Business Class, and the derived classes would contain their specific items and methods. Since all of the companies have a name, and employees, these could be implemented in the base class. Not all of the businesses have an inventory, so that would not be included. The *get_name()* method is included for an example that follows.

Ex. 9.12 – Business Class

```
class Business:
    def __init__(self, name, employees):

        self.__name = name
        self.__employees = employees

    def get_name(self):
        return self.__name
```

Consider a Clothing Store class and a Theater class implementation. The Clothing Store would have inventory, but can inherit the items from the Business class (name and employees). There is no need to include the items that are in the Business class if the Clothing Store Class is derived from the Business Class. This is implemented by defining the Clothing Store Class as inheriting from or extending the Business Class.

```
class ClothingStore(Business):
```

In the example below, the class is defined as a derived class of Business. The initializer for the Clothing Store class calls the __init__ function for the Business class, and passes the three parameters including self that the Business class initializer requires. It then assigns inventory to its' own inventory data attribute. The *get_inventory()* method is included for an example that follows.

```
class ClothingStore(Business):

    def __init__(self, name, employees, inventory):

        Business.__init__(self, name, employees)

        self.__inventory = inventory

    def get_inventory(self):
        return self.__inventory
```

A Clothing Store object would inherit all of the attributes in the Business Class including the methods. A simple example follows that creates an instance of the Clothing Store and calls the *get_name()* and *get_inventory()* methods. Note in the example that "b1" is a *ClothingStore* object and inherits the *get_name()* method from the Business Class.

```
def main():

    b1 = ClothingStore("Ferry's", 4, 20)

    print(b1.get_inventory())
    print(b1.get_name())

main()
```

```
20
Ferry's
```

The Theater business would be handled the same way. The Theater Class would be implemented including any specifics (seats in the example) and inherit everything from the Business Class. The initializer for the Theater class calls the __init__ function for the Business class, and passes the three parameters including **self** that the Business class initializer requires. It then assigns seats to its' own seats data attribute. The *get_seats()* method is included for the example that follows.

```
class Theater(Business):

    def __init__(self, name, employees, seats):

        Business.__init__(self, name, employees)

        self.__seats = seats

    def get_seats(self):
        return self.__seats
```

The object creation and method calls would operate the same as well.

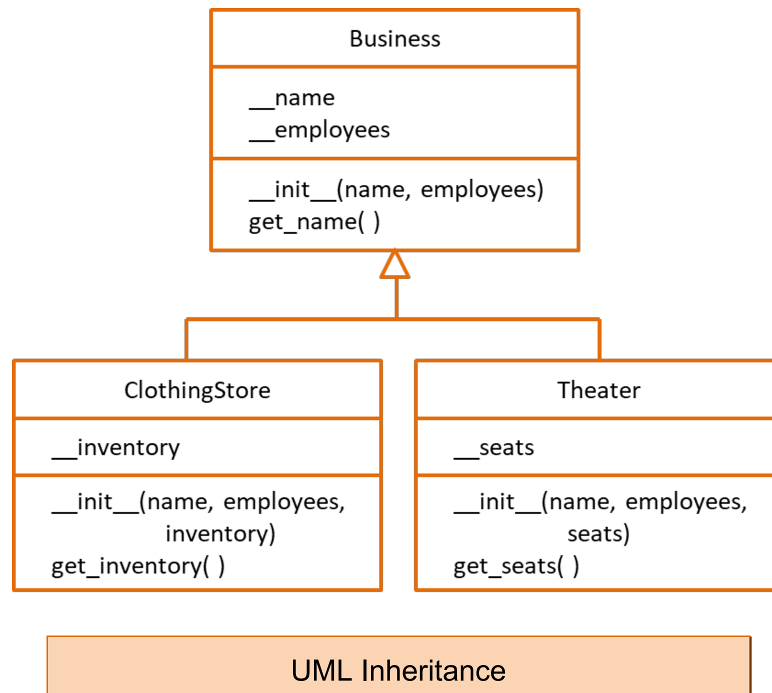
```
def main():

    t1 = Theater('Town Theater', 12, 450)
    print(t1.get_seats())
    print(t1.get_name())

main()
```

```
450
Town Theater
```

The following example creates instances of both derived classes and an instance of the Business class.

```
def main():

    t1 = Theater('Town Theater', 12, 450)
    c1 = ClothingStore("Ferry's", 4, 20)
    b1 = Business('General', 35)

    print(t1.get_name() + '\t' + str(t1.get_seats()))
    print(c1.get_name() + '\t' + str(c1.get_inventory()))
    print(b1.get_name())


main()
```

In the Unified Modeling Language (UML), inheritance is represented with an open arrow head pointing to the base class. The base class diagram includes all of its' attributes, and each of the derived classes would contain theirs. A UML diagram for the Business Class example follows. Inheritance is one way, and base classes no nothing about derived classes.



UML Inheritance

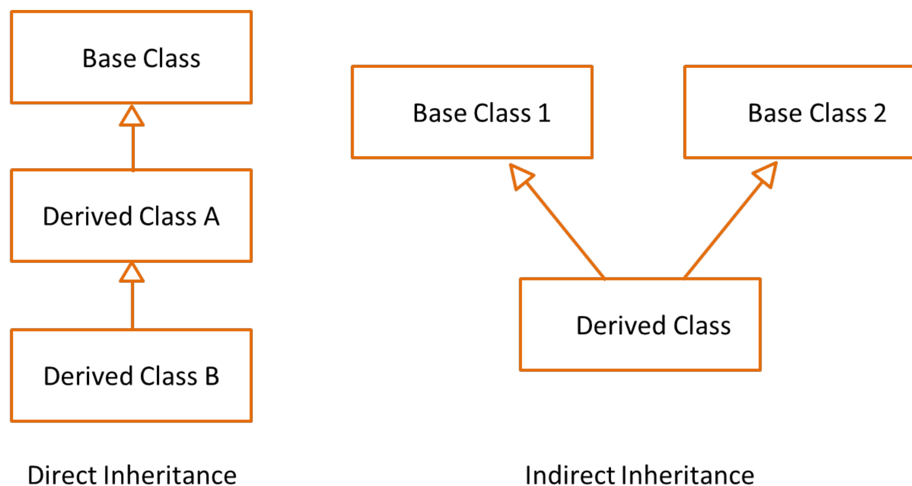
Polymorphism

In the examples, there was a single method for each operation. A derived class can actually have a method with the same name as a method in the base class that operates differently. This is known as ***polymorphism***, or the ability to take on many forms. The derived class method can ***override*** the base class method. Consider a situation where the Business Class has a display method, and the Theater, and Clothing Store derived classes have a display method as well. The proper method would be called based upon which object is calling the method.

Multiple Class Inheritance

A class can inherit (be derived) from multiple classes. The derived class inherits all of the attributes of both classes. The hierarchy diagram below indicates the

direct inheritance of Derived Class B from Derived Class A which inherits from the Base Class. A class can also inherit from two or more distinct classes. The diagram below indicates indirect inheritance. A careful review of both base classes is required to ensure that there are no conflicts.



When defining a class with multiple-inheritance, both classes are listed as parameters for the derived class, and both initializers for the classes are called.

```
class DerivedB(DerivedA, BaseClass):
    def __init__(self):
        DerivedA.__init__(self)
        BaseClass.__init__(self)
```

Both forms of multiple-inheritance add complexity and make the code and classes harder to use in other programs because both of the base classes need to be copied as well. This also lowers cohesion and increases coupling.

Determining Class Identity

As the classes become more complex and methods override other methods, it is often necessary to determine if a class is an instance of or a derived class of another. The ***isinstance*** function returns True if it is and False if it is not. The general format is shown below where object is the reference to the object being tested, and Class is the name of the class (can be base class).

```
isinstance(object,Class)
```

Before a task is completed, a test for the proper object can occur.

Chapter 9 Review Questions

1. Hiding the implementation of a class is referred to as _____________.
2. The data and methods of a class are called ____________.
3. Program statements and other objects access an object's attributes through the ______________.
4. An object is an ___________ of a class.
5. The ________________ declares the data and methods for a class.
6. The method that creates and initializes an object is called the _____________.
7. The values stored in an object's data attributes are referred to as the object's __________.
8. Two underscores preceding a class data attribute indicate that the element is _________.
9. Methods that access an object's data attributes are called ______________.
10. Methods that set or change an object's data attributes are called __________.
11. The _________ function provides a way to output an objects state.
12. _________ refers to the degree to which an object represents a single abstraction without external dependencies.
13. _________ refers to the degree to which an object is dependent upon another.
14. A way in Python to serialize objects into byte streams is called __________.
15. A _______ diagram can be used to document the data and method attributes of a class.
16. ______________ allows a base (super) class to contain common elements for derived (sub) classes.
17. ______________ is the ability to take on many forms.
18. The __________ function can be used to determine if an object is an instance of or derived class of another class.

Chapter 9 Short Answer Exercises

1. What is the difference between a class and an object?
2. What is an instance of a class?

3. What is the object reference in the following statement?

    account.get_balance( )

4. What parameters need to be passed to the following constructor?

    def __init__(self, name, age)

5. What change is needed to indicate that a data attribute is private?

6. What is the __str__ method used for and how is it called?

7. In the following statement, what is the base (super) class? What is the derived (sub) class?

    class Car(Vehicle)

Chapter 9 Programming Exercises

1. Write a class definition for a Circle Class that has a data attribute for radius, a constructor that accepts a radius and initializes the instance attribute, and the following methods:

    get_circumference( )
    get_area( )

2. Using the Circle Class from #1, write a program that creates a Circle object with a radius of 6, and displays the circumference and area of the circle.

3. Implement a Product class that has data attributes for description, price, and inventory. Write a constructor that accepts parameters for the attributes and initializes them, and a method to display a product's information. Write a program to create the product objects below and display them.

Product	Price	Inventory
Mug	8.50	23
T-shirt	12.95	45
Towel	18.50	36

4. Implement a Gas Pump class that has data attributes for gallons pumped, price per gallon, and total sale. The constructor will accept a dollar amount for the gas purchased. The class will have a method to allow setting the price per

gallon, and a method to display the gallons pumped and the sale. Then write a program that will prompt for the price per gallon and the amount in dollars to pump. It will then create a Gas Pump object and display the results. The program will have a loop to create a new pump without restarting the program. Sample main and output shown below.

```
def main():

    keep_going = 'y'

    while keep_going == 'y':
        ppg = float(input('Enter price per gallon: '))
        dollars = float(input('Enter amount: $'))

        g1 = GasPump(dollars)
        g1.set_price_per_gallon(ppg)
        g1.display_result()
        keep_going = input('Enter y for another.')

main()
```

```
Enter price per gallon: 2.95
Enter amount: $20.00
GALLONS PUMPED:  6.67 Price: $20.00
Enter y for another.
```

5. Write a program using the class definition below that creates a Date object, and sets the day, month, and year, then displays the date in mm/dd/yyyy format.

```
class Date:
    def __init__(self):
        self.__day = 1

    def set_day(self, day):
        self.__day = day

    def set_month(self,month):
        self.__month = month

    def set_year(self, year):
        self.__year = year


    def get_date(self):
        return self.__month, self.__day, self.__year
```

6. Create a UML diagram for the Date Class in #5.

7. Implement an *Oven* Class for a microwave oven that accepts a time to cook in minutes that is greater than 0 and less than 12, three power levels (1, 2, 3 – for low, medium, high), and start. The class methods will validate the input, and use default values when invalid data is entered and display an error. Write a program that creates an object and prompts for the input. The program will have a loop to create a new object without restarting the program. Sample output shown below.

```
Enter minutes: 3
Enter power level: 2

Enter "y" to start: y
Cooked:  3 minutes  at  MED  power

Enter "y" for another:
```

8. Implement the *Business* Class from Ex. 9.9 with the data attributes for name and employees and methods for accessing them. Then implement a derived class *Restaurant* that inherits from the *Business* Class and has data elements for tables, and seats.

   Write a program that creates an instance of the *Restaurant* Class named Sally's with 14 employees, 15 tables and 65 seats. Add a method to the program that displays all of the information for Sally's Restaurant.

   Sample output:

```
Name:  Sally's
Employees:  14
Tables:  15
Seats:  65
```

Chapter 9 Programming Challenges

#1 – Product Pickling

Using the Product Class from #3, create the three products, and serialize (dump) them and write them to a file, and close the file. Open the file and *load* the products (retrieve and de-serialize) them into object references (names) that are different from the names they were given when they were created. Then display the information for the products.

#2 – Elevator Class

Implement an Elevator Class for elevators that can travel to floors 1 - 8, “know” what elevator they are (1, 2, 3), “know” what floor they are on, and “know” whether they are active or waiting.

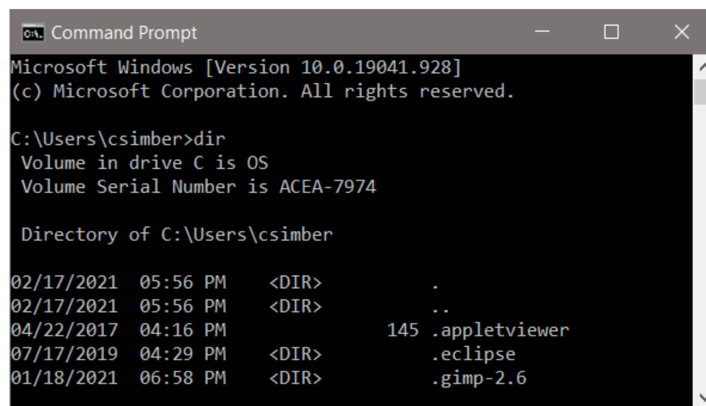
Write a program that creates three (3) elevators, and starts them each at the ground floor. Using a loop with 10 iterations, randomly select one of the elevators to move (this one is active and the others are waiting) and send it to a random floor. The same elevator cannot be moved two times in a row. Display the current state for each of the elevators at each execution of the loop in columns and sets of three as shown. Only one elevator should be active at each interval and the others should remain on their current floors.

```
Elevator  Floor     Status
--------------------------
    1       0       Waiting
    2       8       Active
    3       0       Waiting

    1       4       Active
    2       8       Waiting
    3       0       Waiting

    1       4       Waiting
    2       6       Active
    3       0       Waiting
```

Chapter 10

Graphical User Interfaces (GUIs)

Graphical User Interfaces (GUIs) were originally created by researchers at the Xerox PARC (Palo Alto Research Center) and quickly became the user-preferred choice for interfacing with computers in the 1980's. Prior to this, command line interfaces (shown below) were used to interact with computers, and in some cases they continue to be used.



GUIs are event driven by user input such as clicking on a button or tab, scrolling, or resizing a window. The program responds to user input which determines the sequence of many of the events. Therefore, careful design is required to control access to the events. For instance, a user may click a button to compute a result before entering required values. Scenarios like this should be considered

during interface design, since they increase the input validation aspects of a program. A value needed for computation must be entered by the user before allowing computation, and the value entered must be within the correct range of values to avoid issues such as division by zero. Consider a program that computes the circumference of a circle based on an input of radius.

1. The radius must be input prior to computation
2. The radius input must be a positive number

The graceful handling of incorrect input is required for a robust and well-engineered solution. In a non-GUI program, we might use a loop that iterates until a correct value is entered. It would display an error message to alert the user, and re-prompt for input inside the loop. The same concept is true for a GUI program, but with the added requirement of employing windows to handle the tasks. This situation will be explored later. Generating a main GUI is the first step which requires creating a window with ***controls*** (widgets or components). A control is an element that enables a user to accomplish some function or to access an area of the program. Python is a well suited language for creating graphical user interfaces through the ***Tkinter*** module. The module is installed with Python and provides windows and controls that are easy to program. The Tkinter package is the standard Python interface into TK GUI Toolkit which is used by developers in other languages as well. The name Tkinter is short for TK Interface. Some Tkinter controls are listed here.

Component	Description
Button	causes an action or event when clicked
Canvas	rectangular area for graphics
Check button	On/Off position check boxes
Entry	single line entry control
Frame	container that can hold components
Label	area that displays one line of text
List box	user selection list (option-list)
Menu	list exposed when a menu button is clicked
Radio button	select/deselect component

Tkinter Controls

Before selecting controls for an interface, a preliminary design should be completed to provide a layout for the window and an idea of how it will look and operate. Walking through the program operation steps the way that a user would is helpful at this stage. The user will interact with the program through the GUI, and should be easy to use, have intuitive controls, and labels. A user should not have to wonder how the use the program or what units to enter.

The controls required for the interface depend on what the program does and the user interaction. A few labels and buttons may be adequate, or radio buttons or option lists could be employed. These considerations during the design phase will save time redesigning or reconfiguring an inadequate or problem interface. Programmers often overlook essential aspects of the interface since they know what the program does, how it functions, and the inputs required. The Agile process typically involves stakeholder reviews and in some cases the customer. This provides an opportunity for people not familiar with the planned design to offer suggestions for improvement. It also eliminates surprises when the final product is delivered.

Interface Example

Consider a GUI weather program that receives user input for temperature and wind speed, and computes the wind chill factor when a button is clicked. The pseudocode for the program lists the steps in the program and helps to identify controls.

Ex. 10.1 – GUI program – Wind Chill Factor example

```
Step 1  the user enters temperature              data entry
Step 2  the user enters wind speed               data entry
Step 3  the compute button is clicked            button
Step 4  the input is validated
        -   if the input is valid
                o   compute the wind chill
                o   display the result
        -   otherwise
                o   alert the user to the error   dialog
                o   clear the inputs
Step 5  go to Step 1
```

A sketch of the planned interface helps to verify requirements and locate the controls that will be needed.

Enter the temperature in Fahrenheit:

Enter the wind speed in miles-per-hour:

The wind chill factor is:

Compute

Generating the Interface Window

Programmers use an object oriented approach to GUI development, and the window for the example will be created as an instance of a class (*WeatherWin*). The class below is a single window with a minimum size setting, a title, and a label. The line numbers are included for the explanations that follow. Line numbers in IDLE can be displayed by selecting them from the options menu.

Ex. 10.2 – GUI program example

```
 3 import tkinter as tk
 4 
 5 class WeatherWin:
 6     def __init__(self):
 7 
 8         self.win = tk.Tk()
 9         self.win.title('Weather Program')
10         self.win.minsize(width=550, height=200)
11 
12         self.hdg_label = tk.Label(text='Test Text Label',\
13                                   font=('Consolas',16), \
14                                   fg='orange')
15         self.hdg_label.grid(row=2,column=1)
16 
17 
18         tk.mainloop()
19 
20 wWin = WeatherWin()
```

Line 3 imports the Tkinter module as "tk" which allows using tk as an alias when accessing the library as shown on line 8 when the window is created and on line 12 when the label is created

Line 5 declares the class (in this example WeatherWin)

Line 6 begins the constructor (initialization function)

Line 8 creates the window

Line 9 adds a title to the window border

Line 10 sets a minimum size for the window

Lines 12-14 create a label with text, font style and size options, and color

Line 15 positions the label created on line 12 using grid geometry

Line 18 starts the tkinter main loop

Line 20 creates an instance of the *WeatherWin* class called wWin

The tkinter ***main loop*** in the program is used to "listen" for events like a button click when the program is running. It executes when the program starts and continues running until the user ends the program. When executed, the code generates the window below with the title on the border, and the label. Notice that there is no main function for this program. The code (repeated here) that generates an instance of the class creates the window.

```
wWin = WeatherWin()
```

Building a portion of the project and testing that portion before moving on is referred to as the "build a little, test a little" approach. As new code is added, any issues or errors that surface would be in the added code. It is easier to debug five lines of code than it is to debug fifty lines of code.

Window with Title and Label

Positioning Controls

To position controls on an interface, Tkinter provides Geometry Managers that include the pack(), grid(), and place() methods.

pack() organizes within a block using a few available options

grid() organizes using rows and columns with customizing options

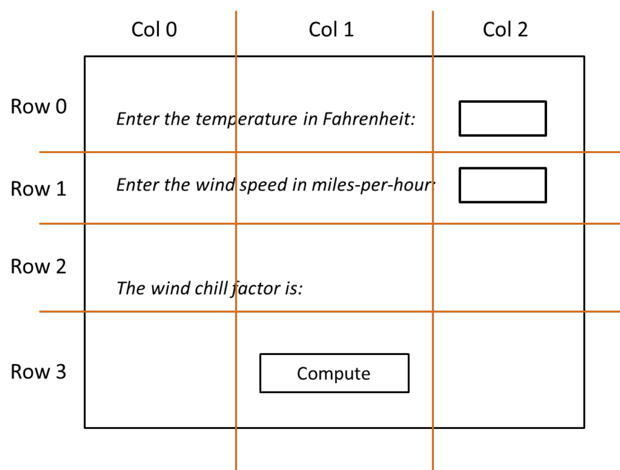
place() organizes using x, y coordinates

Due to the flexibility and options available in ***grid***, it will be used in the examples. The line of code in the example that positioned the label (repeated below) assigns row 1 and column 2, but the label is positioned top-left in the window. This is because grid will size each row and column to the smallest size required for the items that are in them. In this case there is nothing in row 0 or 1 and nothing in column 0, so the position of the label doesn't display where the code indicated. This can be resolved by setting row and column sizes.

```
self.hdg_label.grid(row=2,column=1)
```

GUI Design

Positioning elements should be completed in the design phase. The sketch for the example window included the program title, labels for prompts, entry controls for the user, and a compute button. Since grid positions elements in rows and columns, adding lines to the preliminary sketch provides a better representation of the interface and where elements will be located.



Interface Sketch Highlighting Rows and Columns

The modified sketch with row/column lines shows that some rows have a greater height than others and that the labels span multiple columns. This is not an issue since there are grid options for setting the row height and column width individually using ***configure*** and for allowing a control to span multiple rows/columns. The lines below set the first row height to 50 pixels and the first column width to 100 pixels. The first argument is the row or column number, and the second is the size. Numbering begins at zero for rows and columns.

```
rowconfigure(0,50)

columnconfigure(0,100)
```

Table 10.1 shows some of the grid options available and their descriptions.

Option	Description
column	column location of the control
columnspan	allow a control to span multiple columns
ipadx	horizontal padding within the control's borders
ipady	vertical padding within the control's borders
padx	horizontal padding around a control within a cell
pady	vertical padding around a control within a cell
row	row location of the control
rowspan	allow a control to span multiple rows
sticky	one or more of N, S, E, W to align controls within cells

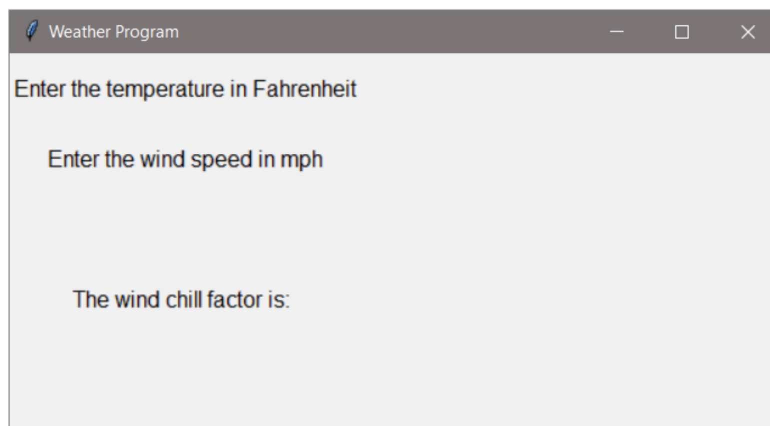
Table 10.1 – Grid Geometry Options

Planning the positions of the controls and considering the use of options can save time adjusting after the fact. Adding padding or a span option to one control can move others which then forces changes to them as well. Then adjusting the options for those controls can counteract the original change or create more needed adjustments. For the example, the columns will be the same width and the row heights will be customized to accommodate the controls. Positioning the labels will be the first step.

Ex. 10.3 – grid configuration and positioning controls on a GUI

```
# configure columns and rows
for c in range(3):
    self.win.columnconfigure(c, minsize=100)

self.win.rowconfigure(0, minsize = 50)
self.win.rowconfigure(1, minsize = 50)
self.win.rowconfigure(2, minsize = 150)
self.win.rowconfigure(3, minsize = 250)


self.temp_label = tk.Label(text='Enter the temperature in Fahrenheit',\
                          font=('Arial',12))
self.temp_label.grid(row=0,column=0)

self.ws_label = tk.Label(text='Enter the wind speed in mph',\
                          font=('Arial',12))
self.ws_label.grid(row=1,column=0)

self.out_label = tk.Label(text='The wind chill factor is: ',\
                          font=('Arial',12))
self.out_label.grid(row=2,column=0)
```

In the code above, a loop is used as an example for setting the column sizes since they are the same. The rows are configured individually. The sizes for the rows and columns are really a guess at this point, and will be adjusted after the window is displayed and reviewed. After each label is created with text and the font option, the location on the grid is assigned. It is best to handle this as a two-step process. The updated code output is shown below, and the labels are centered within the column by default. An option for alignment in grid is called ***sticky*** and it accepts an assignment of N, S, E, or W. Using W for west would put the text against the left hand side of the GUI, but a tab can be added to the label text to move it to the right (as a simple solution).

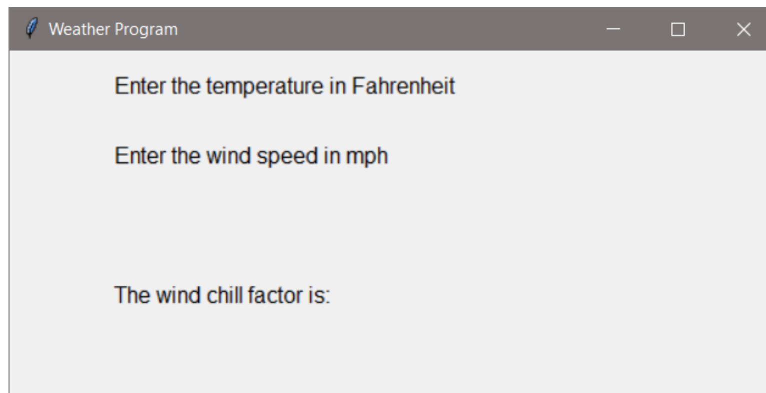


Positioning Labels

The modified code adds a tab to the text for each label and the sticky option is assigned "W" when assigning the grid rows and columns for the label positions.

```
self.temp_label = tk.Label(text='\tEnter the temperature in Fahrenheit',\
                           font=('Arial',12))
self.temp_label.grid(row=0,column=0, sticky='W')

self.ws_label = tk.Label(text='\tEnter the wind speed in mph',\
                           font=('Arial',12))
self.ws_label.grid(row=1,column=0, sticky='W')

self.out_label = tk.Label(text='\tThe wind chill factor is: ',\
                           font=('Arial',12))
self.out_label.grid(row=2,column=0, sticky='W')
```

The result is aligned labels and space away from the left edge of the window.



Positioning Labels and Alignment

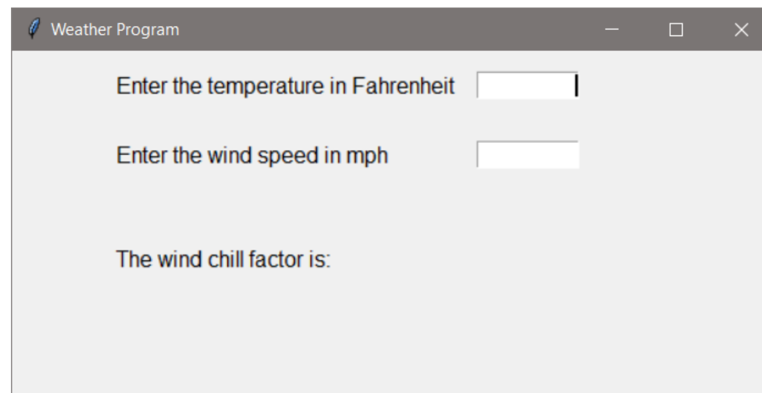
Entry Controls

Next, the entry controls that allow the user to enter the input will be added on the same rows as the prompts for temperature and wind speed, but in the next column. The entry controls accept a width in characters, justification for the text entered by the user, and a font option among others. The focus for temperature is forced so that the curser is in the text entry control when it is created.

Ex. 10.4 – entry controls creation and grid positioning

```
self.temp_entry = tk.Entry(width = 10, justify='right', \
                           font=("Helvetica",10))

self.temp_entry.grid(row=0, column=1)
self.temp_entry.focus_force()
```

The wind speed entry control is handled the same way, but without the forced focus. The resulting display is shown here. The GUI is starting to take shape.



Positioning Entry Controls

There are options for entry controls to customize the foreground, background, font, relief, and there is a show option to display a character such as "*" instead of what the user is typing. The methods include ***get()*** which returns the text in the entry control. This will be used when the button (added next) is clicked.

Button Controls

GUIs typically contain button controls that allow some action to take place when they are clicked. The Tkinter button has many options for customization and can be positioned using grid. In the example, a button that reacts when it is clicked will call a function that gathers the input text, computes the wind chill factor, and displays the result. This is commonly referred to as a ***callback function***. The *command* option for the button assigns the action to be performed when the button is clicked. In the code below a *compute_wc* function will be called. To center the button in the interface, the *columnspan* option is used allowing it to span all three columns (centered by default).

Ex. 10.5 – button creation and grid positioning

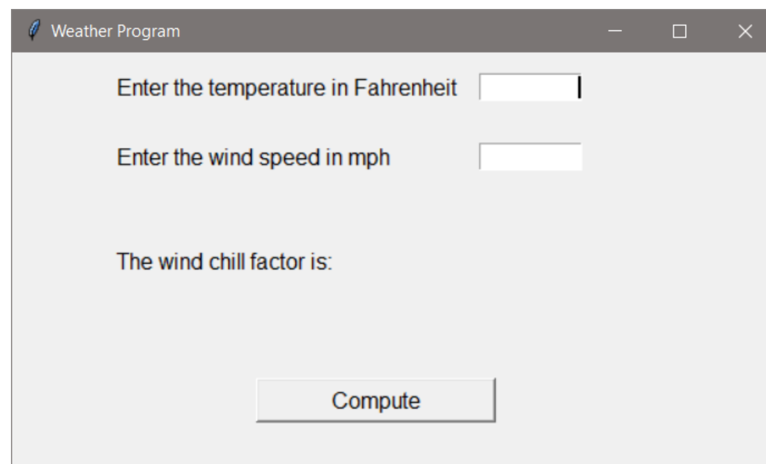
```
self.compute_button = tk.Button(text='Compute', width=18, \
                                font=("Helvetica",12),\
                                    command=self.compute_wc)

self.compute_button.grid(row=3,column=0, columnspan=3)
```

There are options for buttons that allow for customization. The commonly used options are listed in the table below.

Option	Description
bg	background color
fg	foreground (text) color
font	text font for the button face
height	height of the button in text lines (font dependent)
image	image to be displayed on the button
justify	multiple text line alignment (LEFT, CENTER, RIGHT)
padx	padding left and right of text
pady	padding above and below text
relief	type of border: SUNKEN, RAISED, GROOVE, and RIDGE
state	enable or disable the button (normal, active, disabled)
width	the width of the button

Table 10.2 – Button Options

Note that the code for the actual computation of the wind chill factor has not been written yet. Once the button is positioned and displayed, the function for computing will be added. The result of adding the button code is shown below.



The interface controls for the example have been created and positioned, but there is no functionality. The callback function for the button will be created within the class to simplify the process and provide access to the interface controls. Later an example will show how this can be located in another module or by using what is called a lambda expression. For now, the *command* option assignment will execute a function within the class.

```
self.compute_button = tk.Button(text='Compute', width=18, \
                          font=("Helvetica",12),\
                     command=self.compute_wc)
```

The function will include verification of the input, computing the wind chill factor if valid data was entered, and updating the label to show the result of the computation. A simple output statement can be used for testing purposes.

```
def compute_wc(self):
    print('Button was clicked')
```

The callback function will use the entry control's *get()* function to retrieve the input as a string. To use the input in calculations requires casting them to floats (if possible). Print statements are used below for testing.

Ex. 10.6 – compute function obtaining input

```
def compute_wc(self):
    temp_string = self.temp_entry.get()
    ws_string = self.ws_entry.get()
    try:
        temp = float(temp_string)
        ws = float(ws_string)
        print(temp, ws)
    except:
        print('Bad data')
```

When the data entered is invalid, the user must be alerted to the problem and be given the ability to correct the issue without restarting the program. A dialog box can be used for this purpose.

Dialog and Information Boxes

Issues associated with GUI operation are typically handled using a dialog or message box that explains the issue and has an "OK" button that must be clicked

to continue. The ***tkinter.messagebox*** module provides information boxes for this purpose and requires importing that module. Available functions are listed below.

Message Box Functions

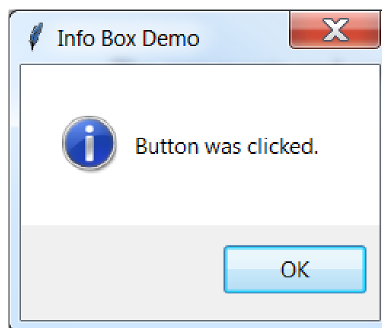
```
showerror()      askquestion()    askretrycancel()
showwarning()    askokcancel()    askyesno()
showinfo()
```

The arguments and options for the functions include:

```
function name    choice of message box (showinfo)
title            the title on the title bar
message          the text to be displayed
```

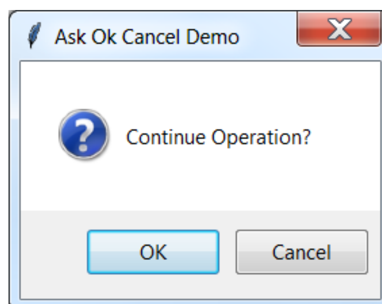
The different functions produce different message boxes, and there are options including setting the window title and text in the dialog as shown below. The example produces the standard ***showinfo*** message box which includes text and the "OK" button.

```
tk.messagebox.showinfo('Info Box Demo', 'Button was clicked.')
```



This line of code produces the ***ask-ok-cancel*** box shown below.

```
tk.messagebox.askokcancel('Ask, OK, Cancel', 'Continue Operation?')
```

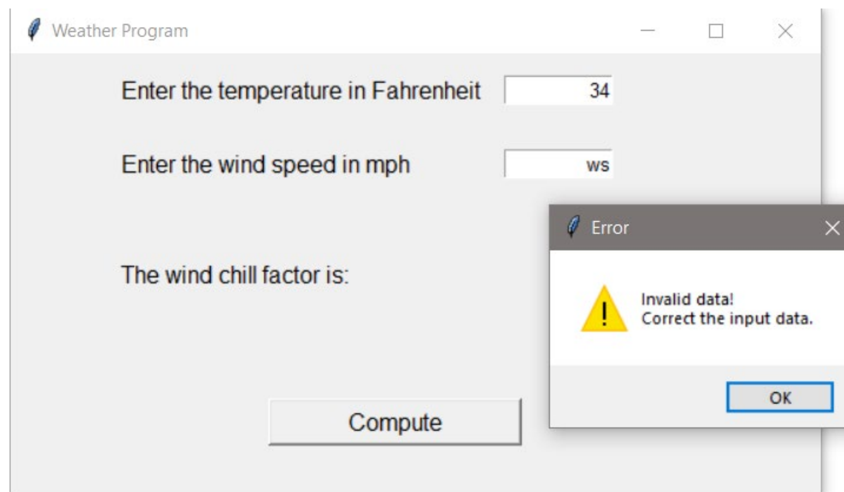


Message boxes provide a simplified way of handling errors. For the compute function, the input will be obtained from the entry controls and checked before a calculation is performed. If the input is invalid, a message box in the except clause will alert the user.

Ex. 10.7 – compute function with message box

```
def compute_wc(self):
    temp_string = self.temp_entry.get()
    ws_string = self.ws_entry.get()
    try:
        temp = float(temp_string)
        ws = float(ws_string)
        print(temp, ws)
    except:
        tk.messagebox.showwarning('Error',
                'Invalid data!\nCorrect the input data.')
```

The message box alerts the user and allows the program to continue after the "OK" button is clicked.



Invalid Data Dialog Box

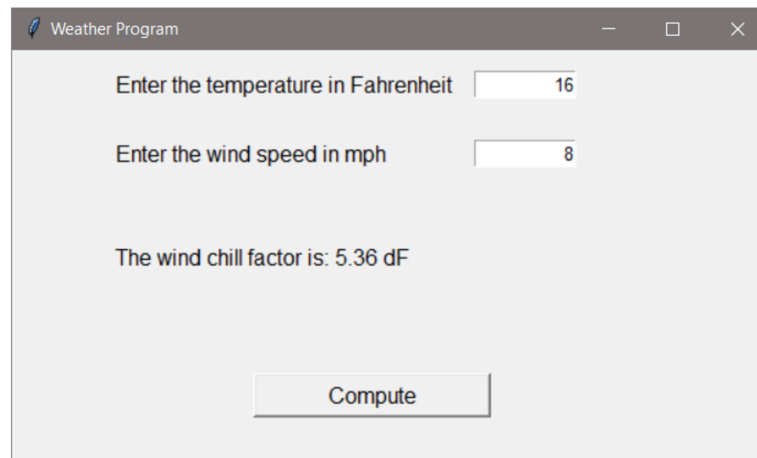
The final part of the example program includes calculating and displaying the wind chill factor when valid data is entered and the button is clicked. The result of the calculation could be handled using an additional label, but the existing

label can be updated to include the results by appending the wind chill factor to the text, and using the ***config*** method to update the label.

Ex. 10.8 – compute function with label update using config

```
def compute_wc(self):
    temp_string = self.temp_entry.get()
    ws_string = self.ws_entry.get()
    try:
        temp = float(temp_string)
        ws = float(ws_string)
        wind_chill = 35.74 + (0.6215 * temp) - \
                     (35.75 * (ws**0.16)) + \
                     0.4275 * temp * (ws**0.16)

        self.out_label.config(text=('\tThe wind chill factor is: ' +
                                    str(format(wind_chill, '.2f'))+ ' dF'))

    except:
        tk.messagebox.showwarning('Error',
                'Invalid data!\nCorrect the data entered.')
```

The results of the changes are shown below.



Updating a Label Using Config

StringVar

Another way to update/change a label is to use a StringVar object. The StringVar modifies any control that uses it whenever the contents of the StringVar is changed. This provides the ability to have an immediate update to a control anytime the value that is stored in the StringVar object changes. A StringVar is

declared and assigned to a control using the ***textVariable*** assignment. Below, a StringVar is declared, and then assigned to a label.

```
my_svar = tk.Stringvar()
my_label = tk.Label(textVariable= my_svar)
```

The StringVar can be updated by a variety of sources using the ***set()*** method. The example has been updated to include declaring a StringVar and assigning it to the output label. An ***IntVar*** is the integer version of the object, and operates the same way.

```
self.output_svar = tk.StringVar()

self.out_label = tk.Label(textvariable=self.output_svar, \
                          font=('Arial',12))

self.output_svar.set('\tThe wind chill factor is: ')

self.out_label.grid(row=2,column=0, sticky='W')
```

The compute function is modified to use *set()* to update the StringVar.

```
self.output_svar.set('\tThe wind chill factor is: ' +
                     str(format(wind_chill, '.2f'))+ ' dF')
```

Radio Buttons

Very often the design or operation of the program requires that only one selection be made by the user. Radio buttons accommodate this because they are mutually exclusive, and if a second button is selected, the button that was previously selected is unselected. The options for radio buttons are similar to other controls including text and fonts, and they can be located using grid locations.

The first statement below declares a StringVar named radio_var to store the radio button selection. The next line uses set '1' to set one of the buttons as a default. The next lines declare a radio button with text and font options, a StringVar is assigned to the radio button's variable, and a value for the button is assigned. The button is then positioned using grid.

```
self.radio_var = tk.StringVar()

self.radio_var.set('1')

self.rad_one = tk.Radiobutton(text = ' Radio Button One',\
                              font=('Arial',12), \
                                variable=self.radio_var, value='1')

self.rad_one.grid(row=3, column=1, sticky='W')
```

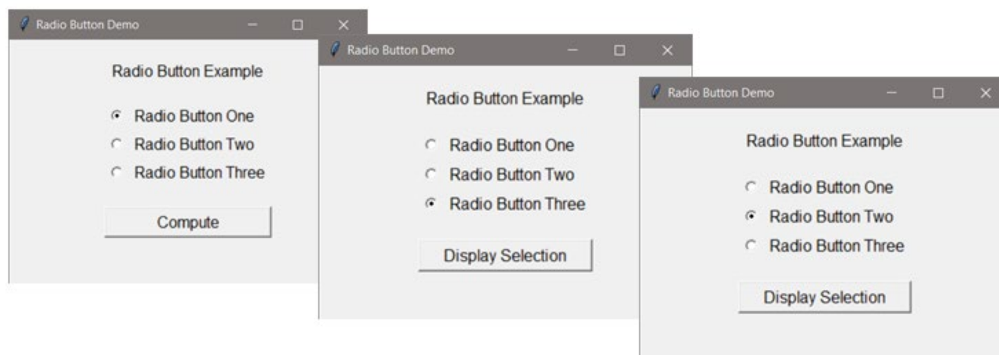
The example below expands on the lines above and implements three radio buttons. The complete program is shown, and an explanation follows.

Ex. 10.9 – Radio Buttons

```
import tkinter as tk


class RadioWin:
    def __init__(self):

        self.win = tk.Tk()
        self.win.title('Radio Button Demo')
        self.win.minsize(width=350, height=250)

        # configure columns and rows
        for c in range(3):
            self.win.columnconfigure(c, minsize=100)

        for r in range(8):
            self.win.rowconfigure(r, minsize=20)

        self.radwin_label = tk.Label(text='Radio Button Example',\
                                     font=('Arial',12))
        self.radwin_label.grid(row=1,column=0, columnspan=3)


        self.radio_var = tk.StringVar()

        self.radio_var.set('1')

        self.rad_one = tk.Radiobutton(text = ' Radio Button One',\
                                      font=('Arial',12), \
                                        variable=self.radio_var, value='1')
        self.rad_one.grid(row=3, column=1, sticky='W')


        self.rad_two = tk.Radiobutton(text = ' Radio Button Two',\
                                      font=('Arial',12), \
                                        variable=self.radio_var, value='2')
        self.rad_two.grid(row=4, column=1, sticky='W')


        self.rad_three = tk.Radiobutton(text = ' Radio Button Three',\
                                      font=('Arial',12), \
                                        variable=self.radio_var, value='3')
        self.rad_three.grid(row=5, column=1, sticky='W')
```

```
        self.compute_button = tk.Button(text='Display Selection', \
                                width=18, font=('Arial',12),\
                                command=self.radio_react)

        self.compute_button.grid(row=7,column=0, columnspan=3)

    def radio_react(self):
        print('The button selected is: ' + str(self.radio_var.get()))


rWin = RadioWin()
```

The radio buttons are positioned using grid, and all of the buttons are assigned the same variable *radio_var*, but are assigned a different integer in *value*. After a selection is made, the user clicks on the *Display Selection* button which has a command to call the *radio_react* function, and the function obtains the value of the selected radio button using *get()*. The default button is initially selected, and when another button is selected, that button is deselected. For the example, the output from the function is displayed whenever the button is clicked.




```
The button selected is: 1
The button selected is: 3
The button selected is: 2
```

After retrieving the selected button using *get()*, conditional statements can be used to respond with specific operations.

```
def radio_react(self):
    button_num = self.radio_var.get()
    if button_num == '1':
        print('Number one')
```

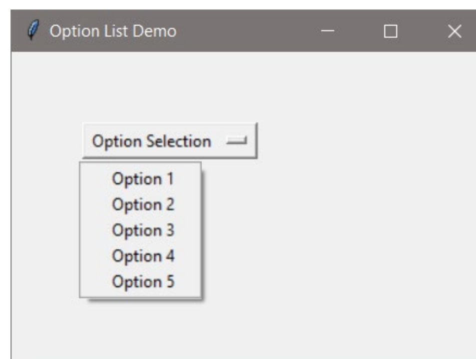
Option Lists

The option list is another mutually exclusive control that allows the user to select from a drop-down list when it is clicked. The option list can be positioned using grid geometry, and has *font* and other options including *state* (disabled/active).

Ex. 10.11 – Option List (window code omitted)

```
    optionList = ('Option 1', 'Option 2', 'Option 3', \
                  'Option 4', 'Option 5')

    self.option_var = tk.StringVar()

    self.option_var.set('Option Selection')
    self.option_menu = tk.OptionMenu(self.win,\
                                     self.option_var, *optionList,
                                      command=self.list_changed)
    self.option_menu.grid(row=1, column=1)

    tk.mainloop()


def list_changed(self, *args):
    print('List changed: ' + self.option_var.get())
```

The implementation is much the same as the radio buttons. A StringVar stores the value selected and a command is linked to the list. The function called needs to receive the arguments and use the *get()* method to obtain the selection.



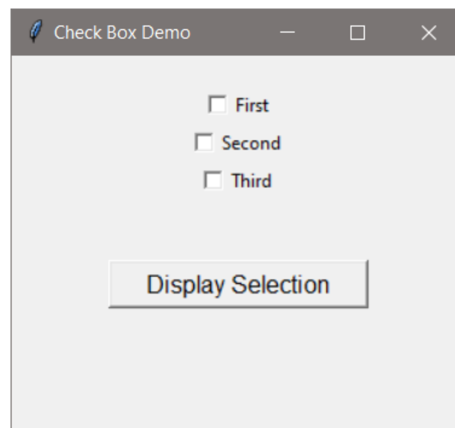
Check Boxes

Check boxes allow multiple selections by the user, and require individual variables for each check box. The code example below includes the declaration of an IntVar for each check box, and the assignments to the variable in the declaration of the checkboxes. An on-value and off-value option assigns an

integer to each of the checkboxes (selected/unselected). The button command is assigned a function that will determine the boxes that are checked by calling the get() function for each checkbox.

Ex. 10.10 – Check Buttons (window code omitted)

```
        self.check_1 = tk.IntVar()
        self.check_2 = tk.IntVar()
        self.check_3 = tk.IntVar()


        self.chk_1 = tk.Checkbutton(text='First',
                                    variable = self.check_1,
                                    onvalue = 1,offvalue = 0)
        self.chk_1.grid(row=1,column=1)

        self.chk_2 = tk.Checkbutton(text='Second',
                                    variable = self.check_2,
                                    onvalue = 1,offvalue = 0)
        self.chk_2.grid(row=2,column=1)

        self.chk_3 = tk.Checkbutton(text='Third',
                                    variable = self.check_3,
                                    onvalue = 1,offvalue = 0)
        self.chk_3.grid(row=3,column=1)

        self.compute_button = tk.Button(text='Display Selection', \
                               width=18, font=('Arial',12),\
                               command=self.check_react)

        self.compute_button.grid(row=6,column=0, columnspan=3)

    def check_react(self):
        print('IN PRINT')
        if self.check_1.get() == 1:
            print('First box')
        if self.check_2.get() == 1:
            print('Second box')
        if self.check_3.get() == 1:
            print('Third box')
```

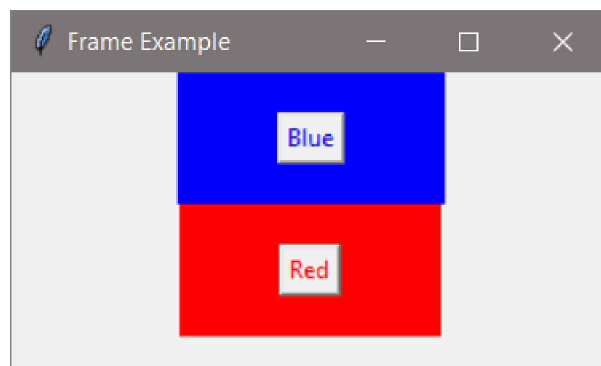


Frames

The tkinter frame (panel) is a container that can hold other controls. It displays as a rectangle and is used to organize other controls and provide additional customization. Multiple frames can hold different controls arranged in different ways, and the frames can be positioned in the main window using grid geometry. The frame example creates two frames with a button on each, and uses padding for x and y around the buttons to expand and display the frames. The color has been added to highlight the frames.

Ex. 10.12 – Frame example

```
class FrameWin:
    def __init__(self):

        self.win = tk.Tk()
        self.win.title('Frame Example')
        self.win.minsize(width=300, height=150)

        self.f1 = tk.Frame(self.win, padx=50,
                           pady=20, bg='blue')

        blue_btn = tk.Button(self.f1, text="Blue",fg="blue")
        blue_btn.pack()

        self.f2 = tk.Frame(self.win, padx=50,
                           pady=20, bg='red')

        red_btn = tk.Button(self.f2, text="Red",fg="red")
        red_btn.pack()

        self.f1.pack()
        self.f2.pack()

fWin = FrameWin()
```



Frame options include:

bd	size of the border (defaults to 2 pixels)
bg	background color
height	height of the frame
relief	type of border: flat, groove, raised, ridge, solid, or sunken
width	width of the frame

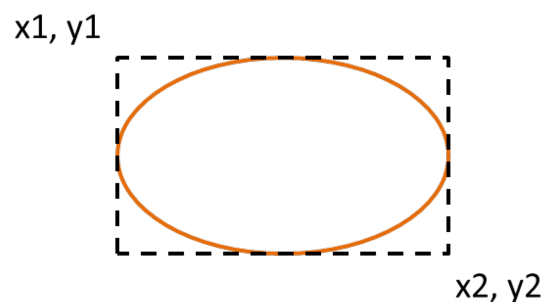
The default relief for a frame is flat. To use the other relief options the border size must be increased using the *bd* option. The default size for a row and column is 1 pixel, so they must be resized to accommodate the frame.

Canvas

The tkinter module also provides some essential functionality for drawing graphics. A canvas can be used to draw graphs, charts, plots, lines, and geometric shapes. It can also be used with the Matplotlib module. The key to drawing on a canvas is remembering that the x, y coordinate system for the canvas locates 0, 0 at the top-left of the canvas and the units are pixels. The general formats for drawing are:

```
create_line(x1, y1, x2, y2, options...)
create_rectangle(x1, y1, x2, y2, options...)
create_oval(x1, y1, x2, y2, options...)
create_text(x, y, text=' ', options...)
```

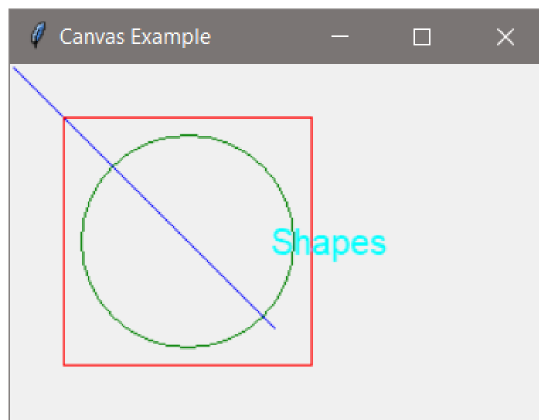
Creating an oval or rectangle requires providing a starting and ending opposite-corner coordinate pair. In the case of the oval, it is a bounding rectangle as shown here.



The example below creates a window and Canvas, and draws a line, rectangle, circle (an oval with equal height and width), and text. The canvas is then positioned on the frame (window) using grid.

Ex. 10.13 – Canvas example

```
class FrameWin:
    def __init__(self):

        self.win = tk.Tk()
        self.win.title('Canvas Example')

        self.cv = tk.Canvas(self.win, width=300, height=200)

        self.cv.create_line(0,0,150,150, fill='blue')

        self.cv.create_rectangle(30,30,170,170, outline='red')

        self.cv.create_oval(40,40,160,160, outline='green')

        self.cv.create_text(180,100, font=('Arial,14'),
                            text='Shapes', fill='cyan')

        self.cv.grid()


fWin = FrameWin()
```



Commonly used options include:

lines – arrow (arrow heads), dash, fill (color), and (line) width

rectangles – dash (line), fill (color), outline (color), and (line) width

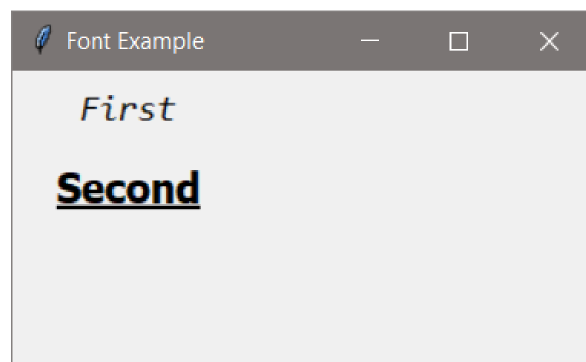
ovals – dash (line), fill (color), outline (color), and (line) width

text – anchor (positioning), fill (color), font, and justify

The font used with *create_text* can be customized using anchor and justify for positioning, as well as the tkinter *font* module which provides the ability to assign a font family (Courier, Consolas, etc.), a size in points, a weight (bold and normal), slant (italic), underline, and overstrike (crossed out text). The example below creates two fonts and assigns each to a different line of text.

Ex. 10.14 – Font example

```
import tkinter as tk
import tkinter.font

class FontWin:
    def __init__(self):

        self.win = tk.Tk()
        self.win.title('Font Example')

        self.cv = tk.Canvas(self.win,width=300,height=150)

        new_font1 = tk.font.Font(family='Consolas', size=14,
                                 slant='italic')


        new_font2 = tk.font.Font(family='Tahoma', size=16,
                                  weight='bold', underline='1')

        self.cv.create_text(60,20,text='First', font=new_font1)
        self.cv.create_text(60,60,text='Second', font=new_font2)

        self.cv.pack()


fWin = FontWin()
```

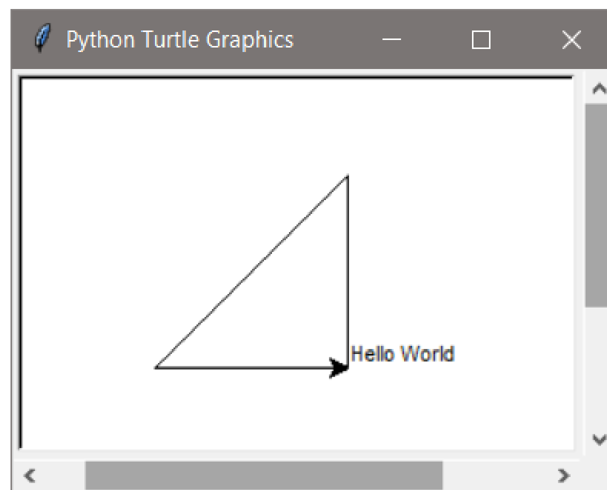


Chapter 8 covered the use of matplotlib for plotting, and creating charts. Drawing can also be accomplished with tkinter or the turtle graphics package that is installed with Python.

Turtle Graphics

The turtle package is a pre-installed Python library that enables drawing pictures and shapes. The "pen" used for drawing is called the turtle although it is shaped like an arrow. Turtle graphics are mainly used to introduce children to computers and a fun way to draw shapes and learn computer commands. The following program imports the turtle package and draws a triangle using three lines, and adds a line of text.

```
import turtle

turtle.goto(0,100)
turtle.goto(-100,0)
turtle.goto(0,0)
turtle.write(' Hello World')
```

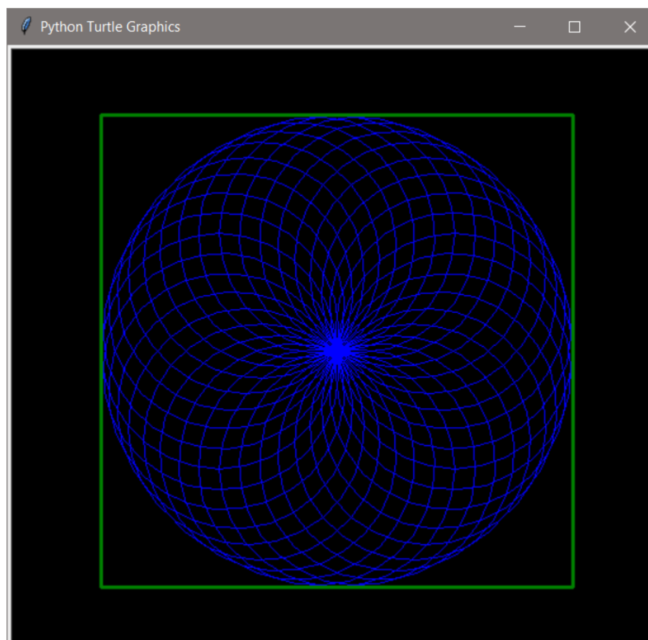


Turtle graphics use a coordinate system with 0, 0 at the center of the turtle output window. The turtle begins at these coordinates and is pointing east which is 0 degrees for the turtle, with 90 degrees being north, 180 degrees being west, and 270 degrees being south. By default the turtle pen is down (writing position), but can be lifted to move it to another location without drawing a line. There are options for pen color, background color, changing the angle and pen size, and various shape commands. The animation speed can be set so that the drawing is completed slowly instead of all at once.

The following example draws 36 circles while changing the angle left 10 degrees, and surrounds the shape with a green square.

Ex. 10.15 – Turtle graphics example

```
import turtle

# set background and pen colors
turtle.bgcolor('black')
turtle.pencolor('blue')

# Draw 36 circles
for x in range(36):
    turtle.circle(100)
    turtle.left(10)

turtle.hideturtle()
turtle.penup()

# change the pen color and width
turtle.pencolor('green')
turtle.pensize(3)

# draw a green square around the shape
turtle.goto(-200,-200)
turtle.pendown()
turtle.goto(-200,200)
turtle.goto(200,200)
turtle.goto(200,-200)
turtle.goto(-200,-200)
```



Chapter 10 Review Questions

1. GUIs are _________ driven by _________ input.
2. The ___________ module in Python can be used to develop GUI programs.
3. The GUI control used to display one line of text is a ___________.
4. The GUI control used to obtain a single line of user input from the keyboard is an ___________.
5. A function or method that is called when an event occurs is referred to as a ________function.
6. The three geometry managers provided by tkinter are: _________, ________, and ________.
7. Mutually exclusive means that _________ item in a group can be selected.
8. The Grid Geometry Manager can be used to ___________ controls in a GUI using rows and columns.
9. _________ is the method used to obtain input from an entry control.
10. The button option that assigns a callback function is the _________ option.
11. The small window used to alert a user to an issue in a GUI application is called a _________________.
12. The _________ object and _________ object provide an immediate update to the controls they are assigned to.
13. _________ and ___________ are mutually exclusive controls and users can select only one, whereas _________ are not and multiple selections can be made.

Chapter 10 Short Answer Exercises

1. Write a statement that creates a label called label1 for self.win with the text "Python is fun!"

2. Write a statement that creates an entry control called user_input for self.win that allows for 10 characters of input.

3. What does the following statement accomplish/allow?

```
import tkinter as tk
```

4. In the following compute button creation, what is the text on the button?

```
self.btn1 = tk.button(text='Compute', font=('Arial',14),
                      command = self.compute_value)
```

5. In the following compute button creation, what is the callback function?

```
self.btn1 = tk.button(text='Compute', font=('Arial',14),
                      command = self.compute_value)
```

6. What does the following statement accomplish?

```
my_string = self.my_entry.get()
```

7. Write a statement for a show message dialog box with the title "Error" and the text "An error has occurred".

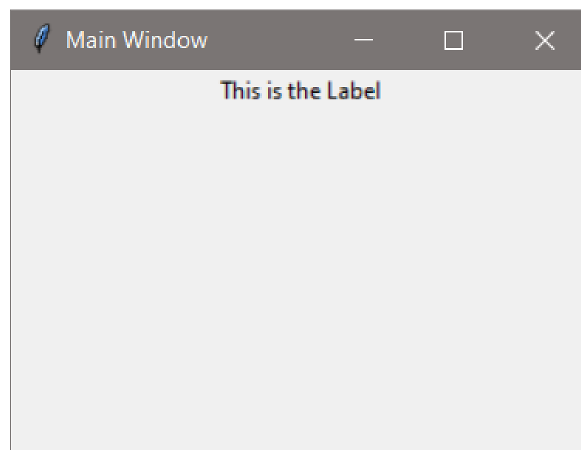
8. What does the following statement accomplish?

```
self.win.columnconfigure(1, minsize=50)
```

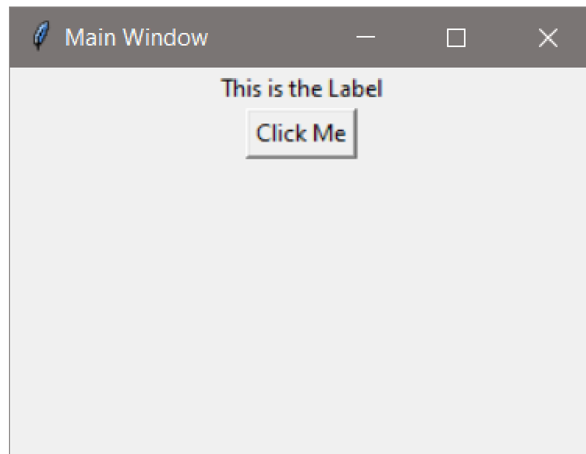
9. Where are the coordinates (0, 0) located on a tkinter Canvas?

Chapter 10 Programming Exercises

1. Implement a class definition for a MyWin Class that generates the window below which is 300w x 200h with a title "Main Window" and a label that says "This is the Label". Pack can be used to display the label.

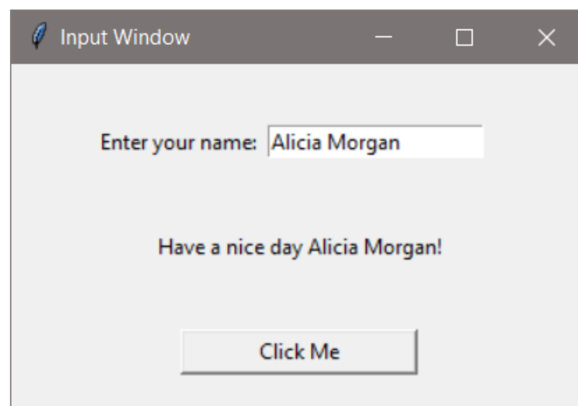


2. Using the MyWin Class from #1, add a button to the window with the word "Click Me" on the button. Pack can be used to display the button.



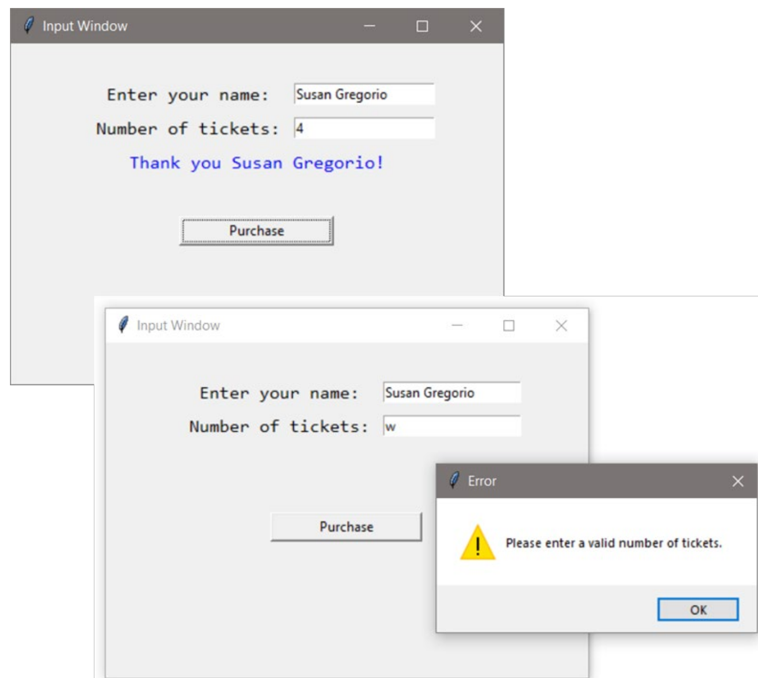
3. Implement a class InputWin that creates a GUI and uses the grid geometry manager with six (6) rows configured with a minimum size of 30 pixels and three (3) columns configured with a minimum size of 50 pixels. Add a label that prompts the user to input their name into an entry control that is 20 characters wide. Add a button (width=18) that obtains the input from the entry control using a command that calls a function that updates a second label and displays "Have a nice day " and the name that was entered. Position the control as follows:

Prompt Label	row 1	column 0
Entry	row 1	column 1
Output Label	row 3	column 0, columnspan=3
Button	row 5	column 0, columnspan=3

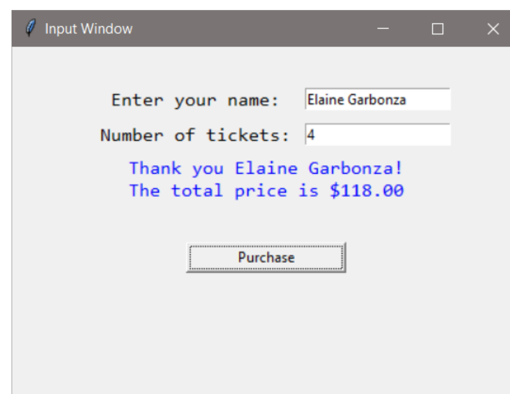


4. Implement a class TheaterWin that creates the GUI in program #3 with the same row and column configurations. Add another label and entry control and request a number of tickets to purchase. The button text should be changed to "Purchase", and the output label text should be changed to "Thank you " and the name entered. Modify the font for the labels to be "Consolas" 12 and the output label text to blue.

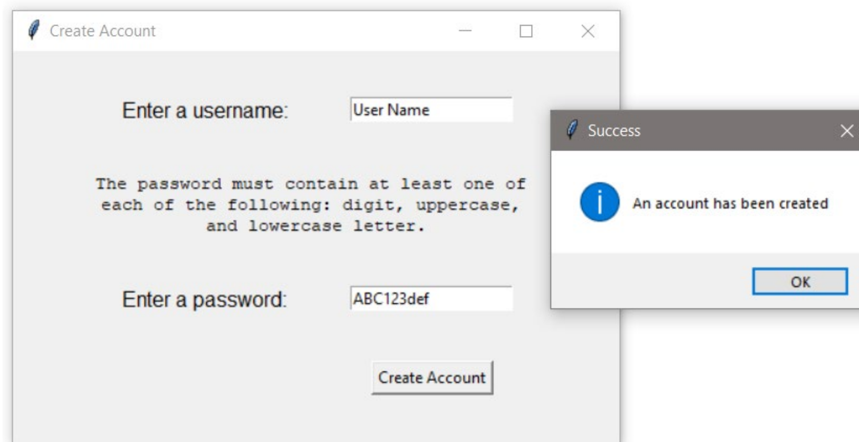
   If the number of tickets entered is not a positive integer, an error dialog box should appear (import tkinter.messagebox). The error dialog box text should alert the user to the issue.



5. Modify program #4 to display the Total cost based on a ticket price of $29.50 as shown below. Note the dollar sign and two decimal places in the output.



6. Implement a Create Account class with a GUI that obtains a user name and password from the user and validates the password (at least 9 characters, at least one digit, upper, and one lower case letter). If the password is valid, display "An account has been created." in a dialog box, otherwise use a dialog box to handle the error. The title on the window should be "Create Account", and the window should display the password requirements to the user.



Chapter 10 Programming Challenges

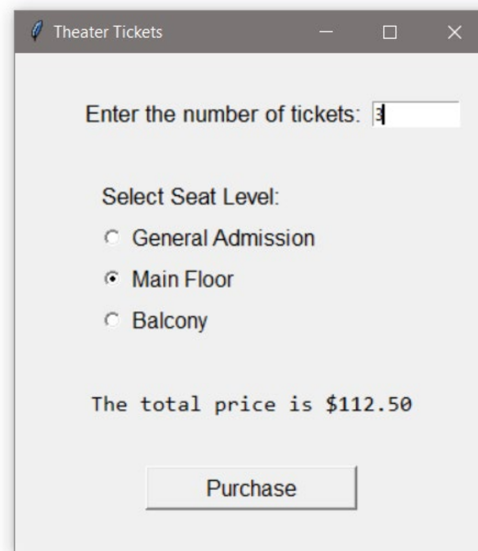
#1 – Theater Ticket GUI with Seat Level Pricing

Design and implement a Class for a Theater GUI that obtains the number of tickets being purchased from the user and allows seat selection: General Admission, Main Floor, and Balcony. The program should use an option list or radio buttons for seat selection, and display an error dialog if the number of seats entered is invalid. Use a label to display the total price based upon the number of tickets sold and the cost for seating as shown below.

General Admission	$18.50
Main Floor	$37.50
Balcony	$26.00

Reminders: the row and column configure values can be assigned individually and as need to accommodate the GUI design. The "sticky" option allows

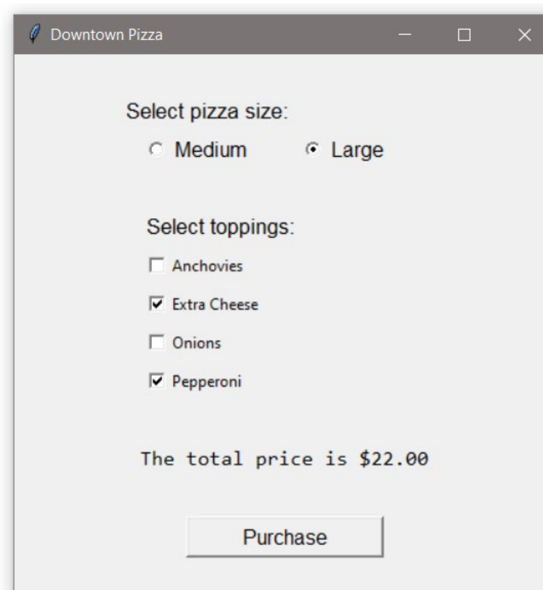
further alignment using “N”, “S”, “E”, and “W”. Radio buttons can use an IntVar or StringVar assignment.



#2 – Pizza Size and Topping GUI

Design and implement a GUI for the Downtown Pizza shop that obtains an order for a pizza by size and topping. The size selection should be done with radio buttons or an option list, and the topping selection with check boxes to accommodate multiple selections. A purchase button will compute the price and display it to the user based upon the prices below.

Medium	$12.50
Large	$15.50
Anchovies	$2.50
X-cheese	$3.00
Onions	$2.50
Pepperoni	$3.50

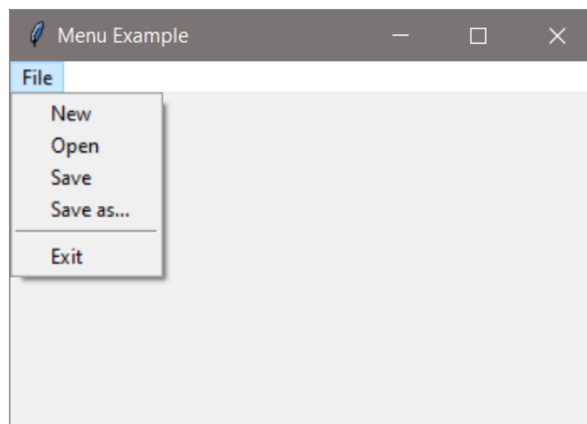


Chapter 11

Menus, Images, and Windows

Menus

Adding a drop-down menu to a window can provide program-level operations such as file handling, and allow users to Open, Save, Save As, and Exit the program. The menu rests on the window frame, and drops down to reveal the options when it is clicked. Multiple drop-down menus can be added.



The Python drop-down is created by assigning a ***Menu()***. The items listed on the menu are added using ***add_command*** and a function to respond to the selection much the way the callback function is assigned to a button. There is a separator

that can be added between selections using ***add_separator*** as shown in the display above and included in the code below.

Ex. 11.1 – Menu (drop-down) example

```
class MenuWin:
    def __init__(self):

        self.main_win = tk.Tk()
        self.main_win.title("Menu Example")
        self.main_win.minsize(width=350,height=180)

        self.menubar = Menu()
        self.file_menu = Menu(self.menubar, tearoff=0)
        self.file_menu.add_command(label="New",
                                   command=self.new_file)
        self.file_menu.add_command(label="Open",
                                   command=self.open_file)
        self.file_menu.add_command(label="Save",
                                   command=self.save_file)
        self.file_menu.add_command(label="Save as...",
                                   command=self.save_as_file)
        self.file_menu.add_separator()
        self.file_menu.add_command(label="Exit",
                            command=self.main_win.destroy)

        self.menubar.add_cascade(label="File",
                                 menu=self.file_menu)

        self.main_win.config(menu=self.menubar)

        tk.mainloop()

    def new_file(self):
        print("New clicked.")

    def open_file(self):
        print("Open clicked.")

    def save_file(self):
        print("Save clicked.")

    def save_as_file(self):
        print("Save As clicked.")
```

In the example, the menu is declared as menubar which is then the first argument when creating the file_menu. The *add_command* method is used for each of the drop-down selections, and *add_separator* places the line on the menu. Items are positioned on the menu in the order they are added. The ***add_cascade*** method provides the label for the menu and places the file_menu on the menubar. To react to the individual menu items, separate functions are assigned

except in the case of the item to "Exit" the program. The ***destroy*** method closes the window and ends the program. The file handling operations are simplified with tkinter file dialogs covered in an earlier chapter.

Images

Adding an image to a window using the tkinter module is similar to other components. Python's Tkinter has a ***PhotoImage*** class for handling images that supports the GIF, PGM/PPM, and PNG formats. If other file formats are needed, the Python Image Library (PIL) contains classes that can handle over 30 formats and convert them to Tkinter compatible image objects. The image file can be located with the program files, which is the default directory, or a path to the file can be used.

To use a *PhotoImage* instance, the method must be imported from the tkinter module as shown here above the import for tkinter to import the method.

```
from tkinter import PhotoImage   # import PhotoImage
import tkinter as tk             # imports tkinter as tk
```

As a technical note, some programmers may use a wildcard import statement with an asterisk as shown here which imports the entire tkinter module.

```
from tkinter import *    # NOT RECOMMENDED
```

It is best to avoid using ***wildcard import statements*** when multiple modules are imported because name clashes can occur when modules have functions or classes with the same name. To avoid using "star" imports as they are often called, run the code. If Python says "Tk" is not defined, then add "from tkinter import Tk, and run the program again. If it then says PhotoImage not found, then add a comma after Tk, and add PhotoImage. Keep doing this to ensure that only the required parts of the library are added.

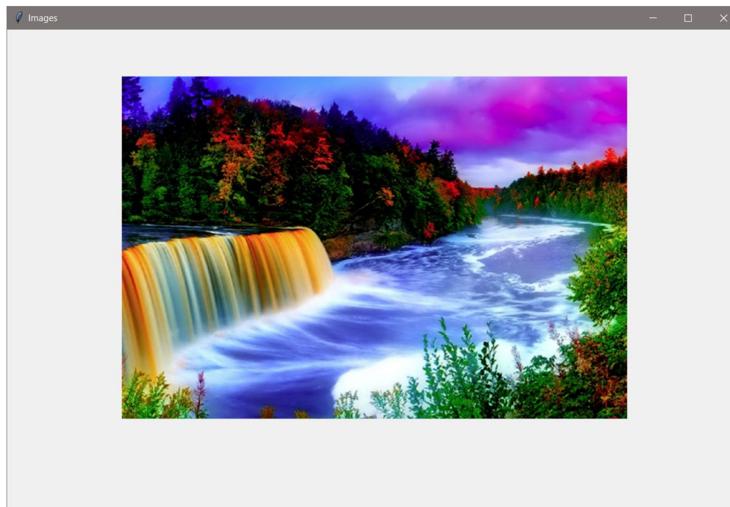
```
from tkinter import Tk, PhotoImage
```

A reference to the image must be retained or Python's interpreter could eliminate it even if it is being displayed. The code to apply an image consists of three lines. The first line assigns the file to a *PhotoImage* object, the second places the image

on a label (a canvas or frame can also be used), and the third retains a reference to the image. The fourth line below positions the image using the grid geometry manager.

Ex. 11.2 – adding an image to a label

```
photo = PhotoImage(file="WaterFall2.PNG")
self.main_win.image_label = tk.Label(image=photo)
self.main_win.image_label.image=photo   # retain a reference
self.main_win.image_label.grid(row=3, column=1)
```



Centering a Window in the Display Area

When windows are created they are displayed in the top-left corner of the display monitor. A simple method for centering uses ***geometry*** to change what tkinter sees as the top-left corner. The statement below provides window dimensions and the window placement information in pixels.

```
# Geometry Arguments - no spaces
#                 window size 400 x 300
#                 + x position + y position

self.main_win.geometry('400x300+500+300')
```

Note the use of quotes around the expression and the use of “x” for the dimensions and “+” for the window placement. There cannot be any spaces in the expression. The window in the example is 400 by 300 and is positioned 500 pixels from the left edge of the display area and 300 pixels down from the top of the display area. In display graphics, 0, 0 is the top left corner. Note that these placement values are display resolution dependent. On a computer with a different resolution setting, the window would not be centered.

For accurate centering, the display area’s width and height are obtained and the window size is subtracted. These values are then divided by two, and the arguments are cast to integers and then passed to geometry as a string using the format required for *geometry*. Note the “x” after the first “%d” in the formatting. The geometry arguments are the top left corner of the window being centered. This is the reason for dividing by two.

Ex. 11.3 – accurate centering of a window in the display area

```
# window size - width = 400, height = 300

# Use screenwidth and screenheight to calculate centering

x_Left = int((self.main_win.winfo_screenwidth()- 400)/2)
y_Top = int((self.main_win.winfo_screenheight()- 300)/2)

self.main_win.geometry('%dx%d+%d+%d' %(400,300,x_Left,y_Top))
```

Freezing Window Size

The controls on a window are often positioned using a specific height and width for the window. If a user stretches the window in any direction, the controls may move and ruin the desired arrangement. To avoid this situation, the ***resizable*** function can be set to false for height and width which prevents resizing by the user. Two versions are shown.

Ex. 11.4 – freezing window size

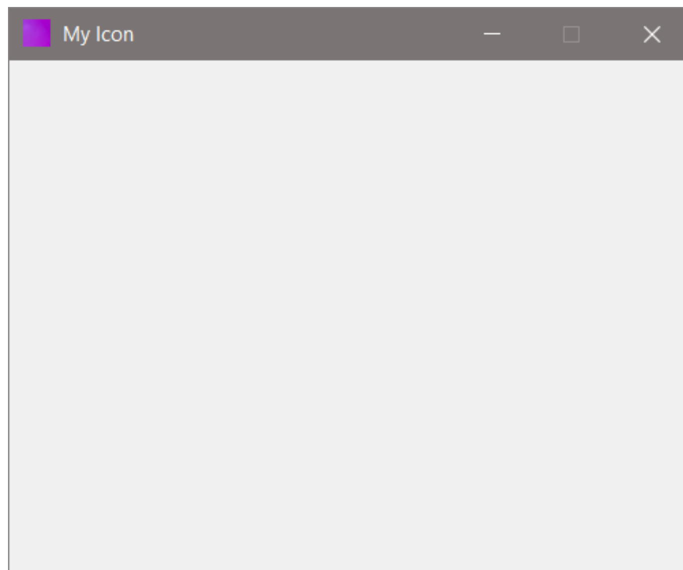
```
self.main_win.resizable(height = False, width = False)

self.main_win.resizable(False,False)
```

Window Icons

Changing the icon requires an image in the .ico format, and using the *iconbitmap()* method. There are software packages such as GIMP and others available online for creating icons or converting other image formats. Once the icon is created, the filename or path is passed to the method.

Ex. 11.5 – changing the window icon

```
self.main_win.iconbitmap('myIcon.ico')
```

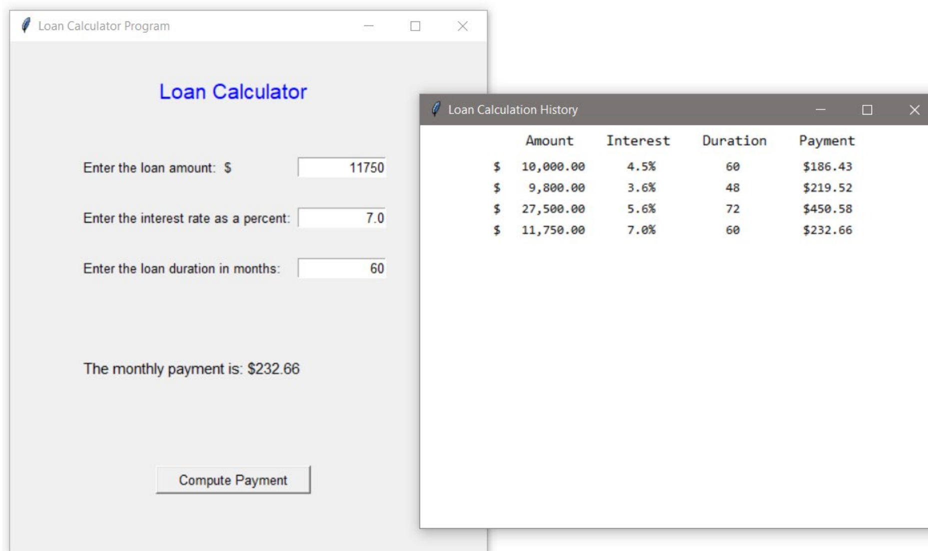


Another method for changing the icon is ***iconphoto()*** which accepts other image formats. The first argument indicates that only this window is to get the icon. Note that tkinter's PhotoImage method would need to be imported.

```
self.main_win.iconphoto(False, t.PhotoImage(file='/path/icon.png'))
```

Updating a Second Window

Many GUI programs display data to the user as it is being computed and display the previous results for comparison. This may be often in a second display window. As an example, consider a program that computes a value when new data is entered and a second window that displays the historical results.



The example is a GUI program that computes a loan payment based on user input of the loan amount, interest rate, and duration of the loan. Entry controls on the main window obtain the input and a button click calls a compute function to compute the monthly payment amount. A StringVar is used to update the output label on the main interface with the monthly payment amount (in a previous example, the config method was used). Recall that whenever a StringVar is changed, the control that it is assigned to is immediately updated. A second window displays the computation history (this will be addressed later in the example).

The GUI design includes three prompt labels, three entry controls for the data, and a "Compute Payment" button. The code to create and assign the StringVar is shown here including the initial text using the *set()* method.

Ex. 11.6 – creating and assigning a StringVar

```
self.pymt_var = tk.StringVar()

self.payment_label=tk.Label(textvariable=self.pymt_var,\
                            font=("Arial",11))

self.payment_label.grid(row=6,column=2,columnspan=2,sticky='W')
self.pymt_var.set('The monthly payment is: ')
```

The payment amount is displayed on the main window by modifying the StringVar in the function that computes the payment amount. A change to the StringVar using *set()* automatically updates the label.

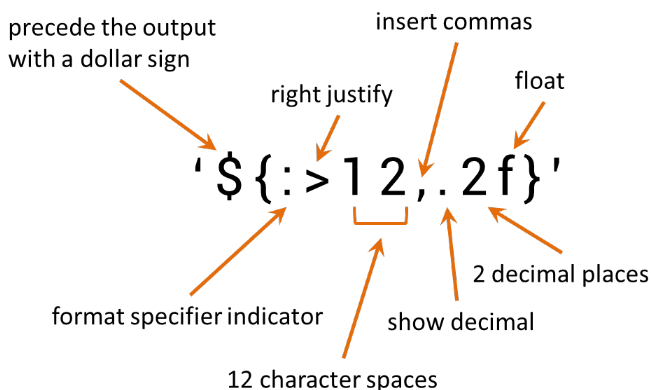
```
# Update the stringvar on the interface
new_str = 'The monthly payment is: $' + str(format(mp,'.2f'))
self.pymt_var.set(new_str)
```

To display the history of loan computations, the second window is created when the first payment is computed and includes the column headers. Once the payment amount has been calculated, the data is formatted for display. The new Python formatting types use placeholders (braces) and additional specifiers. The example below formats the loan amount for the display window. Note that a dollar sign precedes the opening brace and the entire expression is inside quotes followed by format.

Ex. 11.7 – data format specifiers

```
fltA = float(amt)          # convert the value to float
fltA_string = '${:>12,.2f}'.format(fltA)
```

An expanded view of the formatting follows.



Once the data is formatted, a new label is created and placed on the next row of the output display (*rc* increments the row). This provides the historical data.

```
self.dd_win.data_lbl = tk.Label(self.dd_win,
                          font=('Consolas',10), bg='white', \
                             text= fltA_string + fltI_string +
                             intD_string + fltMP_string)

self.dd_win.data_lbl.grid(row=rc,column=1,sticky='W',columnspan=4)
```

Reading data from a file and displaying it could be handled in a similar way. A loop would read from the file and display the data by creating the labels as the

values are read. They could also be read into a list or tuple and again a loop would be used to create labels.

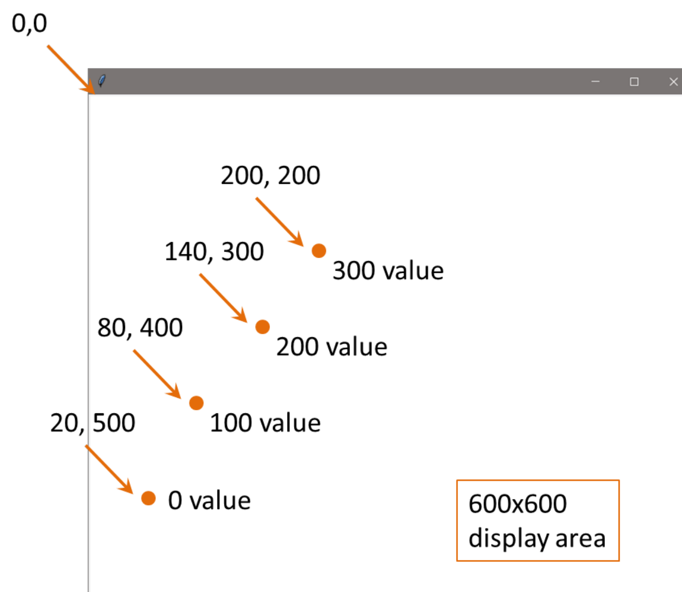
Plotting to a Second Window

Data can also be plotted to a display window by drawing on a canvas. The following example computes a fahrenheit temperature from a Celsius input and plots both values in a separate display. The window owning the canvas is the first argument when it is created.

Ex. 11.8 – plotting on a canvas in a second window

```
self.plot_win = tk.Tk()
self.plot_win.minsize(600,600)
self.plot_win.title('Temperature Conversion Chart')

self.canvas = tk.Canvas(self.plot_win,width=720,height=600,
                        bg='white')
self.canvas.create_text(350, 50, font='Helvetica 16 bold',
                        text='Temperature Conversions')
self.canvas.pack()
```

A consideration when drawing on a canvas is the x, y coordinate system and working relative to the top-left corner which is 0, 0. To move something upward, the "baseline" must be determined first down from the top. An example follows.

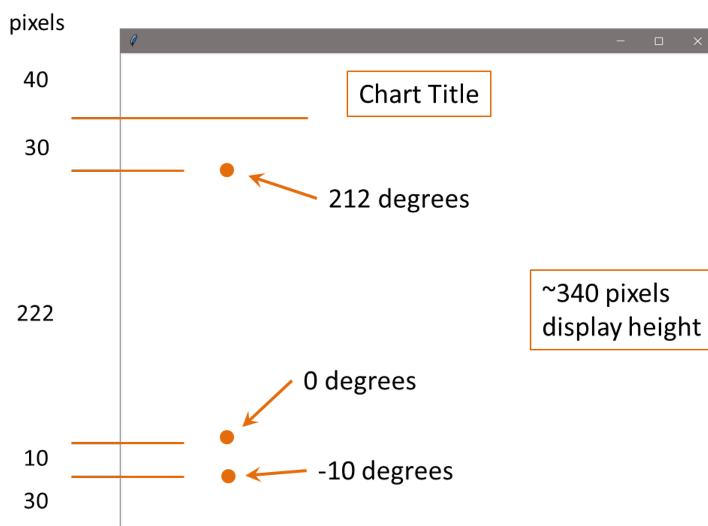


The image above is an example sketch to plot four circles representing some values (0, 100, 200, 300). The value for each "y" coordinate must be related to the lowest value and the size of the window. The sketch contains a 600x600 window, the circle for the lowest value is drawn at the "y" coordinate of 500 (500 pixels down from the top), and the others are relative to that point. The display size and positioning of the lowest point determines the "y" coordinates for the other data points.

The Celsius to Fahrenheit example below uses the same algorithm by first determining an optimum size for the window based upon the output range of values, the scaling factor (pixels), and moves the "x" coordinate for each set of values computed. The program accepts a range of Celsius inputs from -10 to 100 degrees Celsius so there are 110 Celsius data points. The conversion range for this set of values would be 14 to 212 degrees Fahrenheit, so there are 198 Fahrenheit data points. The total range to be plotted is then -10 to 212 which is 222 data points.

Celsius range	-10 to 100
Fahrenheit range	14 to 212
Total range	-10 to 212 = 222 data points

Consider that one pixel could represent one degree, so the window needs to be at least 222 pixels in height. Consider that a title for the chart and spacing requires additional height. A design sketch makes it easier to determine locations.



The starting (lowest) point for reference is a "y" coordinate of 290 (down) and the others are relative to that point. The Celsius plot statement is shown below.

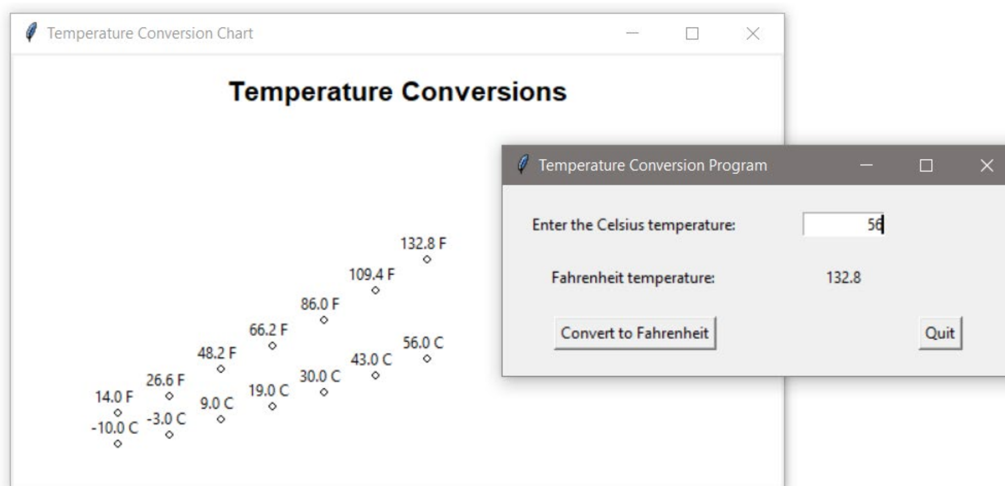
```
self.canvas.create_oval(self.data_num*40, 290-self.celsius, \
                        5 + self.data_num*40, 295-self.celsius)
```

The first argument is a counter called data_num that is used for the "x" coordinate to move each new computation to the right 40 pixels. The second is the relative point of 290 (down) with the Celsius temperature subtracted (up). The next two arguments are the "x2" and "y2" cordinates for the oval.

All four plotting statements are shown below. The text is placed 5 pixels above the oval as an offset.

```
self.canvas.create_text(self.data_num*40, 280-self.fahrenheit, \
                        text= str(self.fahrenheit)+ ' F')

self.canvas.create_oval(self.data_num*40, 290-self.fahrenheit, \
                        5 + self.data_num*40, 295-self.fahrenheit)

self.canvas.create_text(self.data_num*40, 280-self.celsius, \
                        text=str(self.celsius)+ ' C')

self.canvas.create_oval(self.data_num*40, 290-self.celsius, \
                        5 + self.data_num*40, 295-self.celsius)
```

The display with sample output is shown below.

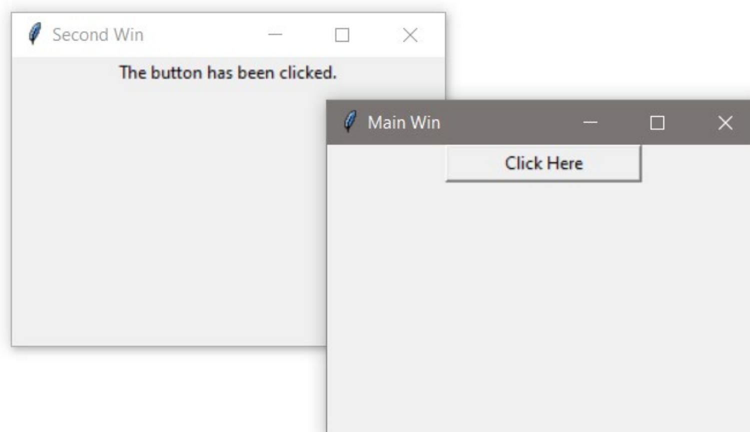


Chapter 8 included creating charts using matplotlib, and there are other charting packages as well, but basic charts can be created using the tkinter module.

Interacting with a Second (Toplevel) Window

Windows created in addition to the main window, are referred to as ***Toplevel*** windows. The following is a simple example that creates a main window with a button, and a second window that reacts to the button click. The change is handled through a StringVar. Note that the second window is declared as a tk.Toplevel, and that the StringVar is not assigned to a window.

```
class TwoWins:
    def __init__(self):

        main_win = tk.Tk()
        main_win.title('Main Win')
        main_win.geometry('300x200')
        main_win.btn = tk.Button(text='Click Here', \
                                 width=18,command=self.update)
        main_win.btn.pack()


        sec_win = tk.Toplevel()
        sec_win.title('Second Win')
        sec_win.geometry('300x200')

        self.update_var = tk.StringVar()
        self.update_var.set('The label')

        sec_win.lbl = tk.Label(sec_win,
                               textvariable=self.update_var)
        sec_win.lbl.pack()

        tk.mainloop()

    def update(self):
        self.update_var.set('The button has been clicked.')

iWin = TwoWins()
```



Closing Windows

When a user exits a program either by clicking on a "Quit" button (if provided) or by clicking on the "X" at the top right corner of the window, the program should end. This includes closing any windows created by the program. There are several ways of handling this and a few examples are necessary. The first example includes a quit button that calls a function assigned to the command. The function uses the destroy method to end the program (and the main loop), since the destroy method cannot be assigned directly to the command.

Ex. 11.9 – command assigned function for closing a window

```
    self.quit_button = tk.Button(text = 'Quit', width=12,
                                 command = self.close_prog)
    self.quit_button.grid(row=1,column=1)


    tk.mainloop()

def close_prog(self):
    self.main_win.destroy()
```

Clicking the "X" on the window would also end the program, however there may be other statements to execute when the program ends and the function provides a way to execute them.

The next example adds a second window to the close function, but if the user clicks on the "X" of either window, only that window is destroyed. The other window would still be displayed.

```
def close_prog(self):
    self.main_win.destroy()
    self.second_win.destroy()
```

For the next example, an understanding of a lambda expression is needed.

Lambda Expressions

A lambda expression is an inline function with no name. Lambda expressions are not necessary, but in some situations, they make writing the code easier. When a function is simple and will be called only once, a lambda expression

makes sense. It can be anonymous (no name) and defined where it will execute. One frequent use of a lambda is in programming "callbacks" for the command assigned to a button. A button requires a function object to be assigned to the command. A way of handling this is to have the command be a call to a function and then to have that function perform the operation as shown here.

Ex. 11.10 – button command for print function

```
self.new_button1 = tk.Button(text='Button 1', width=16,
                             command=self.on_click)
def on_click(self):
    print('Button selected')
```

In the above example, the print command cannot be assigned directly to the button. The command must call the function *on_click* which then handles the print function. Using a lambda function would eliminate the call to the function as shown below. The keyword lambda is followed by a colon and the function.

Ex. 11.11 – lambda button command

```
self.new_button2 = tk.Button(text='Button 2', width=16,
                       command=lambda : print('Lambda !'))
```

The earlier example that used a function that called destroy to close the window can be rewritten using a lambda expression as well.

```
self.quit_button = tk.Button(text = 'Quit', width=12,
                   command = lambda:self.main_win.destroy())
```

The final example uses protocol and the event of the window closing so that when a user clicks on the "X", the program has control and can execute other statements like closing other windows. Below, both windows react to being closed by the system and call the function that closes them both.

```
self.main_win.protocol("WM_DELETE_WINDOW", self.close_prog)
self.sec_win.protocol("WM_DELETE_WINDOW", self.close_prog)
```

```
def close_prog(self):
    self.main_win.destroy()
    self.sec_win.destroy()
```

Chapter 11 Review Questions

1. A drop-down menu on a window can provide _____________ operations.
2. The ______________ method closes a window and ends the program.
3. An asterisk with an import statement is referred to as a ________________.
4. The _____________ method is used to center a window in the display area.
5. The ____________ function can be used to prevent a window from being resized.
6. The __________ file format is used with the iconbitmap() method to change the window icon.
7. When a StringVar is assigned to a control, any change to the StringVar value immediately __________ the control.
8. When plotting on a canvas, the 0, 0 coordinates are located at the __________ of the canvas.
9. A ________ expression is an inline function with no name.

Chapter 11 Short Answer Exercises

1. What function is assigned to the “Exit” menu item in the following statement?

```
self.file_menu.add_command(label=”Exit”,
                           command=self.main_win.destroy( ))
```

2. Where will the following statement place the window when it is created?

```
self.main_win.geometry(‘300x300+100+200’)
```

3. What is the size of the window in the following statement?

```
self.main_win.geometry(‘300x300+100+200’)
```

4. Where will the following statement place the window when it is created?

```
x_crd = int((self.main_win.winfo_screenwidth() – 300)/2)
x_crd = int((self.main_win.winfo_screenheight() – 300)/2)
self.main_win.geometry(‘%dx%d+%d+%d’, %(300,300,x_crd,y_crd’)
```

5. What does the following statement accomplish?

```
self.main_win.resizable(False,False)
```

6. In the following expression, how much character space is allocated in the formatting?

```
value_string = '{:>10}'.format(value)
```

7. In the following expression, what is the effect of the greater than character?

```
value_string = '{:>10}'.format(value)
```

8. In the following expression, why is the word lambda included?

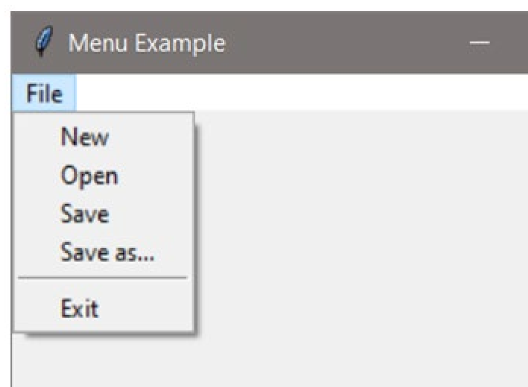
```
tk.Button(text='Click', command = lambda : print('Click')
```

9. What window is the "owner" of the canvas in the following statment?

```
self.canvas = tkCanvas(self.plot_win, width=500, height=500)
```

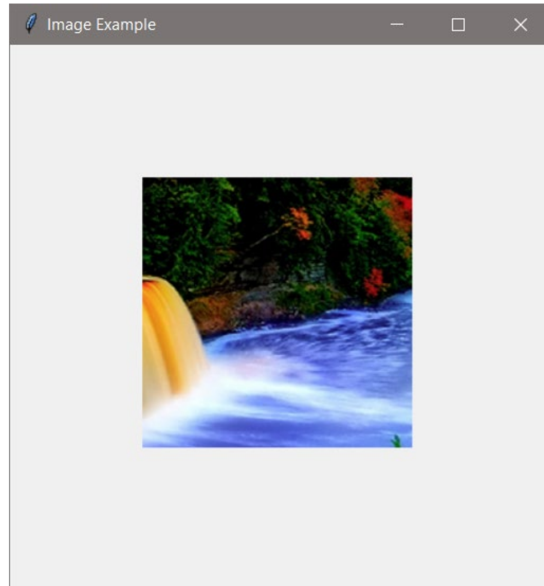
Chapter 11 Programming Exercises

1. Implement a window with the title and menu shown below. When the menu items are clicked, call a function that prints that an item was clicked. A single function can be used.

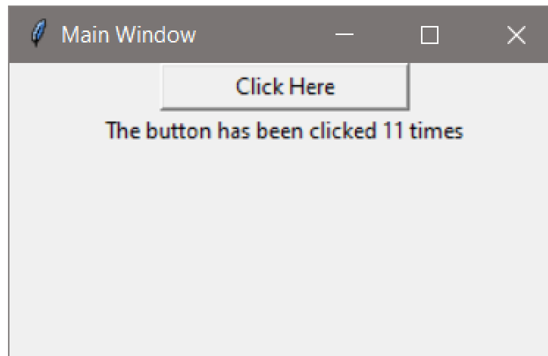


2. Implement a 400x400 non-resizable window that is centered in the display area when the program runs.

3. Implement a window with an image. The window should be 410x410 and the image 200x200. Center the image in the window.



4. Implement a program with a window that has a button that updates a label that displays how many times the button was clicked. Use a StringVar in the solution.



5. Implement a window with a button that creates a second window when it is clicked.

6. Implement a two window program. The first window will have a button that updates a label on the second window and displays how many times the button was clicked. Use a StringVar in the solution. The second window should be a Toplevel window.

7. Implement a window that is 300 x 300 with a canvas, and plot the text below at those coordinates. Make the font for the text Consolas, 12, and bold.

50, 50	250, 50	150,150
50, 250	250, 250	

Chapter 11 Programming Challenges

#1 – Two-window Close Both

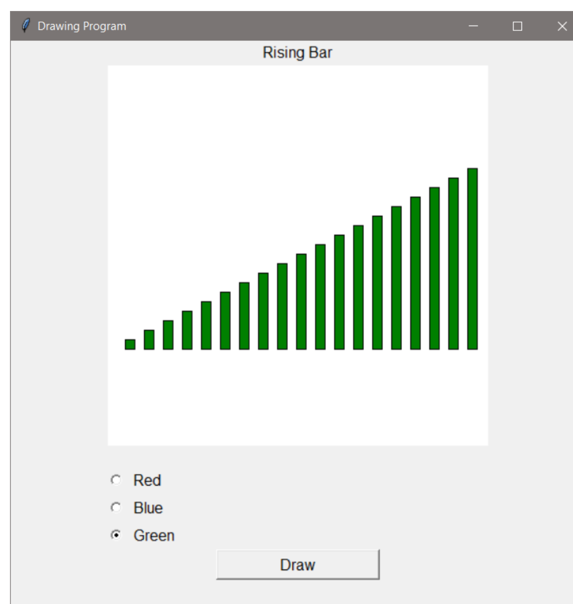
Design and implement a program with two windows. When either window is closed the other window should be destroyed and the program should end.

#2 – Display and Plot Values

Design and implement a GUI program that allows the user to input a radius (in pixels) for a circle that is drawn on a canvas in a second window. The circle should be centered in the window.

#3 – Draw Rising Bars

Implement a 600x600 window with a 400x400 canvas with a background, three (3) radio buttons that select a color, and a "Draw" button. When the button is clicked, draw a rising set of 19 bars in the color selected.



Appendix A

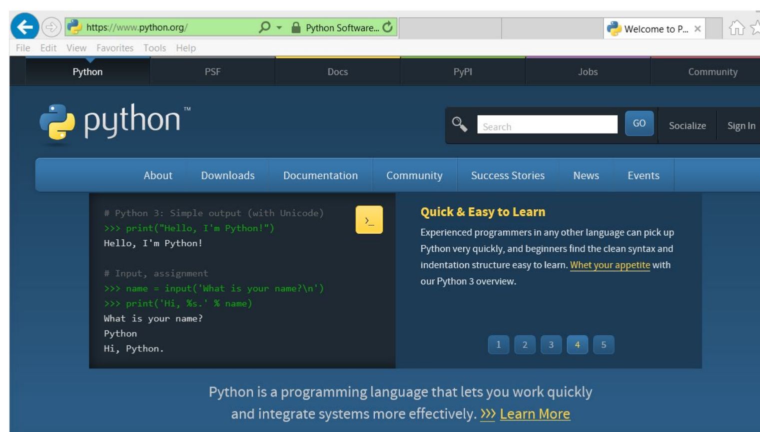
Decimal	Binary	ASCII	Decimal	Binary	ASCII	Decimal	Binary	ASCII
32	0010 0000	space	64	0100 0000	@	96	0110 0000	`
33	0010 0001	!	65	0100 0001	A	97	0110 0001	a
34	0010 0010	“	66	0100 0010	B	98	0110 0010	b
35	0010 0011	#	67	0100 0011	C	99	0110 0011	c
36	0010 0100	$	68	0100 0100	D	100	0110 0100	d
37	0010 0101	%	69	0100 0101	E	101	0110 0101	e
38	0010 0110	&	70	0100 0110	F	102	0110 0110	f
39	0010 0111	‘	71	0100 0111	G	103	0110 0111	g
40	0010 1000	(	72	0100 1000	H	104	0110 1000	h
41	0010 1001	)	73	0100 1001	I	105	0110 1001	i
42	0010 1010	*	74	0100 1010	J	106	0110 1010	j
43	0010 1011	+	75	0100 1011	K	107	0110 1011	k
44	0010 1100	,	76	0100 1100	L	108	0110 1100	l
45	0010 1101	-	77	0100 1101	M	109	0110 1101	m
46	0010 1110	.	78	0100 1110	N	110	0110 1110	n
47	0010 1111	/	79	0100 1111	O	111	0110 1111	o
48	0011 0000	0	80	0101 0000	P	112	0110 0000	p
49	0011 0001	1	81	0101 0001	Q	113	0110 0001	q
50	0011 0010	2	82	0101 0010	R	114	0110 0010	r
51	0011 0011	3	83	0101 0011	S	115	0110 0011	s
52	0011 0100	4	84	0101 0100	T	116	0110 0100	t
53	0011 0101	5	85	0101 0101	U	117	0110 0101	u
54	0011 0110	6	86	0101 0110	V	118	0110 0110	v
55	0011 0111	7	87	0101 0111	W	119	0110 0111	w
56	0011 1000	8	88	0101 1000	X	120	0110 1000	x
57	0011 1001	9	89	0101 1001	Y	121	0110 1001	y
58	0011 1010	:	90	0101 1010	Z	122	0110 1010	z
59	0011 1011	;	91	0101 1011	[	123	0110 1011	{
60	0011 1100	<	92	0101 1100	\	124	0110 1100	|
61	0011 1101	=	93	0101 1101	]	125	0110 1101	}
62	0011 1110	>	94	0101 1110	^	126	0110 1110	~
63	0011 1111	?	95	0101 1111	_	127	0110 1111	DEL

Appendix B

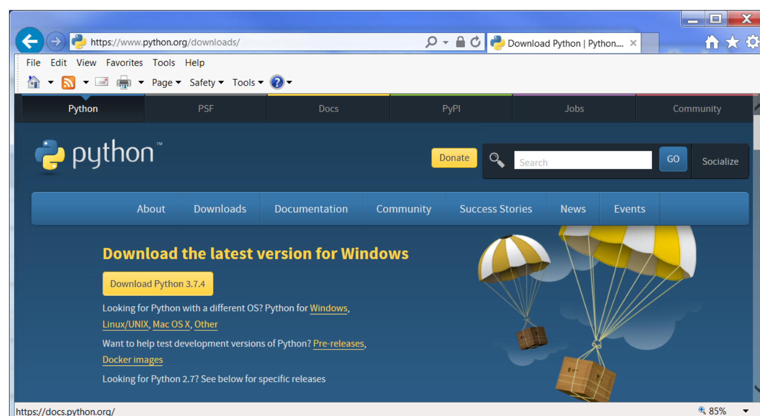
Obtaining Python with IDLE

- Python and IDLE can run on any machine, and can be installed and runs fine on a flash drive
- The IDLE IDE is installed with Python version 3.7.1 and above
- The tkinter module is installed with Python
- Python is available from Python.org   https://www.python.org

Browse to the Python web site shown here and select "Downloads".



In the Downloads window shown below, select the "Download Python 3.7.4" button or select as appropriate for your computer. A later version may be available.



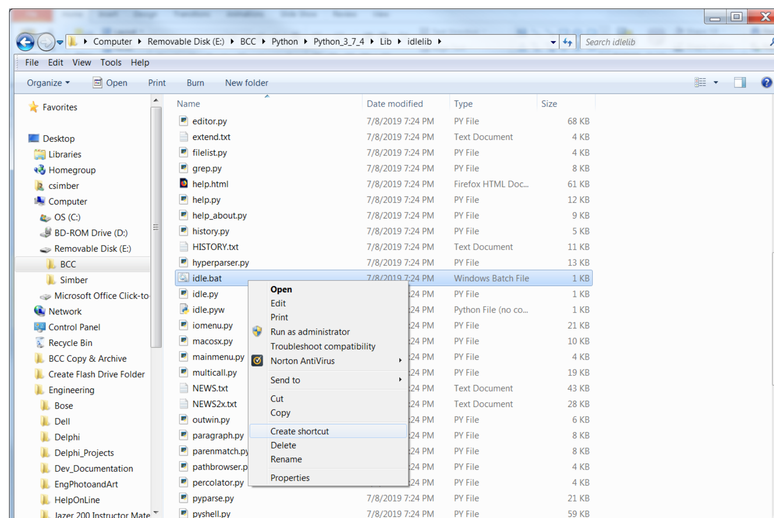
Select the folder where to install the program and download or save it to a folder for installation by double clicking on the file.

Appendix B

The folders and files shown below are installed with Python.

Name	Date modified	Type	Size
DLLs	7/20/2019 3:55 PM	File folder	
Doc	7/20/2019 3:58 PM	File folder	
include	7/20/2019 3:55 PM	File folder	
Lib	7/20/2019 3:55 PM	File folder	
libs	7/20/2019 3:55 PM	File folder	
Scripts	7/20/2019 4:00 PM	File folder	
tcl	7/20/2019 3:58 PM	File folder	
Tools	7/20/2019 3:58 PM	File folder	
LICENSE.txt	7/8/2019 7:33 PM	Text Document	30 KB
NEWS.txt	7/8/2019 7:33 PM	Text Document	676 KB
python.exe	7/8/2019 7:31 PM	Application	96 KB
python-3.7.4.exe	7/20/2019 3:45 PM	Application	25,063 KB
python3.dll	7/8/2019 7:30 PM	Application extens...	58 KB
python37.dll	7/8/2019 7:29 PM	Application extens...	3,522 KB
pythonw.exe	7/8/2019 7:31 PM	Application	94 KB
vcruntime140.dll	7/8/2019 7:24 PM	Application extens...	85 KB

Note: The IDLE executable is not at this level. It is in Lib/idlelib and is launched with idle.bat. Double clicking idle.bat will launch the IDE. To simplify launching IDLE each time, creating a shortcut is recommended. In some cases a desktop shortcut may have been installed when the program was installed.



IDLE is launched by double-clicking: Lib\idlelib\idle.bat

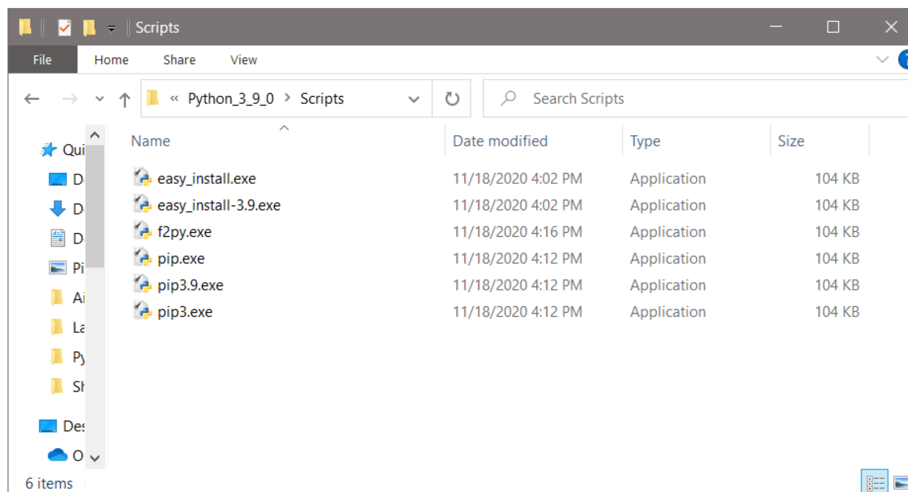
Documentation can be found at:

https://docs.python.org/2/library/idle.html

Appendix C

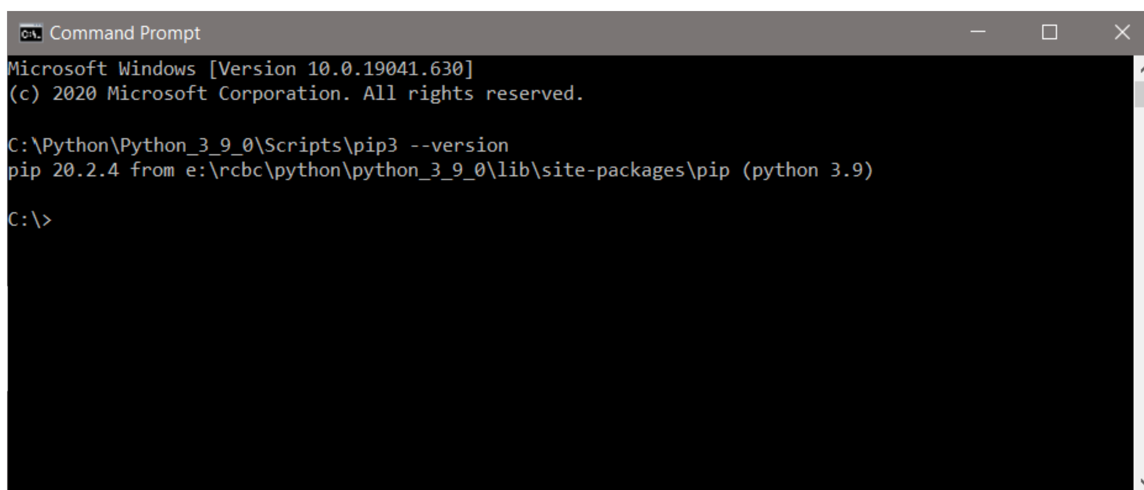
The PIP Module Installer

PIP is already installed if you are using Python 3.4 or above. PIP is a command line program (no GUI), and is run typically from a command prompt. To confirm that PIP is installed, open the Python folder and then Scripts folder which will include PIP files.



PIP can also be verified by opening the Python folder and then the Lib folder, and then the site-packages folder. The PIP installer is run from the Scripts directory. To test for PIP and the proper directory, open a command prompt and type the path to python\Scripts and then pip3 then minus minus version. In this example, Python 3.9.0 is installed (which comes with pip3). The directory is Python\Python_3_9_0.

C:\Python\Python_3_9_\Scripts\pip3 - - version

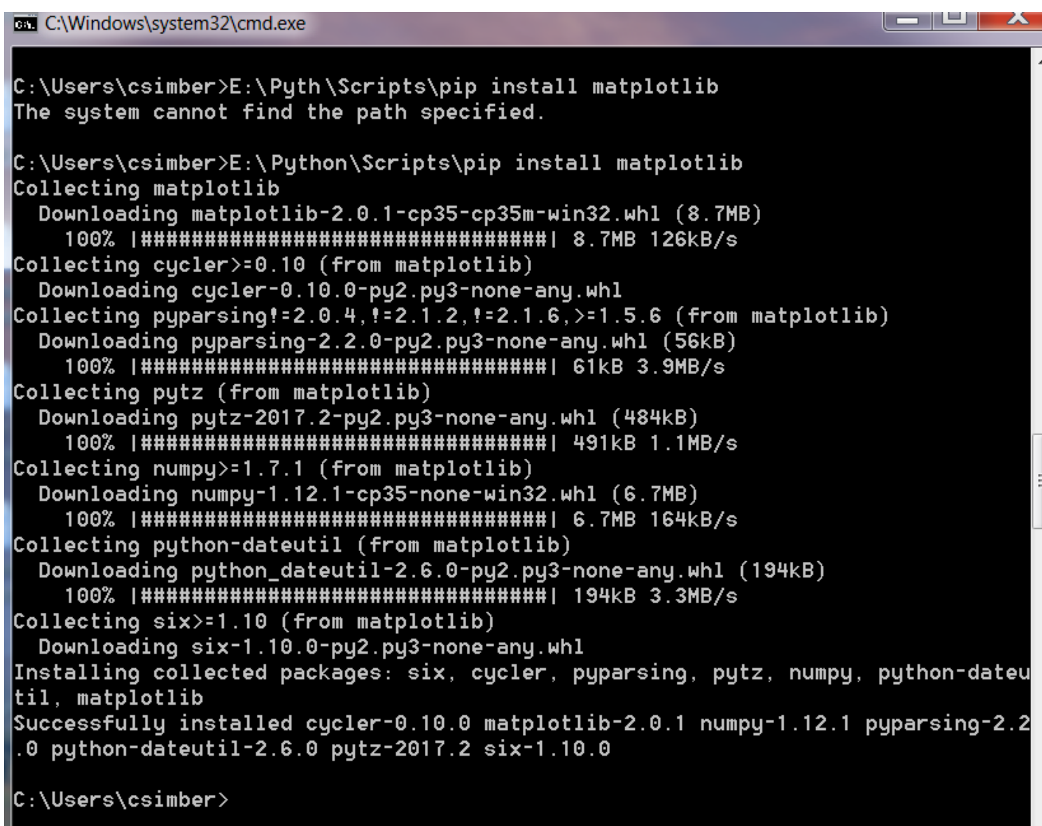


Appendix C

It is important to use complete paths when using PIP, and to be careful of typographical errors. The example below is installing the matplotlib module which is a Python plotting tool. It installs the module using PIP which is being run from the following subdirectory:

E:\Python\Scripts

The first command shown below omitted the letters "on" from "Python" in the command. The second successfully used the installer.



If running from a command console is an issue, PIP can be run through the Python interpreter. The complete User Guide for PIP is available at:

https://pip.pypa.io/en/stable/user_guide/

Appendix D

Links to Helpful Information

Matplotlib:

https://matplotlib.org/

Matplotlib Tutorial:

https://matplotlib.org/3.1.1/tutorials/introductory/pyplot.html

PEP 8 Style Guide for Python Code:

https://www.python.org/dev/peps/pep-0008/

Python Organization: Downloads, Documentation, etc.

https://www.python.org/

Python Tutorial:

https://docs.python.org/3/tutorial/index.html

W3Schools Python Tutorial:

https://www.w3schools.com/python/

Index

A

abstraction 203
access specifiers, classes 198
accessor 199
accumulator, loop 96
acos(x) 130
active (state) 237
add_cascade() 252
add_command() 251
add_separator() 251
Addition (+) operator
    defined 47
    concatenation 40
Agile Development 13
    Process 14
Algebraic Expressions 51
algorithm 11
and operator 75
Animation Speed 243
apostrophe, displaying 32
append() 168
appending data
    file (mode) 144
application software 4
Argument, passing 30
    functions 118
Arithmetic Operators 47
ASCII 5
asin(x) 130
Assignment operator 32
atan(x) 130
attributes, object 193
axis labels 176

B

bar chart 179
binary 4
bit 4
block of code 66
bool data type 36
    example 80
Boolean 65
    expressions 68
    logic 68
    return values 122
    variables 79
breaking long statements 52
buffer 142
Button
    callback function 228
    command 229
    create 228
    lambda 264
    options 229
    text 228
Byte 5

C

callback function 228
calling functions 31
Canvas 240
case-sensitive 33
casting 46
Centering windows 254
Central Processing Unit (CPU) 1
characters 30
    comparing 74

Index

escape 44
finding in strings 161
indexing 162
newline 45
tab 45
stripping 149

Charts
bar 179
flow 11
line 176
pie 180

check button, checkbox 237
centering, window 254
Class 194
attribute 200
constructor 195
definition 195
identity 213
initializer 195
cohesion 204
collaborative development 126
Collections 181
Comments 29
compiler 8
Compound expressions 75
concatenation 37
Concatenation 40
lists 170
condition-controlled loops 90
Conditional statements 65
config 233
configure 225
Configuration Management 12
Constant 37
Controlled loops 90
Controls 220
constructor, class 195
Copying
characters 163
cos(x) 130
Count-controlled loops 92
coupling 204
CRC Cards 204
create_line() 240
create_oval() 240
create_rectangle() 240
create_text() 240

D

Data
appending to files 144
file design 142
formatting 258
reading from files 145
writing to files 142
Data types 36
Decision structures 65
def, keyword 112
default directory 141
define, variable 32
degrees(x) 130
del 184
delimiter 148
Delivery and Maintenance 15
derived 209
destroy() 253
Development
Agile 13

Index

- cycle 14
- methodologies 13
- process 13

Dialog boxes
- Error 231
- File Open 155
- Information box 231
- Message box 231

Dictionaries 181
difference() 186
Division 50
Drop-down menus 251
dump() 207

E

e variable 130
elif 72
ELOC 30
else clause 69
else with try/except 154
encapsulation 193
end = 44
endswith() 167
Entry control
- Text entry 227
- focus 227

Errors
- cost by phase 10
- dialogs 230
- IndexError 163
- IOError 153
- object 154
- Traceback 25
- SyntaxError 26
- TypeError 153
- ValueError 47

Escape characters 44
Event handler
- main loop 223

Exceptions 152
- handling 152

exponentiation 51
extend 209

F

Files 139
- extensions 140

File modes 141
File objects 140
- appending 144
- close() 142
- mode 141
- open() 140
- open dialog 155
- read() 145
- readline() 145
- read numeric data 147
- writing numeric data 144
- writing text 142
- writeline() 172

File dialog 155
finally clause 154
find() 167
float data type 36
floating point division 50
Flowchart 11
Flow of Control 66
Font

Index

defining 242
label 222
button 228
entry control 228
style 223
for loop 92
format() 40
Format, New 258
Frame 239
Functions 27
callback 228
calling 112
definition 112
keyword arguments 120
main 112
naming 123
passing argument to 118
value-returning 121
void 112

G

gcf() 177
get() 228
getters 199
global constant 117
global variable 115
grammar 8
Graph 240
Graphical User Interface 219
Grid geometry 224
GUI
controls 220
defined 219
design 222
positioning controls 224

H

hardware 1
hard drive 3
header, function 112
high-level language 7
hypot(x) 130

I

Icon, window 256
iconphoto() 256
IDE 8
IDLE
launching 21
obtaining B
Shell 21
Text Editor 23
if 66
if-else 69
if-elif-else 72
Image 253
Immutable 40
import
statements 45
wildcard 253
In operator 165
indentation 70
IndexError 163
indexes
characters 162
lists and tuples 168
negative 163

Index

out of range 163
strings 162
infinite loop 91
Info dialog box 230
Inheritance 209
Initializer method 195
Input
devices 4
keyboard 45
instance, class 194
instantiated, class 194
int() - casting 46
integer 5
interpreter 8
intersection() 186
IntVar 234
Investment Program 102
IOError exception 153
IPO document 123
isalnum() 166
isalpha() 166
isdigit() 166
isinstance() 213
islower() 166
issubset() 186
issuperset() 186
isupper() 166
iteration 90
Iterative Enhancement 13

J

Java 6

K

keyboard input 45
keywords 8
keyword arguments 120

L

Label control 222
Lambda 263
len() 163
Libraries 19
Line graph 177
List 167
append() 168
concatenate 170
index() 169
insert() 168
max() 169
min() 169
remove() 168
reverse() 169
sort() 169
split() 171
two-dimensional 174
list() 175
load() 207
local variable 114
log(x) 130
logic errors 15
Logical operators 75
Loop 89
accumulator 96
counter 97
for 92

Index

nested 100
while 90
low-level language 6
lower() 149
lstrip() 149

M

machine cycle 7
magic numbers 37
Main Function 112
Main loop 223
main memory 2
Math module 130
Mathematical operators 48
matplotlib 175
module 175
plotting 175
max() 169
Memory 3
Menu 251
Methods, defined 15
methods, objects 197
min() 169
minsize() 222
Mixed-type expressions 49
Modular Programming (files) 126
Modularization 111
Modules 126
Modulus (%) operator 51
Multiplication (*) operator 48
Mutable 167
mutator 199

N

Named constant 37
Negative indexes 163
Nested if 71
Nested loop 100
Newline (\n) character
adding 172
defined 142
removing 149
non-volatile memory 3
not operator 78
not in operator 165
Numbers
floating point 6
formatting 41
integer 5
random 131
Numeric Lists 167

O

Objects 193
access specifiers 198
as arguments 203
attributes 193
file 142
methods 193
pickling 207
public interface 193
state 196
StringVar 233
Object Behavior Diagram 205
Object Oriented 193
Object Sequence Diagram 205

Index

OOP 193
open function 141
Open file dialog 155
operands 47
Operators
- AND, OR, NOT 75
- IN and NOT IN 165
- logical 75
- mathematical 48
- precedence 48
- relational 68

Option List 237
or operator 77
Order of Operations 66
Output 2
- devices 4
- displaying 30
- file 146
- formatting 40
- window 256

override 212

P

pack() 224
Parameter 118
pass-by-value 119
Passing arguments 119
PhotoImage 253
pi variable 130
pickling 207
pie chart 180
pip installer 175
place() 224
plot() 175
Polymorphism 212
Precedence 48
pre-test loop 90
print() 30
private access 199
protected access 198
Pseudocode 10
public interface 193
public access 198
Pyplot 175
Python 19
- Installing and running 20
- Shell 21
- Exiting 27

Q

Quit button 263
Quotes, displaying 32

R

radians(x) 130
Radio buttons 234
randint() 131
random numbers 131
randrange() 131
range() 94
read(), file 145
readline(), file 145
Relational operators 68
remove characters 149
remove() 168
repetition structures 89
replace() 167

Index

Requirements Decomposition 9
resizable() 255
return statements 121
reverse() 169
round() 50
rstrip() 149
Runtime errors 27

S

Saving programs 24
scope, variable 114
Script Mode 31
Scrum 13
SDLC 9
search() 167
secondary storage 3
sentinel 99
sep = 39
serialize 207
Sets 184
- add() 184
- difference() 186
- discard() 185
- intersection() 186
- issubset() 186
- issuperset() 186
- remove() 185
- symmetric difference() 186
- union() 186

set() 234
setters 199
Short—circuit Evaluation 78
show() 175
Showerror() 231
Showinfo() 231
Showwarning() 231
sin(x) 130
Slicing 164
SLOC 30
Software 4
- Development Lifecycle 9
- Development Process 14

sort() 169
source code 8
split() 149
sprint 13
sqrt(x) 130
state, obejct 196
startswith() 167
Storyboarding 102
str type 31
String 30
- concatenation 40
- comparing 74
- modification methods 149
- slicing 164
- testing methods 166

StringVar object 233
strip() 149
Subtraction (-) operator 48
symmetric difference() 186
Syntax 8
- syntax errors 15

System software 4

T

tan(x) 130
Test and Integration 15

Index

Text files 147
textVariable 234
Theater Example 53
tick marks 175
tkinter module 220
tkinter main loop 223
Toplevel, window 262
Traceback 25
Truncation 47
try/except 152
Tuple 175
Turtle 243
Two-dimensional Lists 174
type conversion 46

U

UML Diagram 204
Unified Modeling Language 204
uniform() 132
union() 186
upper() 167

V

value-returning function 121
Variables 32
- constants 37
- global 115
- local 114
- naming 36
- scope 114

void function 112
volatile memory 2

W

Waterfall Model 14
Weight 242
While loop 90
Widget 220
Wildcard import 253
Window
- border title 222
- centering 254
- chart 176
- destroy 253
- IDLE edit 23
- interaction 262
- menu 251
- minsize() 252
- resizable() 255

writelines() 172

X

x axis 175
x cords 175

Y

y axis 175
y cords 175

Z

Zooming, plots 176