Python Textbook

Python Textbook

DR. MARK TERWILLIGER

Contents

1. Chapter 1: Introduction

Chapter 1

INTRODUCTION

Topics Covered:

- Benefits of coding
- Computer basics
- Algorithms
- Flowcharts
- Programming

"*Everybody should learn to program a computer, because it teaches you how to think.*"

  – Steve Jobs, Apple founder

Benefits of Coding

Why should I learn to code? If you are not planning to be a software developer, this is a reasonable question. If you are a business student or a psychology major it may be difficult to see how coding fits into your career plans. The goal of this book is to help you understand how learning to code is a skill that will help you in any career. Through learning to code you will no longer only be a computer user; you will become a coding Jedi with the ability to command the computer to obey your will.

Benefits of learning to code include:

1. **You can become a better problem solver**: Employers in every

industry consistently rank problem solving as the most desired skill for new hires. Coding helps you to become a better problem solver by teaching you to break down problems into a logical and structured format. In addition, it's often necessary to find creative problem solutions that are different than anything that's been done before. This analytical process will come in handy whenever you need those skills to tackle a challenging problem at work or in your daily life.

2. ***You may need to code in your job***: It doesn't matter if you are a scientist in a research lab or a financial planner working in an office; there may come a time when you need to write code. You may need to solve a small task that your current software cannot perform. Sometimes the need to code shows up in a surprising context. Creating database queries and spreadsheet macros are essentially instances of coding. With your newfound programming skills, you will be able to write a program to solve that task and impress your boss.

3. ***You can develop a basic understanding of how software works***: In virtually every career, you will be working with technology and software on a daily basis. Once you have written programs yourself, you will gain an appreciation and understanding of how software works. You will gain insights into what features can be better exploited and be able to identify deficiencies or limitations in the programs that you are using.

4. ***You can learn to be persistent***: Albert Einstein famously stated, "It's not that I'm so smart, I just stay with problems longer." Coding helps you learn to be persistent when facing difficult problems. You may get stuck and hit roadblocks on your journey, but the satisfaction of sticking with it and finding a solution is worth the effort. It takes persistence to be successful in almost any endeavor, and coding helps you learn persistence.

5. ***You can communicate about technology effectively***: Learning the basics of programming will be helpful in job situations

where a non-techie will need to talk to someone in the computing field. There are so many terms and phrases that you will pick up while learning to program. You won't have to speak the techie language perfectly, but you will know enough to pick up on important conversations among computing professionals, especially if you are working with software developers.

Although this book will focus on all of these coding benefits, in this chapter, we will discuss a few problem-solving tools. Before we do that, though, it's important that you are familiar with some computer basics.

Computer Basics

Before we begin our journey to coding Jedi, we need to make sure we have a basic understanding of how computers work. Computers are constructed from hardware and software. The **hardware** makes up the physical components of a computer. Most general-purpose computers consist of four parts: (1) the Central Processing Unit (CPU) or processor, (2) memory or Random Access Memory (RAM), (3) inputs like a keyboard or mouse, and (4) outputs like a monitor or printer.
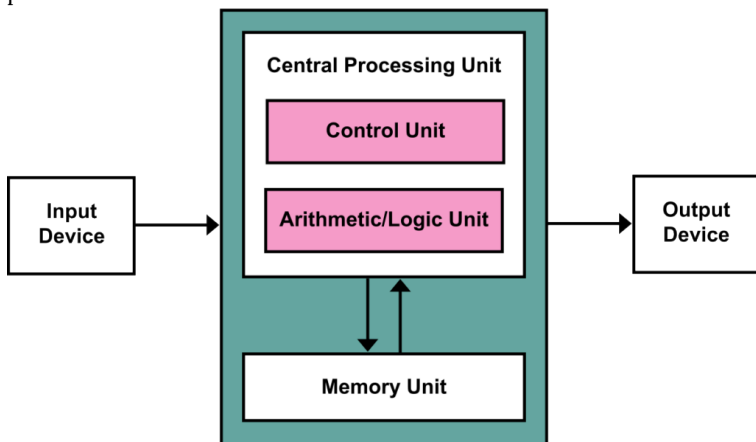
Figure 1.1: The von Neumann computer architecture.

The **software** is the computer programs. This consists of both the operating system (Windows, Macintosh, Linux, etc.) and the applications (word processor, spreadsheet, games, web browser, etc.).

As illustrated in the previous figure, most computers today use the ***von Neumann architecture***. This means that programs and data are loaded into RAM before a program runs. When you double-click on a program icon to run a program, you may notice a slight delay before your program appears. That is your computer loading the program and any necessary data from the external storage into the internal memory unit.

When we create software, most of the time our programs will follow the ***data processing cycle***. This consists of three stages: (1) input, (2) processing, and (3) output. This cycle is illustrated below.

Figure 1.2: The Data Processing Cycle.

Two of the primary tools that programmers rely on when developing solutions are algorithms and flowcharts. We will discuss each of these tools and explain how they can be used to help solve problems, as well as help simplify the coding process.

Algorithms

An **algorithm** is a set of general steps to solve a problem. We encounter algorithms frequently in our everyday lives. As we drive to school, we are following an algorithm (turn right, go 1.5 miles, turn left at the second light, etc.). When we get home and decide to treat ourselves by baking a cake (beat two eggs, add one cup of flour, add one tablespoon of salt, stir vigorously, etc.), we are following an algorithm.

Let's take a look at an example. Suppose you have a parent that is constantly yelling at you because you leave your bedroom lamp on. Maybe you are curious as to how much that is actually increasing

the monthly electricity bill. You decide to write an algorithm that will solve this problem.

With a little digging on the Internet, you discover that in order to find the cost of electricity, you will need to know three things: (1) the wattage of your light bulb, (2) how many hours did you leave it on, and (3) the price that your electric company charges. You also discover that to compute this cost, you simply multiply the wattage by the hours, then divide that by 1,000 times the price of electricity. You divide by 1000 since electric companies charge by center per kilowatt-hours, and we are asking the user to enter the time in hours. Therefore, your algorithm ends up looking like this:

Algorithm for computing cost of electricity:

- Have the user input wattage, hours, price
    - cost = (wattage x hours) / (1000 x price)
    - Output cost

Algorithms for computer programs need to be a lot more precise than algorithms for people. For example, many shampoo bottles will include the following "algorithm:"

a)   Wet your hair
b)   Apply a small amount of shampoo
c)   Lather
d)   Rinse
e)   Repeat

A computer program would need to clarify – "How do you wet your hair?" "What is a small amount of shampoo?" "How do you lather?" And most importantly, only repeat one time! Computers are way too literal and need extremely detailed instructions.

For a list of steps to be considered an algorithm, it must have the following five characteristics:

1. **Inputs** – zero or more well-defined data items that must be provided to the algorithm
2. **Outputs** – one or more well-defined results produced by the

algorithm

3. **Definiteness** – the algorithm must specify every step and the order the steps must be taken in the process
4. **Effectiveness** – every step must be feasible. You couldn't have a step, for example, that said "list every prime number"
5. **Finiteness** – the algorithm must eventually stop

Flowchart

A *flowchart* is a visual representation of a problem solution. Different shapes in a flowchart have different meanings. Arrows, or arcs, connect the shapes and provide the flow of control for your problem solution. The following table illustrates the meaning of some of the more commonly used flowchart shapes.

Shape picture	Shape Name	Shape Purpose
	Ellipse	Start/End
	Parallelogram	Inputs/Outputs
	Rectangle	Formulas/ Actions
	Diamond	Decisions

Table 1.1: Common Flowchart symbols.

Going back to the problem of computing the electricity, let's take a look at those steps expressed using a flowchart. There are many tools for creating a flowchart. We recommend using the free web http://draw.io to build your flowcharts.

```
          ┌─────────────┐
          │    Start    │
          └──────┬──────┘
                 │
                 ▼
       ┌──────────────────────────┐
      ╱ Input wattage, hours, price╱
     └──────────────┬──────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────┐
    │ cost = (wattage x hours) / (1000 x price) │
    └───────────────────┬──────────────────┘
                        │
                        ▼
              ┌──────────────────┐
             ╱   Output cost     ╱
            └─────────┬──────────┘
                      │
                      ▼
              ┌──────────────┐
              │     End      │
              └──────────────┘
```
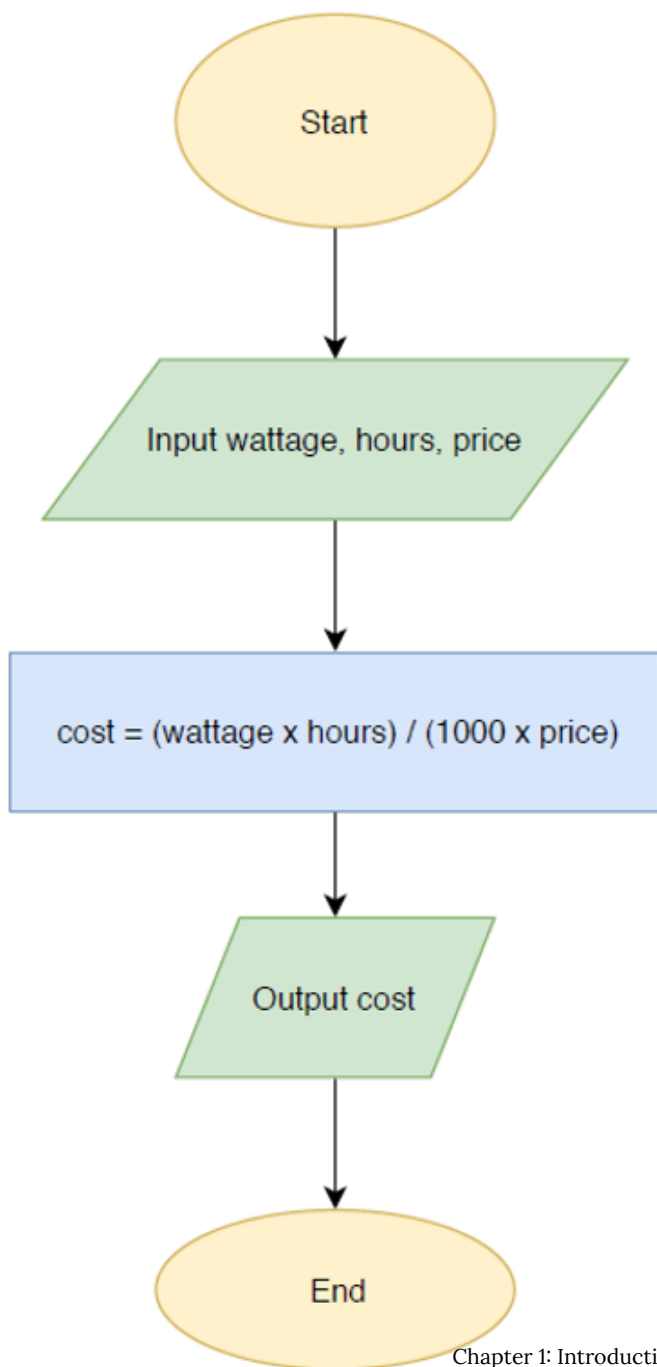
Figure 1.3: Flowchart to compute the cost of electricity.

What is Computer Programming?

A computer, in its simplest form, is nothing more than a collection of silicon, plastic, metal, glass, and wire. When you turn it on, it does precisely what it's been instructed to do – nothing more and nothing less. That's where programming comes in. A ***program*** is a set of instructions that tell the computer what it's supposed to do; ***programming*** is the process of preparing these instructions. Because computers interpret their programs very literally, programmers need to be quite explicit in the directions that they prepare.

The Origins of Programming

Since the first computers were developed in the 1940s, the discipline of computer programming has undergone a continuous evolution. In those days, computers were often programmed by means of large patch panels, using wires to represent individual instructions. Early on, though, it was recognized that flexibility would be increased if computer programs could instead be encoded as numeric data and stored in the computer's memory. While an improvement over patch panels, these machine language programs were still difficult to work with. Even then, most computers used binary numbers, and the ***machine language*** programs were nothing more than strings of zeros and ones.

To streamline the programmer's job, special ***assembly languages*** were developed. Rather than having to remember, for example, that the binary pattern 00101010 is the instruction that tells the computer to add two values, while the pattern 00101011 stands for subtract, the assembly language programmer uses special mnemonic names, such as ADD or SUB. In addition, assembly languages introduced the concept of using labels to stand for addresses within the computer's memory. Thus, the instruction:

ADD A,B

might be used to tell the computer to add the values in memory locations 123 and 147, rather than the binary form:

00101010 01111011 10010011

Of course, the computer didn't, and still doesn't, understand assembly language directly. Instead, special programs, called assemblers, were (and are) used to translate assembly language to its binary equivalent.

Although assembly language programming is still an option with computer systems, it's used only sparingly, primarily for performing very low-level tasks where direct communication with the computer's hardware is required. Most programming is instead done using more sophisticated languages. This is because assembly language is still quite difficult to work with, requiring even the simplest tasks to be broken down into sequences of several, or even several hundred, instructions. Also, virtually every computer system has its own unique assembly language. To run an existing assembly language program on a new computer system requires translation of the program into the new system's assembly language – often a formidable task.

The Development of High-Level Languages

High-level programming languages were first introduced in the 1950s. Whereas each instruction in an assembly language represents a single machine language instruction, a single high-level language instruction will usually translate into several machine language instructions. This implies, of course, that high-level languages are far more expressive than assembly languages. It also implies that the translation process required to convert programs written in these languages into a form that the computer can process is far more complex.

There are two strategies for translating high-level languages. The first, software called a *compiler* translates programs fully into machine language. Once translated, the compiled program can be run at any time without any additional translation required. In contrast, other languages are interpretive. When a program written in an interpretive language is run, an *interpreter* program reads one

instruction at a time, and determines how to carry out the required action.

To better understand the difference between compiling and interpreting, imagine that you have an article written in a foreign language. You could hire someone to translate the article to English and give you a written copy of this translation. This is what a compiler does. Alternatively, you could hire someone to read the article aloud, translating it to English as they read. This is what an interpreter does. Notice the important difference between these two approaches. When the article is "compiled" for you, you can refer back to the translated version at any time; the "interpreted" version, however, is not retained, and you'd need to seek out your interpreter again if you want to review the article's contents.

There are literally hundreds of different high-level programming languages. When first learning to program, one of the first challenges is the selection of an appropriate language. The TIOBE (The Importance Of Being Earnest) index is a measure of the popularity of programming languages. This list gets updated monthly, but here is a recent glimpse of the top ten high-level, general-purpose languages:

1. C
2. Java
3. Python
4. C++
5. C#
6. Visual Basic
7. JavaScript
8. PHP
9. Go
10. R

For this book, we have chosen the Python programming language. In Chapter Two, we will explain why Python is a great choice for learning to code. We will also show you how to download and install

Python for free in just a matter of minutes. Before you know it, you will be writing your first programs.

Chapter Review Exercises:

1.1. Describe five benefits for a person not working in a computing career to learn how to code.

1.2. Explain the function of each of the following flowchart shapes:

a. Arrow
b. Diamond
c. Ellipse
d. Parallelogram
e. Rectangle

Programming Projects:

1.1. Answer the following questions about algorithms:

a. Give the steps for withdrawing money from an ATM.
b. Explain how your steps for part (a) meet all five of the essential characteristics for an algorithm.
c. Develop an algorithm for subtracting two 3-digit numbers.
d. Explain how your steps for part (c) meet all five of the essential characteristics for an algorithm.
e. Create an algorithm for another common, everyday task you are familiar with.
f. Explain how your algorithm for part (e) meets all five of the essential characteristics for an algorithm.

2. Chapter 2: Python Basics

PYTHON BASICS

Topics Covered:

- Python overview
- Download Python
- Creating our first program
- Saving and running a program

"*I use Python almost regularly at my job. I have used it for forecasting time series data to estimate staffing requirements and various other business metrics. I have also used it to automate an ETL process by using Pandas to break contents of a large excel file into smaller excel files by categories. This was a huge time saver.*"

– Ali Murad, Financial Analyst, University of North Alabama Graduate

Python Overview

In this book we use Python as the introductory programming language. One of the primary reasons we chose Python is because it is considered an easy to learn language. The designer of the

language emphasized code ***readability***. This is an important feature of a language since it makes the code easier to understand.

In addition to being a great language for beginners, Python is also a ***very powerful*** language. It is used in a broad range of applications, including Geographic Information Systems (GIS), artificial intelligence (AI), visualization, machine learning, and robotics.

Python is an ***object-oriented*** language, which means your program data is stored as objects that have properties and functions. Most modern programming languages, such as C++, Java, and Visual Basic, are object-oriented.

Finally, as we saw in the chapter one, Python is one of the most popular programming languages today according to the TIOBE index. Knowing a language that is popular is helpful, since an employer is more apt to value someone having that skill. Also, the resources to learn Python, including books, web sites, tutorials, and videos, are plentiful.

Python uses an ***interpreter*** to translate your high-level code into a form that the CPU can understand and execute. Python is an ***open source*** language. This means that you can not only download it for free but can even view and modify the code used to create the Python interpreter. The language is also portable, which means the code that you write in one operating system (Windows, Mac, Linux) will work in another. This is a major advantage of using an interpreter rather than a compiler, since interpreted languages don't need to be translated into a specific computer's machine code.

There are two versions of Python, 2.X and 3.X, that are common in use today. These versions are not backward compatible. That is, if you are using version 3 of Python and try to run a program that works in version 2, it may not work. In this book, we will be using the version 3. This is an unusual circumstance; historically, most programming languages tried to maintain backward compatibility so that existing programs would not need to be modified when updates were made to the language features.

Download Python

As mentioned earlier, Python is free. To download Python to your

own computer, you must first visit the official website at http://python.org. The top of the Python main web page looks like this (at least until the next language update arrives!):
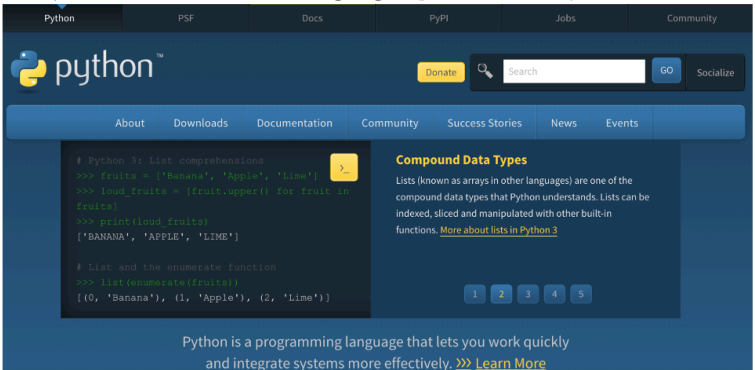
Figure 2.1: The Python website located at python.org.

If you scroll over the "Downloads" tab, a smaller window will appear looking like that seen below. The web site will recognize your computer's operating system so you won't have to make any difficult decisions. Just click on the button with the current version number (i.e., "Python 3.9.2"). Once you download the file, simply run it and follow any instructions that pop up along the way. It should take a couple of minutes and you'll be ready to start coding.

Figure 2.2: Downloading Python from the python.org website.

You should notice a "Documentation" tab on this initial web page, as well. This is where the official documentation for the Python language resides. This page will be a good resource throughout your programming adventures so make a note of it.

Once you have Python successfully downloaded, there are several ways that you can write and run your Python programs. Idle is an **integrated development environment (IDE)** for Python. An IDE allows you to create, edit, run, troubleshoot, save, and open your code all in the same easy-to-use program.
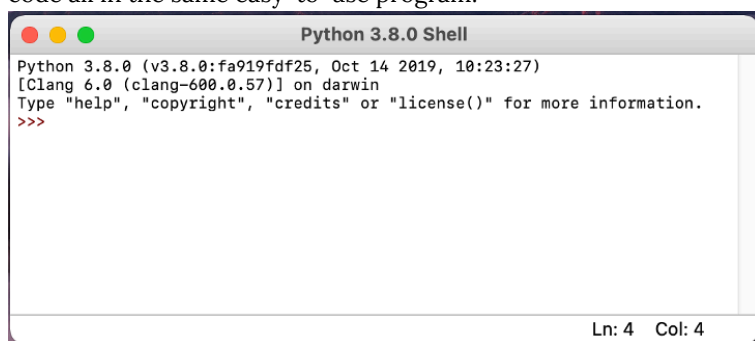
Figure 2.3: The Python Idle software.

When you open Idle for the first time, a window similar to the one above will appear. The Python version number (In this case, 3.8.0, since the author was too lazy to update his computer to version 3.9.2) is shown in the window title and the first line inside the window. The Idle environment has two modes. Initially, the environment is in *interactive mode*, which allows you to evaluate expressions and individual instructions. When you want to develop a larger program, however, you will want to switch to *script mode*.

The string ">>>" is the screen *prompt* which tells you that the environment is waiting for you to type a Python expression or instruction. The interactive mode can be used to play around and just experiment with different commands. Try typing the four expressions shown below following the prompt. You will notice that

Python uses different colors for **syntax highlighting**. The **syntax** are the rules of the language. More about that later.

```
>>> 3+2
5
>>> 4 * 5
20
>>> 2**5
32
>>> print ("Hello")
Hello
>>> "cat" * 5
'catcatcatcatcat'
```

Figure 2.4: Experimenting in Interactive mode.

When you type 3+2 and press the Enter key, the result of 5 is returned. Of course, this comes as no surprise. In the interactive mode, the expression you enter is executed immediately. You will notice that all of the program output is displayed by Idle as blue. Obviously, 4*5 resulted in 20. Why did 2**5 return 32? If it isn't obvious, see if you can research (i.e., Google) the answer.

Whenever you want to display something in Python, you use a special function called **print**. All of the built-in Python functions appear as purple in Idle. Also note how the word "hello" appeared in double quotes. This means it is a **string**, or sequence of characters. Idle displays strings using green. The last example had a string, followed by *, followed by a whole number. How did Python evaluate that expression?

Creating our First Program

Most of the time, we want to write larger programs, which are sometimes called *scripts*. You do not want to write a full-fledged script using the interactive mode. Instead, go into script mode by clicking on "File" and then "New File." A convenient way to program in this mode is to resize this new script window and put it in the right side of your screen. Resize the interactive window and place it on the left side of your screen:
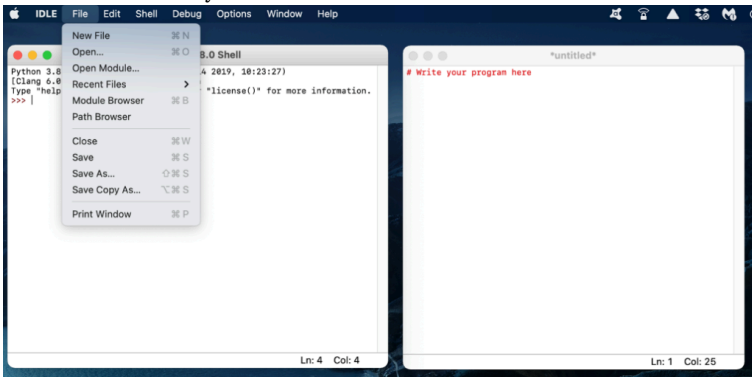
Figure 2.5: Creating a new Python program.

Now, let's get back to tackling the electricity problem from the chapter one. In your new script window, type the following program, replace the X's with your name and the current date. We will break each instruction down, line by line, afterwards.

Note: It is important that you actually type in this program to get used to this environment of creating, editing, troubleshooting, saving, and running programs.

```
# Program to compute cost of electricity
# Written by XXX XXXXX
# CS101 – Introduction to Computer Programming
# Date: XX/XX/XXXX

print("Computing the cost of electricity")
print()

wattage = float(input("Enter wattage: "))
hours = float(input("Enter number of hours used: "))
price = float(input("Enter price per kWh in cents: "))

cost = (wattage*hours) / (1000*price)

print ("Cost of electricity:", cost)
```

Figure 2.6: The electricity program.

The lines at the top of this programming that begin with the pound sign (#) are called **comments**. When the interpreter goes to translate and run your code, it will ignore comments. These lines are added to document your program. That is, the comments will describe your program and various aspects of it. As a minimum, you should include the 4-line comment block illustrated in this example.

As mentioned earlier, the **print** function is used to display information to the screen. Each print statement will send an output to a new line. With the print function, you include what you want displayed inside parentheses. Anything inside the parentheses of a function is called a **parameter**. With no parameter, the print function simply outputs a blank line.

The **input** function is used to get information from the user via the keyboard. The program will pause until the user types information followed by the Enter key. Data will be read in as a string and stored as a **variable**, which is simply a named memory location. The parameter for the input function is a **prompt** that is displayed as a hint to the user indicating what should be entered at the keyboard.

You should notice in the program the same input statements for wattage, hours, and price that we saw in the algorithm and flowchart in chapter one. Since the input function returns the user's input as a string, we need to use the **float** function to convert that string to a floating-point number. Data types will be discussed in more detail in the upcoming chapters.
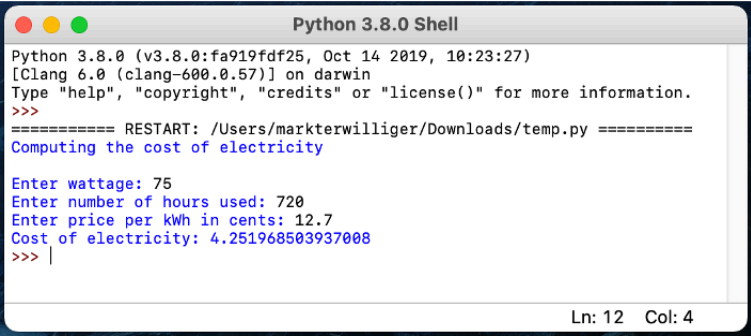
After the input instructions, we use our formula to compute the cost of the electricity based on the values entered by the user. The final step is to output the cost using a print statement. The print statement uses two parameters – a string literal that functions as a label, and the variable cost. Notice that the label has quotes around it and the variable does not. When printing a variable, the contents of the variable will be output, not the actual name of the variable.

Saving and Running a Program

Finally, we are ready to run the program. In Idle, click "Run" on the menu and then "Run Module" to execute your program. Alternatively, the "F5" function key is treated as a shortcut to trigger this same action. When you run the program, the output and interaction will occur in your interactive screen.

Note: Idle will force you to save your program before you run it. All of your Python programs will end with a ".py" file extension. You should save all of your programs to a common location that you can access conveniently. Also, you should save using a meaningful file name like "electricity.py".

Now, let's find out how much that lamp is costing the family. Suppose your lamp uses a 75-watt bulb. A month is approximately 720 hours (24 hours per day x 30 days) so we will input 720 for the second input. Finally, the price of electricity varies depending on many things, including the state you live in. Suppose your electric company charges 12.7 cents per kilowatt hour. The interaction would like that shown:

```
Python 3.8.0 Shell

Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: /Users/markterwilliger/Downloads/temp.py ==========
Computing the cost of electricity

Enter wattage: 75
Enter number of hours used: 720
Enter price per kWh in cents: 12.7
Cost of electricity: 4.251968503937008
>>>
                                                        Ln: 12   Col: 4
```

Figure 2.7: A program interaction for the electricity program.

The program shows us that it will cost $4.25 to leave that lamp on for a month straight. Of course, we have a lot more decimal places than we need, but we will worry about formatting our output later. We have a working program that solves our problem!

It is possible that your program did not run successfully. If you mistype one or more instructions, you may have created a ***syntax error***, which is when one of your instructions breaks the rules of the language. For example, suppose you forget the closing right parenthesis at the end of the first print instruction. As shown below, Idle will display a pop-up screen with an error message. The position of that error will be highlighted in red in your program code.
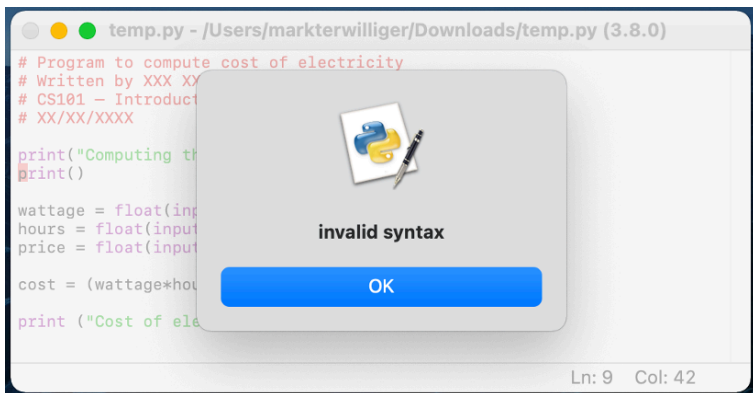
Figure 2.8: A syntax error.

Now that you've successfully created, saved, and run your first Python program, it's time to start digging deeper. In the chapter three, we will discuss the most common numeric data types used in Python programs, as well as the operations and functions associated with each.

INTERACTIVE – Do I really need to download Python?

It is highly recommend that you follow the instructions provided in this chapter to download Python to your computer. It is possible to create and run Python programs using your web browser. A website called http://trinket.io provides a full Python interpreter. We provide this plug-in along with some instructions in our Try Python page. You can give it a spin right now in the screen below. After you successfully run the "hello world" program, add another print statement to display your full name, and then run the program again.

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=21*

Chapter Review Exercises:

2.1. Define the following terms:

a) Comment   b) Float   c) IDE   d) Input function

e) Object-oriented   f) Open source   g) Parameter   h) Portable

i) Print function   j) Prompt   k) String   l) Syntax error

Programming Projects:

2.1. Many athletes are concerned with reaching their ideal **training heart rate** during their workouts.

- The **maximum heart rate (MHR)** can be found by subtracting your age from 220. It is recommended that the **ideal training heart rate** for exercise is between 55 and 85 percent of this MHR.

- Write a program to ask a user to enter her age. Your program will compute and display her maximum heart rate, as well as the lower and upper range of the ideal training heart rate.
- Write an algorithm to solve the problem. Save it as a Word document.
- Draw a flowchart to model the solution to the problem. Save it as a PDF document.
- Implement the solution to your problem as a Python program. Add 4 lines of comments at the top.

3. Chapter 3: Numeric Data

NUMERIC DATA
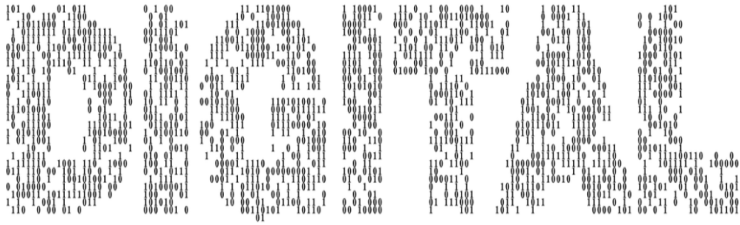
Topics Covered:

- Data types
- Numeric operators
- Precedence rules
- Assignment
- Error types

"I think that great programming is not all that dissimilar to great art.

Once you start thinking in concepts of programming it makes you a better person...as does learning a foreign language, as does learning math, as does learning how to read."
  – Jack Dorsey, Twitter creator and founder and Square CEO

Data Types

Ultimately, everything that is stored within a computer's memory is a number. The computer's processing unit has to decide, based on context, what the numbers in the memory represent. If the CPU is asked to execute the instruction at memory location 385, the CPU assumes that the number stored in that memory location should be treated as a machine language instruction. If that instruction at location 385 indicates that the contents of memory location 1376 should be added to the contents of memory location 2795, then the CPU will treat the values in those memory locations as numbers.

A major distinction between different programming languages is the set of resources provided by that language for interpreting the values stored in the computer's memory. The different mechanisms provided by the language are generally referred to as the language's data types.

The simplest data type supported by virtually all programming languages is the numeric data type. Just like we recognize two kinds of numbers, whole numbers and fractions, computers normally distinguish between int, or whole number values, and float, or fractional values. Technically, a float value is actually like a decimal representation of a fraction. And just like some fractional values, like 1/3, don't have an exact decimal representation, many float values are actually approximations for the fractional value that is being represented.

In practice, we will use an **int** when storing a whole number such as a person's age (i.e., 19) or golf score (i.e., 97). **Float** is short for floating point number and can stores things like a grade point average (i.e., 3.48) or a bank account balance (i.e., 578.25).

As a programmer, you have to decide on the data type as you

anticipate how your data will processed and viewed. For example, maybe you are writing software that will ask the user to enter today's temperature in Fahrenheit degrees. If you expect all input values to be whole numbers (i.e., 72), then you would choose an **int**. If a user is allowed to enter numbers with decimals (i.e., 68.45), then a **float** would be the proper data type.

Numeric Operators

As you begin to solve problems, you will use **numeric operators** to create formulas. The next table shows some of Python's most common numeric operators.

Operation	Operator	Python Example	Result
Addition	+	3+5	8
Subtraction	–	14-9	5
Multiplication	*	4*7	28
Floating point division	/	19/4	4.75
Integer (floor) division	//	19//4	4
Remainder (modulus)	%	19%4	3
Exponent	**	2**5	32

Table 3.1: Python operators.

Precedence Rules

Suppose a student decided to use Python in the interactive mode to find her average of three exam scores: 70, 80, and 90. When she typed in the expression, the result is shown below:

```
>>> 70+80+90/3
180.0
```

Of course, that doesn't seem right. There is a good chance you already figured out the problem. The precedence rules (order of

operations) you learned in your math courses apply in Python expressions.

The *precedence rules* follow this order:

1. Parenthesis
2. Exponent
3. Multiplication and division (left to right)
4. Addition and subtraction (left to right)

The student's expression had two additions and a division. Since division has a higher precedence than addition, 90 was divided by 3 first. That quotient was then added to 70 and 80, resulting in the 180. To force the addition to occur first, we would need to add parenthesis since they have the highest precedent level:

```
>>> (70+80+90)/3
80.0
```

When we write programs, we usually want to store or save results of expression so that we can later use them or display them. To store data, we use a *variable*, which is simply a named memory location. Python has rules for creating variable names.

Rules for creating variable names:

- Use letters, digits, and underscores
- Variable names are case-sensitive
- Do not use reserved words

A good practice for creating a variable is to use meaningful names like **hoursWorked**, **firstName**, and **exam3**. The table below provides a list of potential variable names.

Variable name	Valid name?	Comments
City	yes	This is a good name
city	yes	Good name, too, but it is a different variable than **City**
print	no	**print** is a Python function
Print	yes	It is valid, but may be confused with print
Hours-worked	no	The dash (-) is not a valid character
Hours_worked	yes	This is a good name
w	yes	It is valid, but 1-letter names should be avoided
mph	yes	Valid, but **milesPerHour** would be much better
input	no	**input** is a Python function
last name	no	You can't have spaces in a variable name

Table 3.2: Python variable names.

Variable Assignment

To give a value to a variable you use an *assignment* statement. The assignment operator is the equal sign (=). The instruction **age = 19** will assign the value 19 to the variable **age**. Technically, **age** is a reference that identifies the memory location where your program stored the number 19. You might picture your computer's memory as a bunch of cells:
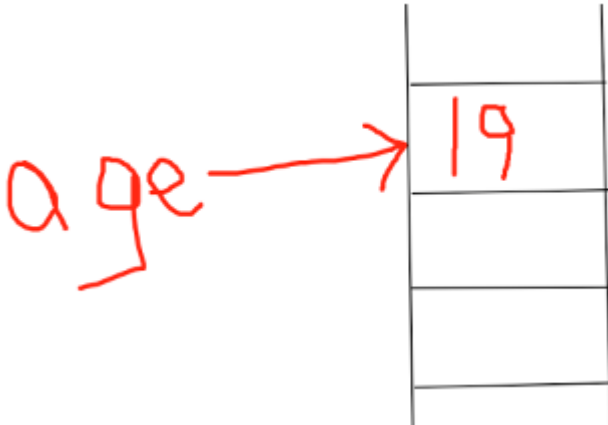
Perhaps later you decide to change the value of **age**, setting it to 23 with the instruction **age = 23**. The **age** variable now points to the number 23 in memory. (Side note: The number 19 is now orphaned and will be removed by garbage collection.) Your computer's memory will look like this:
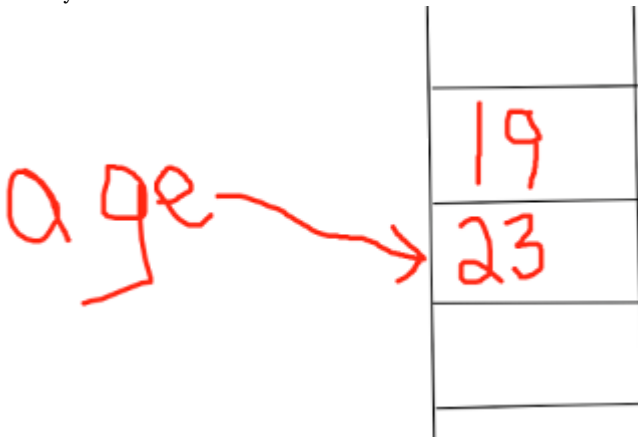
Once you assign a value to a variable, you can retrieve it to use in another formula or you may want to output it using the print function. It's recommended that you use the interactive mode to experiment with assigning and printing variables. Here are a couple of examples:

```
>>> number=27
>>> print(number)
27
>>> age=(30-3)/2
>>> print(age)
13.5
>>> price=0.95*29.99
>>> print(price)
28.490499999999997
>>> print(temp)
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print(temp)
NameError: name 'temp' is not defined
```

Figure 3.1: Examples of the Python assignment operator.

Error Types

You noticed in that last example that a message popped up. Errors show up in red when you work in Idle, so we try to avoid that color. In this example, we tried to print the contents of a variable named **temp**, but we never created or defined that variable. Python will not permit you to access a variable if you haven't defined it.

Every language has its own *syntax*, which are the rules of language. This includes the language structure, whitespace, reserved words, etc. If one of your instructions breaks one of these language rules, it is called a *syntax error*. Unfortunately, many beginning programmers (as well as some experienced programmers) mistakenly believe that, once they have removed all syntax errors, their programmers are good to go. The table below shows three types of common programming errors. A *runtime error*, such as dividing by zero or taking the square root of a negative number, occur during program execution. With a *logical error*, your program runs but the intended results or outputs are not correct.

Type of Error	Description	Examples
Syntax	An instruction breaks the rules of the language	**print(5 number1+number2=total**
Runtime	The given data causes a problem while your program is running	**print(10/0) math.sqrt(-25)**
Logic	Your program still runs, but the results are not correct	**Avg=exam1+exam2+exam3 / 3 triangleArea=base*height*2**

Table 3.3: Programming Error Types.

Common Python Functions

As we continue to solve problems using Python, more functions will be needed to perform necessary commands. The table below lists a couple more common Python functions along with examples of each in action:

Function	Python Examples	Result
Absolute value	abs(-23) abs(5.8)	23 5.8
int conversion	int(3.8) int("47")	3 47
float conversion	float(5) float("-29.3")	5.0 -29.3
round	round(23.7) round(-5.2394) round(4.7777,1) round(34.88888,2)	24 -5 4.8 34.89

Table 3.4: Common Python Functions.

INTERACTIVE – Debugging

Often times, we describe an error in a computer program as a ***bug***. The process of removing those errors is therefore called ***debugging***. To give you a little practice debugging, consider the program below. It is trying to compute and display the hypotenuse of a right triangle

given the two triangle legs inputted by the user. There are several errors embedded in this program, though. See if you can identify and correct each error. If you are familiar with a 3-4-5 triangle, that would be an easy case for you to test. To begin, try simply running the program and see what clues that gives you.

> One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=24

Some Example Applications

Let's take a look at a couple of examples to see if we can tie it all together. First, we will develop a program to compute the winning percentage for a baseball team. The problem-solving in this example is pretty straight-forward. As you can see from the comments, the program is broken into input, processing, and output sections. The team wins and losses must be converted to integers since the input function returns a string. You might also notice that we included the round function to display the winning percentage to the nearest tenth.

```
# Program to compute baseball winning percentage
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX
print("Computing the baseball winning percentage")
print()

# Here is the input section
teamName = input("Enter team name: ")
wins = int(input("How many wins? "))
losses = int(input("How many losses? "))

# Compute some formulas
totalGames = wins + losses
winningPct = 100 * wins / totalGames

# Output section
print (teamName, "won", round(winningPct,1), "% of their games.")
```

Figure 3.2: Baseball Winning Percentage Program.

Here is a sample run of this program. Notice that the program output is displayed as blue, and the user input is shown in black.

```
Computing the baseball winning percentage

Enter team name: Lions
How many wins? 23
How many losses? 19
Lions won 54.8 % of their games.
>>>
```

You may be curious as to when we would ever need the "%" or "//" operators. It turns out that they come in handy solving many problems. Suppose you worked as a cashier at a store, and you wanted to write a Python program to compute and display change for a customer. We will first show you some code that will solve this problem. Then, we will take a look at a sample run to help you understand how these two numeric operators are used.

```
# program to compute and display change for a customer
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX

print("This program will determine the change to give a customer")
print()

change = int(input ("Enter change (0 to 99)? "))

quarters = change // 25
print("Quarters:", quarters)

change = change%25
dimes = change // 10
print ("Dimes:", dimes)

change = change%10
nickels = change // 5
print ("Nickels:", nickels)

pennies = change%5
print ("Pennies:", pennies)
```

Figure 3.3: Money Changing Program.

In the sample run below, the user input 68 as the number of cents the customer should be given. If you look at 68//25, the integer division returns 2 as the number of quarters. The expression 68%25

results in 18 as the remainder. Once the program computes the quarters, it next computes dimes in a similar fashion. The expression 18//10 results in 1 dime and the remainder 18%10 is 8. The number of nickels is 8//5, or 1. Finally, the number of pennies is 8%5, or 3.

```
This program will determine the change to give a customer

Enter change (0 to 99)? 68
Quarters: 2
Dimes: 1
Nickels: 1
Pennies: 3
>>>
```

Chapter Review Exercises:

3.1. Show the output of the Python code.

a = 18 − 3 * 4 + 7

print (a)

3.2. Show the output of the Python code.

b = 9/5

print (b)

3.3. Show the output of the Python code.

c = 9//5

print (c)

3.4. Show the output of the Python code.

d = 17%3

print (d)

3.5. Show the output of the Python code.

e = 43**

print (e)

Programming Projects:

3.1. If P dollars (called the *principal*) is invested at *r*% interested compounded annually, then the *future value* of the investment after *n* years is given by the formula:

$$future = P\left(1 + \frac{r}{100}\right)^{n}$$

Write a program to ask the user for the principal, annual interest rate given as a percentage, and the number of years of an investment. Compute the future value and display it using 2 decimals. Also, compute the total interest earned on the investment and display that value using 2 decimals.

4. Chapter 4: Strings

STRINGS

Topics Covered:

- Strings
- Common String Methods
- Casting

Strings

To computer programmers, a ***string*** is simply a sequence of characters. A specific example of one is called a ***string literal***, and is surrounded by quotes. Examples include "Donald Duck", "35630-1418", "Florence, Alabama", and "". That last example, called an ***empty string***, has no characters. You may have also noticed the second example had numeric digits in it. A string may contain any of

the printable characters you see on your keyboard, as well as some characters that don't show up there.

We use string variables all of the time to store data that is not numeric. The computer stores this data in memory using an encoding called ASCII (American Standard Code for Information Interchange). You might visualize a string as a table, with each slot storing a single character. Take a look at the following 11-character string literal "Roar Lions!":

0	1	2	3	4	5	6	7	8	9	10
R	o	a	r		L	i	o	n	s	!

Figure 4.1: A Python string.

Above the table, the position of each character is shown. This position, often called *subscript* or *index*, begins counting at zero. While it may seem unnatural to refer to the first position in the table as position 0, that's a computer-related quirk you'll see repeatedly, for reasons that are well beyond the scope of this book. The subscript often comes in handy when you need to process or extract information from a string. The following example using Python's interactive mode illustrates this concept.

```
>>> phrase = "Roar Lions!"
>>> phrase[3]
'r'
>>> phrase[4]
' '
>>> phrase[5]
'L'
>>>
```

Figure 4.2: Accessing characters within a string.

Taking a look at this example, you should notice that Idle output the resulting strings with single quotes. You can use single or double quotes for strings. When displaying the 5th character – the character in position **4** – a space is printed. You can use **[m:n]** to create a substring, or *slice*, of a string. It returns the string that starts at position **m** and ends at position **n-1**.

The following example shows four examples of the slice. The first displays a string beginning at position **3** and goes up to but does not include position **5**. The second example displays characters **7** through **10**. The third example does not include an ending number after the colon. This will display all characters at position **3** and beyond. The final example is missing the beginning subscript before the colon. All characters before position **5** are displayed.

```
>>> message = "I love quizzes"
>>> message[3:5]
'ov'
>>> message[7:11]
'quiz'
>>> message[3:]
'ove quizzes'
>>> message[:5]
'I lov'
>>>
```

Figure 4.3: String slices.

You can use the Python **len()** function to determine the length of a string. The following shows you a couple examples using **len()**:

```
>>> phrase = "Roar Lions!"
>>> len(phrase)
11
>>> thing = ""
>>> len(thing)
0
>>>
```

Figure 4.4: The Python len function.

Common String Methods

In chapter one, we mentioned that Python is an object-oriented language. A data item like a string is treated as an object. This means that in addition to storing data like "Roar Lions!", the string object also has built-in functions, or *methods*, that it can reference using the name of the string variable followed by a period followed by the function name. The examples below will demonstrate several of the methods provided for string objects:

```
>>> message = "I love quizzes"
>>> message.find("quiz")
7
>>> message.find("Love")
-1
>>> message.upper()
'I LOVE QUIZZES'
>>> message.lower()
'i love quizzes'
>>> message.title()
'I Love Quizzes'
>>> message.count("z")
2
>>> message.count("dog")
0
>>>
```

Figure 4.5: Python string methods.

The find message searches the string for a pattern and returns the position where it matches. The pattern "quiz" was found at position 7. The pattern "Love" was not found so a -1 was returned. Notice that the matching is case-sensitive.

When applied to numbers, the plus (+) operator is used to add the two operands. You can also use this same plus operator with two strings. It will perform *string concatenation*, which simply means the strings are joined together. When a language defines an operator to perform different functions depending on its operands, this is called *operator overloading*. It's intuitive and easy to use, as illustrated below:

```
>>> str1 = "dog"
>>> str2 = "house"
>>> str1 + str2
'doghouse'
>>> first = "Guido"
>>> last = "van Rossum"
>>> fullName = last + ", " + first
>>> fullName
'van Rossum, Guido'
>>> city = "Florence"
>>> state = "Alabama"
>>> zipcode = "35632"
>>> city + ", " + state + " " + zipcode
'Florence, Alabama 35632'
>>>
```

Figure 4.6: Combining strings with the plus (+) operator.

In the examples above, you can see how you can apply the plus operator multiple times and it will concatenate the strings from left to right. The example demonstrates how you could use plus operator to combine strings and then store the result to a variable.

As previously mentioned, the *input* function allows the user to enter information from the keyboard. The result is a string that is usually assigned to a variable. Most of the time, you will want to provide a prompt as a parameter to the function so that the user knows that your program is waiting for some input. Here is an example that shows a few examples:

```
name = input("What is your name? ")
sign = input("What is your sign? ")
major = input("What is your academic major? ")

print(name, "is a", sign, "and likes to study", major)
```

Figure 4.7: Getting string input from the user.

A sample run of that code is shown next:

```
What is your name? Jill
What is your sign? Scorpio
What is your academic major? Accounting
Jill is a Scorpio and likes to study Accounting
>>>
```

Figure 4.8: Program interaction.

Casting

In the previous example, all of the input values were used as strings. When we need the user to enter a number that will be used in an arithmetic expression, we need to use *type casting*, which is converting from one type to another. Here are a couple of examples from the interactive window that demonstrate type casting:

```
>>> int("37")
37
>>> float("24.95")
24.95
>>> int(35.67)
35
>>> str(-14)
'-14'
>>> str(3.084)
'3.084'
>>>
```

Figure 4.9: Examples of casting.

INTERACTIVE – Fun with strings

Try out the program below to learn more about strings and the

string slicing function. Experiment with different inputs and try to predict the output before you run the program.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=26

Don't forget to tip your wait person!

Now let's look at a complete example. Suppose we wanted to write a program that asks the user for a restaurant server's name, amount of restaurant bill, and tip percentage. The program should compute and display the tip amount and the total bill.

```
# Program to compute and display the tip for a meal
# CS101 - Introduction to Computer Programming
# Written by XXXX XXXXXX
# Date: XX/XX/XXXX
print ("Welcome to the tip calculator")
print()
mealCost = float(input ("Cost of meal? "))
tipPercentage = float(input ("Percentage of tip? "))
waitPerson = input ("Name of wait person? ")
tipAmount = mealCost*tipPercentage/100
print (waitPerson, "gets a tip of", tipAmount)
total = mealCost + tipAmount
print ("The total is", total)
```

Figure 4.10: The tipping program.

When we run this program, a sample interaction appears below:

```
Welcome to the tip calculator

Cost of meal? 24.87
Percentage of tip? 14.5
Name of wait person? Steve
Steve gets a tip of 3.60615
The total is 28.47615
```

Figure 4.11: Program interaction.

As you can see from the code, a type cast was needed to convert the meal cost and tip percentage from a **float** to a **string**. A type cast was not needed for the wait person since it is already the intended **string** type.

Chapter Review Exercises:
4.1. Show the output of the Python instructions:
str = "Tigers win the game"
print (len(str))
4.2. Show the output of the Python instructions:
str = "Tigers win the game"
print (str[6:9])
4.3. Show the output of the Python instructions:
str = "Tigers win the game"
print (str.find("win"))
4.4. Show the output of the Python instructions:
str = "Tigers win the game"
print (str.find("lose"))
4.5. Show the output of the Python instructions:
str = "Tigers win the game"
print (str.upper())
4.6. Show the output of the Python instructions:
str = "Tigers win the game"
print (str.count("e"))

5. Chapter 5: Printing

PRINTING

Topics Covered:

- The print function
- Rounding
- Format specifiers

The print function

Even when we are working with a text-based user interface (TUI), we would like to design programs that are easy to use and intuitive for the user. In order to do this effectively, it is necessary to understand how to take full control of how you display data on the screen. To begin, we will take a closer look at how Python's **print** statement works.

```
>>> print("dog")
dog
>>> print()

>>> print("cat","dog","fish")
cat dog fish
>>> print(1,2,3)
1 2 3
>>>
```

You can see from these examples that the print statement allows any number of *parameters*, or items between the parentheses. In the first example, there was only one parameter, **"dog"**. The second example had no parameters so it just printed a blank line. The final two examples each had three parameters. Notice how a space was displayed between each item; this sets Python apart from many other commonly-used programming languages.

If you wanted something other than a space between each item, you can define the separator with the *sep* attribute of the print statement. Here are some examples of programmer-defined separators:

```
>>> print("cat","dog","fish",sep="###")
cat###dog###fish
>>> print("cat","dog","fish",sep=" and ")
cat and dog and fish
>>> print("cat","dog","fish",sep="")
catdogfish
>>> print("cat","dog","fish",sep="\t")
cat     dog     fish
>>> print("cat","dog","fish",sep="\n")
cat
dog
fish
>>>
```

You can see in each example, the items that were normally separated by a space are now separated by the string that followed **sep=**. The third example shows how you can use the empty string to print items without any separation. The final two examples begin with a backslash (\) and create what is called an *escape sequence*. The character that follows the backslash defines a special character. The **"\t"** produced a tab and the **"\n"** created a *newline*.

We have previously seen that each *print* statement will generate output on a new line. Occasionally, you would like to display something, but you do not want to move the following output to the next line. In this case, you can define how the line should end be setting the end attribute to a string that should terminate the print. Here is an example:

```
first="John"
middle="Jacob"
last = "Jingleheimer"
final = "Schmidt"

print(first,end=" ")
print(middle,end=" ")
print(last,end=" ")
print(final)
```

In this code example, the first three *print* statements are directing each output to be terminated with a space instead of the default

new line. Without these three **end="** " clauses, the four print statements would create four lines of output. Instead, the output looks like this:

```
John Jacob Jingleheimer Schmidt
>>>
```

Rounding Numbers

We introduced the **round** function in chapter 3. It can be used to round a floating point to the nearest integer. It can also be used to round a floating point number to a specified number of decimal places. It is important to note that when you use round in a print statement with a variable, the value of that variable does not actually change. This example will illustrate the point:

```
number = 29/8
print("number is",number)
print("rounded number is",round(number))
print("rounded to 1 decimal place, number is",round(number,1))
print()
print("now, number is",number)
```

The output when that code segment is executed looks like this:

```
number is 3.625
rounded number is 4
rounded to 1 decimal place, number is 3.6

now, number is 3.625
>>>
```

There are cases in which you would like to store the rounded result of an expression. Perhaps you are rounding a currency expression to the nearest hundredth so you can keep track of dollars and cents. In the following example, we compute the simple interest for an amount of $465.83 deposited in an account earning 4.25% interest for 2.5 years. We will display that interest as calculated, plus rounded using 2 decimal places. Here is the Python code along with the output displayed by the program:

```
principal = 465.83
rate = 0.0425
time = 2.5

interest = principal * rate * time
print("interest is", interest)
print("rounded, interest is", round(interest,2))
print("now, the interest is", interest)
```

```
interest is 49.494437500000004
rounded, interest is 49.49
now, the interest is 49.494437500000004
>>>
```

You can either round a previously calculated result, or you can include the round function in the computation, as shown below:

    roundedInterest = round(interest,2)   # store to a new variable
    interest = round(principal*rate*time,2) # round formula

Format Specifiers

Sometimes, rounding alone doesn't provide enough control over the output's appearance. Let's take a look at the simple interest example again, but this time use data that produces an interest value that naturally ends with only one decimal place. Here is the Python code and the sample run:

```
principal = 465
rate = 0.04
time = 2.5

interest = round(principal * rate * time,2)
print("interest is", interest)
```

```
interest is 46.5
>>>
```

With a currency amount, we often want two decimal places displayed, along with a dollar sign immediately in front of the amount. Since a space separates the label and number by default, we can use the separator to close the gap. We can use a *format*

specifier to force two decimal places to be printed. Here is the new and improved version, along with the program's output:

```
principal = 465
rate = 0.04
time = 2.5

interest = round(principal * rate * time,2)
print("interest is $", "%.2f"%interest, sep="")
```

```
interest is $46.50
>>>
```

In the example above, the format specifier "%.2f" was placed before the variable **interest** with a percent symbol, %, separating the two. This expression forces the output to display two digits after the decimal point. Python provides quite a bit of functionality with format specifiers, as illustrated below:

```
a=22
b=4444
c=666666
print("%8d"%a)
print("%8d"%b)
print("%8d"%c)
print()
d=333
e=55555
f=7777777
print("%-8d"%d,  "coding")
print("%-8d"%e,  "is")
print("%-8d"%f,  "fun")
```

The "%xxd" specifier is used to define the total width of the output field. By default, the result is aligned to the right, or right justified, with spaces attached to the left of the value. The dash (-) can be used to change the alignment to the left so spaces are instead added to the right when integers d, e, and f are displayed:

```
        22
      4444
    666666

333         coding
55555       is
7777777     fun
>>>
```

Python also allows you to combine rounding and field-width specification into a single formatting specification. This is especially useful when you are trying to align numbers into columns. In the Python code below, we will display each of four numbers using two decimal places and a total of seven characters:

```
amount1 = 4.777777
amount2 = 893
amount3 = 6.3
amount4 = 32.111
print("%7.2f"%amount1)
print("%7.2f"%amount2)
print("%7.2f"%amount3)
print("%7.2f"%amount4)
```

In this output, notice that the value **893.00** takes up six characters so one space was added on the left. Also, observe that the decimal points all align vertically, as well as the digits in the tenths and hundredths place.

```
   4.78
 893.00
   6.30
  32.11
>>>
```

We have seen how the letter d is used for integers and f is used for floating point numbers. You can use the letter s when formatting string variables or expressions. The alignment for strings is similar to that of integers. In the next example, we will display the first name, last name, and career rushing yards for three former football players. In each case, both the first and last names will be displayed in 10-character fields, aligned on the left. The rushing yards will be displayed in eight-character fields; special formatting is used to specify that commas should be used to separate the numeric values in three-digit groups.

```
first1 = "Barry"
last1  = "Sanders"
yards1 = 15269
first2 = "LaDanian"
last2  = "Tomlinson"
yards2 = 13684
first3 = "Jim"
last3  = "Brown"
yards3 = 12312
print("%-10s"%first1, "%-10s"%last1,f"{yards1:8,d}")
print("%-10s"%first2, "%-10s"%last2,f"{yards2:8,d}")
print("%-10s"%first3, "%-10s"%last3,f"{yards3:8,d}")
```

```
Barry       Sanders       15,269
LaDanian    Tomlinson     13,684
Jim         Brown         12,312
>>>
```

INTERACTIVE – Formatting output

Look at the code snippet below and try to predict the output. Run the program and check your guess. Take a look at the code where there is a backslash and consecutive double quotes after the height is printed. Can you figure out what's going on?

> 🖳 *One or more interactive elements has been excluded*

As noted ahead of the example, a different technique was used to control the appearance of the rushing yards. Python provides a number of different mechanisms for controlling the format of your output. In the following example, three different techniques are used to display a floating-point value in a field that is six characters wide, and allows two digits after the decimal point. We believe that the first method, %-formatting, is the easiest to understand, and will be more than adequate for our needs. To learn more about the other two mechanisms, the **format()** method and the *f-string*, you can consult other resources, including books and websites.

```
number = 82.77777
print("number is", "%6.2f"%number)
print("number is", "{0:6,.2f}".format(number))
print("number is", f"{number:6.2f}")
```

The corresponding output:

```
number is  82.78
number is  82.78
number is  82.78
>>>
```

With a Python f-string, you can put expressions between curly brackets **{}**. In the example below, we print an f-string that includes the variables **this** and **that**, as well as their product **this*that**:

```
>>> this = 3
>>> that = 5
>>> print(f"{this} times {that} is {this*that}")
3 times 5 is 15
>>>
```

You can use f-strings to specify the spacing and precision of variables or expressions that you want to display. Below we print the value of the floating point variable named **number**, first with **1** decimal place, then with **5** decimal places:

```
>>> number = 42.777
>>> print(f"number = {number:.1f}")
number = 42.8
>>> print(f"number = {number:.5f}")
number = 42.77700
>>>
```

Chapter Review Exercises:

5.1. Show the **exact** output of the Python code (use ^ to indicate spaces):

amount1 = 25.8888
amount2 = 4
amount3 = 382.62
amount4 = 843.676767
print("%8.3f"%amount1)
print("%8.3f"%amount2)
print("%8.3f"%amount3)
print("%8.3f"%amount4)

6. Chapter 6: Selection

SELECTION

Topics Covered:

- Decision making in Python
- Comparison operators
- Boolean expressions
- Else and elif statements
- Nested if-else blocks

Decision Making in Python

In order to allow a computer to handle different data values in different ways, programming languages need to provide some type of decision-making capabilities. In Python, the simplest form of

decision-making capability is provided by the **if** statement, which has the following format:

> *if Condition:*
>   *Action*

In this prototype, **Condition** stands for any expression that can be evaluated to either **True** or **False**. Most often, the **Condition** is a simple comparison. In Python, you can use the following comparison operators:

Operator	Meaning	Example
==	is equal to	A == B
<	is less than	A < B
>	is greater than	A > B
!=	is not equal to	A != B
<=	is less than or equal to	A <= B
>=	is greater than or equal to	A >= B

These operators should be self-explanatory. You need to be careful, though, when comparing fractional values, since equality rarely holds between two fractional variables or expressions. This is because fractional values are approximated in the computer. When a series of arithmetic operations is performed using approximated values, it is almost inevitable that these values will undergo a significant amount of round-off in the final decimal places. Here are a couple of examples of conditions, or Boolean expressions, evaluated in the Python Interactive window:

```
>>> 5>12
False
>>> 5<=5
True
>>> 7==12
False
>>> 9!=5
True
>>>          .
```

The **Action** component of the **if** statement is the instruction, or series of instructions, to be carried out if the condition is **True**. If more than one instruction is to be carried out, the whole series of instructions should be indented at the same level. This tells the computer to treat the entire group of instructions, commonly referred to as a ***block***, as a single unit. Python's Idle environment assists you by automatically indenting when you type a colon at the end of a line.

To illustrate, the following are examples of legal **if** statements in Python:

if hoursWorked <= 40:
  grossPay = rate * hours
if prevID != currID:
  idCount = idCount + 1
  prevID = currID

In the first example, the variable **hoursWorked** is compared to 40; if it is less than or equal to 40, the value of **grossPay** is computed by multiplying **rate** by **hours**.

In the second example, the variables **currID** and **prevID** are compared. It's assumed that these are both numeric variables. If the values don't match exactly, then two actions are carried out. First, 1

is added to the variable **idCount**; presumably, the program is going to count how many distinct identification numbers are processed. Second, the value of the variable **currID** is being copied into the variable **prevID**.

It is worth emphasizing here just how important the **indentation** of blocks is with Python. Most languages use either keywords, such as begin/end, or curly brackets { }, to indicate blocks of code. The **white space** (space, tab, new line) in these languages is ignored. For a block to work in Python, the indenting for each instruction has to be created using the same keystrokes. For example, you cannot use the TAB key on the first line of the block and then consecutive spaces on the second line, even if they appear to visually align vertically.

The standard **comparison operators (<, <=, >, >=, ==,** and **!=)** can be used to compare strings, as well as for comparing numeric values. You should be aware, though, that string comparison is case sensitive. The ASCII values of characters are used when comparisons are made between two strings. Here are some examples:

```
>>> "fish" < "turkey"
True
>>> "dog" < "cat"
False
>>> "twinkie" < "twister"
True
>>> "cat" < "Cat"
False
>>>
```

In the first example, "fish" is less than "turkey" because the ASCII

value of "f" (102) is less than that of "t" (116). A "d" has a value (100) greater than that of "c" (99), so "dog" is not less than "cat". When strings begin with the same character, the comparison moves on to succeeding characters until characters in the two strings differ. The string "twinkie" is less than "twister" since the fourth character "n" had an ASCII value (110) less than that of "s" (115). Finally, upper case letters have smaller ASCII values than lower case letters. The string "cat" is not less than "Cat" because the "c" has a value of 99 and the "C" has a value of 67. Since 99 is not less than 67, the expression **"cat"<"Cat"** returns **False**.

The bool Data Type

In some cases, it's handy to be able to remember if something has, or hasn't, occurred. One way to do this is to set a special "marker" value in some variable. For example, the variable **flag**, of type **int**, might be initially set to 0, but then changed to 1 if some special event takes place. The test:

if flag == 1:

could then be used to see if the event has taken place. Of course, it's important for the programmer to remember if the value 1 stands for "yes" or "no" within the program, which may not be easy. A better way is to use a more descriptive variable name, and give that variable the type **bool**, which stands for **True** or **False**. Typically, the **bool** variable might be set to **False** to indicate that an awaited event hasn't yet taken place, and then subsequently changed to **True** after the event does occur. For example, the variable **buttonPressed** might be used to indicate that a mouse button has been pressed. When the button is pressed, **buttonPressed** could be set to **True**, while a release of the button would cause the value to be changed to **False**.

To test the variable, it is sufficient to say:

if buttonPressed:

instead of:

if buttonPressed == true:

While this may seem like a minor difference, it does improve the overall readability of the program.

The if – else Statement

Often, a programmer has to choose between two alternate actions, depending on the result of a comparison. For example, the task of figuring an hourly employee's gross pay, when overtime is paid for hours in excess of 40, might be programmed as:

if hoursWorked <= 40:

  *grossPay = hoursWorked * hourlyRate*

if hoursWorked > 40:

  *grossPay = hourlyRate*40 + 1.5*hourlyRate*(hoursWorked−40)*

Can you guess what would happen if the programmer accidentally wrote the first comparison as **hoursWorked < 40**? If you think about it, you'll realize that this could be a serious error - **grossPay** won't be computed for workers with exactly 40 hours. Sadly, this type of problem arises far more frequently than you might imagine. In some circumstances, Python is able to tell you that you have a potential problem. If the variable **grossPay** in the above example hadn't been given an initial value before the **if** statements are encountered, the Python interpreter will trigger an error since the variable wasn't initialized. But if the value of grossPay had already been computed for a previous employee, then this employee would end up receiving the same gross pay!

In addition to this being a nasty error condition waiting to happen, the redundant comparison of **hoursWorked** to 40 is also at least a little troubling; it just doesn't seem necessary. To simplify this type of situation, most programming languages allow another form of the **if** statement, usually called the **if - else**. Using Python's version of this instruction, the above sequence could be written as:

if hoursWorked <= 40:

  *grossPay = hoursWorked * hourlyRate*

else:

  *grossPay = hourlyRate*40 + 1.5*hourlyRate*(hoursWorked−40)*

Whenever the keyword **else** is encountered, it is automatically

associated with the most recent **if**. Perhaps the simplest way to understand its use is to think of it like this:

> *If some condition is true,*
>> *here's what I want you to do.*
>
> *But if that condition is false,*
>> *I want you to do this, instead.*

As with the simple **if** statement, multiple actions are specified by indenting those instructions to the same level. For example, if a company's payroll required regular and overtime pay to be calculated separately, it might be done like this:

> *if hoursWorked <= 40:*
>> *regularPay = hoursWorked * hourlyRate*
>> *overtimePay = 0*
>
> *else:*
>> *regularPay = 40 * hourlyRate*
>> *overtimePay = 1.5 * hourlyRate * (hoursWorked − 40)*

As you model the decision-making process using a flowchart, whenever a condition occurs, you should include that condition inside of a diamond. Two arrows will exit the diamond, one designating the path if the condition is **True**; the other path when the condition is **False**. Below are flowcharts modeling an **if** statement and an **if-else** statement.
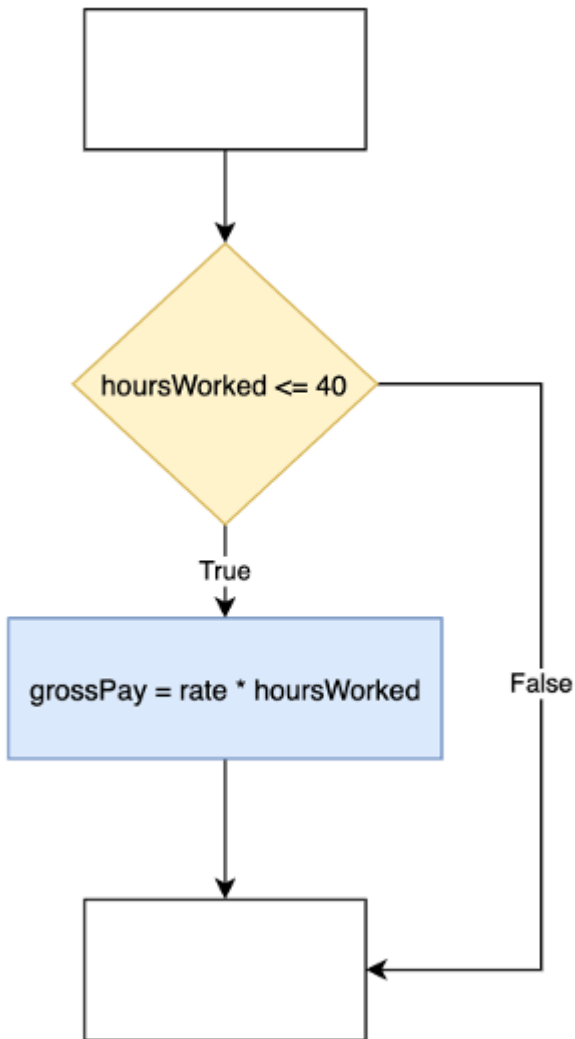
Flowchart Modeling an if statement.
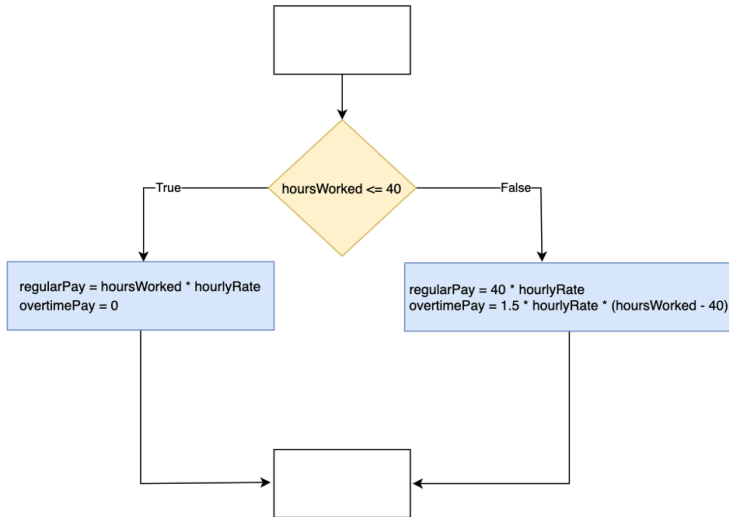
Flowchart Modeling an if-else statement.

General Programming Constructs

No matter which programming language you are using to solve a problem, there are **three general computer programming constructs** that are required:

1. **Sequence** – the computer will execute one instruction after the next.
2. **Selection** – this is the decision-making process using if or if-else statements.
3. **Repetition** – creating a loop. We will study this in chapter 7.

Suppose the local bagel shop asked you to write a simple program to compute the charge for each customer's purchase. Bagels normally cost 75 cents each, but if you buy a half-dozen or more, then the charge is 60 cents per bagel. Of course, once we learn the quantity, a decision needs to be made. Our newly acquired if-else statement would be perfect here:

```
# Program to compute charge for bagel sale
# Written by XXX XXXXX
# CS101 – Introduction to Computer Programming
# Date: XX/XX/XXXX
quantity = int(input("How many bagels do you want? "))
if quantity<6:
charge = .75 * quantity
else:
charge = .60 * quantity
print("Total charge is", charge)
```

Compound Conditional Expressions

In many cases, it's necessary to consider multiple conditions to decide if some action should or should not be performed. In some cases, two or more conditions must be true; in other cases, at least one condition, from a list of possible conditions, must be satisfied. In these situations, you can combine individual comparisons in a single expression. For example, a salesperson might be due to receive a bonus if he or she has at least 20 customers, and accounts for at least $10,000 in sales. Or maybe a student will receive an F in a class if his or her test average is below 60, or if he or she has more than 10 unexcused absences.

The key words in these two examples are "and" and "or." When two or more conditions are joined with the word "and," it implies that ALL of the conditions must be satisfied in order for the entire expression to be considered true. In contrast, the word "or" implies that AT LEAST ONE of the specified conditions must be satisfied to obtain a result of true.

The salesperson example from above might be represented in Python like this:

if numCustomers >= 20 and totalSales >= 10000:

   ...

while the student example might instead be expressed like this:

if testAvg < 60 or absences >= 10:

   ...

While less common, it's sometimes necessary to mix "and" and "or" conditions in a single expression. Python has default rules for interpreting these expressions, but it's probably best to instead use parentheses to explicitly dictate the order in which the individual expressions should be combined.

Although not required, you can add parentheses, which many people find to be more readable:

if (testAvg < 60) or (absences >= 10) :

or even:

if ((testAvg < 60) or (absences >= 10)) :

Finally, in some contexts, it's desirable to perform some action if some condition or conditions are NOT met. Typically, the word **not** is placed outside of a parenthesized expression. For example, you could check to see if a salesperson is not eligible for a bonus like this:

if not (numCustomers >= 20 and totalSales >= 10000):

   ...

An easier way to ask this same question, though, might be to ask:

if numCustomers < 20 or totalSales < 10000:

   ...

Nested if Statements

Whenever one or both of the actions associated with an **if – else** instruction contains another **if** statement, the resulting form is referred to as a nested **if**. The nested form arises naturally in contexts where a choice must be made between several different options. To illustrate, consider the following scenario.

To receive a grade of A, a student's average must be at least 90. A B will be assigned if the average is at least 80, but less than 90, while a C is assigned for a grade of at least 70, but less than 80. For an average of 60 or higher, but less than 70, the letter grade is D, while an average below 60 leads to a grade of F.

Using only **if** statements, this grade assignment problem could be programmed like this:

if average >= 90:

grade = 'A'
if average >= 80 and average < 90:
  grade = 'B'
if average >= 70 and average < 80:
  grade = 'C'
if average >= 60 and average < 70:
  grade = 'D'
if average < 60:
  grade = 'F'

While this sequence will certainly do the trick, it requires several redundant comparisons, and also contains a number of potential trouble spots, where either numbers or comparison operators could easily be messed up.

An alternate form, which significantly reduces the number of comparisons required, is the following:

grade = 'A'
if average < 90:
  grade = 'B'
if average < 80:
  grade = 'C'
if average < 70:
  grade = 'D'
if average < 60:
  grade = 'F'

While this form is clearly shorter, it's far more difficult to understand, because it relies on a trick. Every student is initially assigned an A, and then their grades are reassigned as each new test shows their average to be lower. Often, programmers forget that other people need to be able to understand their programs, too, and will rely on tricks that, while the trick they came up with might be obvious to them, it might not be obvious to others.

The elif statement

When you have an if statement immediately following an else section, Python provides **elif**, which combines the two statements.

A cleaner version of the grade checker, which takes advantage of the **elif** statement to assign a grade to each student exactly once, would be:

```
if average >= 90:          # Test for an A
    grade = 'A'
elif average >= 80:        # Can't be an A, so test for a B
    grade = 'B'
elif average >= 70:        # Can't be a B, so test for a C
    grade = 'C'
elif average >= 60:        # Can't be a C, so test for a D
    grade = 'D'
else:                      # It wasn't a D, either
    grade = 'F'
```

The comments on the right clarify why this form works. When an **if** statement follows an **else**, it can rely on the fact that the condition associated with the preceding **if** is already known to be **False**. This characteristic is shown in most well written nested **if** blocks – the secondary **if** statements are placed so they follow an **else**, thereby ensuring that the earlier **if** condition is already known to be **False**.

Whatever you do, don't place the **if** within the **if** portion, as illustrated below:

```
if average < 90:                  # D O N ' T
    if average < 80:              # E V E R
        if average < 70:          # W R I T E
            if average < 60:      # N E S T E D
                grade = 'F'       # I F
            else:                 # S T A T E M E N T S
                grade = 'D'       # T H A T
        else:                     # L O O K
            grade = 'C'           # L I K E
    else:                         # T H I S
        grade = 'B'               # ! ! ! !
else:
    grade = 'A'
```

While this form will, in fact, work properly, it's a rare person who could figure it out on the first try!

INTERACTIVE – Can you debug it?

The program below asks the user to enter a number. It will then display whether the input number is positive, negative, or zero. The only problem, though, is that the code contains three errors. Can you fix it?

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=30

Let's take a look at a complete program that will need selection to work. The program will process a savings-account withdrawal. It should ask the user for their current balance and the amount of the withdrawal. If the transaction was successful, it should display the customer's new balance. If the balance wasn't large enough to handle the withdrawal, the program should output "Transaction denied". If the new balance is less than $100, then "Balance below $100" should be printed.

Before we tackle the code, we can model the solution to this problem using the following flowchart:
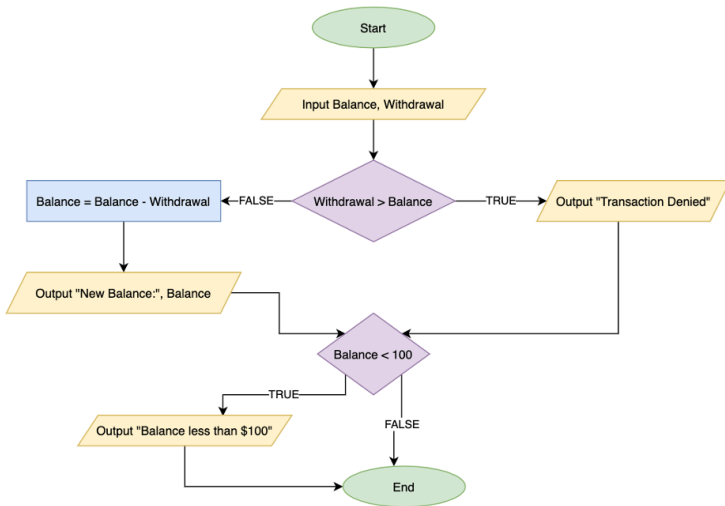
Flowchart Modeling the Bank Withdrawal Problem.

As you observe from the flowchart, there are two triangles, or decisions, involved. The first is an if-else statement that checks to see if the customer has sufficient funds for the withdrawal. The second decision is an if-statement checking to see if the customer's balance if below $100. The Python solution looks like this:

```
# Program to compute process a savings account withdrawal
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX

balance = float(input("What is the current bank balance? "))
withdrawal = float(input("How much would you like to withdrawal? "))
if withdrawal>balance:
    print("Transaction denied")
else:
    balance = balance - withdrawal
    print("Your new balance is $", balance)

if balance < 100:
    print ("Balance below $100")
```

Chapter Review Exercises:

6.1. Given the following Python instructions:

a = 12

b = 12

c = 8

What is the value of the following condition (True or False)?

a != b

6.2. Given the following Python instructions:

a = 12

b = 12

c = 8

What is the value of the following condition (True or False)?

a < b and b > c

6.3. Given the following Python instructions:

a = 12

b = 12

c = 8

What is the value of the following condition (True or False)?

a < b and b > c

6.4. Show the output if the following Python instructions were executed:

```
a = 4
b = 5
c = 13
if a+b < 10:
    print(c)
else:
    print(a)
```

6.5. Show the output if the following Python instructions were executed:

```
miles = 10
if miles<20:
    price = 5 + .30*miles
else:
    price = 2 + (20-miles)*.10
print(price)
```

Programming Projects:

6.1. Write a program that will ask for the name and age of two people. The program should then display a message saying either "X is older than Y", "X is younger Y", or "X is the same age as Y" (assuming the first person has name X and the second person has name Y).

6.2. Leo's Print Shoppe charges 8 cents per copy for the first 50 copies and 5 cents per copy for the copies beyond the first 50. Write a Python program that asks for the customer's name and how many copies they need. Your program should output the customer's name and the total cost (using a dollar sign and 2 decimal places).

- Write an algorithm to solve the problem. Save it as a Word document.
- Write a flowchart to model the solution to the problem. Save it as a PDF document.
- Implement the solution to your problem as a Python program. Add at least 4 lines of comments at the top.

7. Chapter 7: Repetition

REPETITION

Topics Covered:

- Repetition
- While loop
- For loop
- Range()
- Finding maximum

Repetition

Suppose a student was trying to compute the average of three exam scores. It would be relatively easy to construct a Python program to do this:

exam1 = float(input("Enter exam #1 score: "))
exam2 = float(input("Enter exam #2 score: "))

```
exam3 = float(input("Enter exam #3 score: "))
average = (exam1 + exam2 + exam3) / 3.0
print("The average is", "%.1f"%average)
```

Okay, but what if there were five exam scores instead of three? Well, that's obvious. Just create two more variables, add two inputs, and then divide the sum by 5.0 instead of 3.0:

```
exam1 = float(input("Enter exam #1 score: "))
exam2 = float(input("Enter exam #2 score: "))
exam3 = float(input("Enter exam #3 score: "))
exam4 = float(input("Enter exam #4 score: "))
exam5 = float(input("Enter exam #5 score: "))
average = (exam1 + exam2 + exam3 + exam4 + exam5) / 5.0
print("The average is", "%.1f"%average)
```

Well, what if there were 100 exam scores? Of course, you could add another 95 variables and 95 input statements, but there must be an easier way. If we create a **total** variable and add each exam score to it once it's been input, we don't need to remember the exam scores. What we would like to do is repeat the same two instructions (input number, add it to the **total**) 100 times.

Perhaps the one feature of computers that has contributed the most to their success is their ability to repeat instructions until some type of event has taken place. There are several different instructions that can be used to implement repetitions within a Python program. These loop instructions, along with the selection instructions discussed in chapter 6, are examples of *control structures*, since they alter the sequential flow of the program.

The while Instruction

The first repetition instruction we'll examine is the **while** instruction, which has the following form in Python:

```
while Condition:
    Action
```

As with the **if** statement, the component labeled **Condition** is any **True** or **False** expression, and is usually, but not always, some

type of comparison. In operation, the instruction tells the computer to repeatedly:

a) Test the **Condition** to see if it is **True**

b) If the **Condition** is **True**, carry out the **Action**

The key feature that distinguishes this from an **if** statement is that fact that this sequence can be carried out over and over, as necessary, until the value of the **Condition** becomes **False**.

Like the **if**, the **Action** component of a **while** instruction may consist of more than just a single instruction. In this case, it is necessary to indent at the same level all of the instructions that make up the **Action**.

To illustrate the behavior of the **while** instruction, consider these examples:

sum = 0
number = 10
while number > 0:
  sum = sum + number
  number = number − 1

This instruction sequence will add the values 10, 9, 8, and so on, down to 1, to the variable *sum*. In other words, the instruction sequence is a long way to say:

$$sum = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1$$

In this example, the indenting of the block is needed to capture the two instructions that make up the **Action**. To see why this matters, take a look at the alternative:

sum = 0
number = 10
while number > 0:
  sum = sum + number
number = number − 1

In this case, even though both instructions were intended to be within the loop, the only instruction actually in the loop is **sum = sum + number**. Since the value of the variable **number** never changes from its initial setting of 10, this loop will never terminate. This type of (erroneous) program segment is called an *infinite loop*.

As a third example of the **while** instruction, consider this sequence:

```
sum = 0
number = 0
while number > 10:
    sum = sum + number
    number = number + 1
```

In this case, the apparent intent of the programmer was to count from 0 up to 10. However, because the **Condition** is incorrectly stated, the value of the **Condition** is initially **False**. When this occurs, Python will bypass the instructions that make up the loop entirely. The **while** instruction is commonly referred to as a *pre-test loop* structure; it tests the **Condition** at the very start of the loop, before it carries out the specified **Action**.

We have seen how useful the Python Interactive mode is for testing lines of code and experimenting. With multiple-line instructions like if statements or while loops, you can still use the Interactive mode. After typing the colon and ENTER after your control structure's Condition, the cursor moves to the next line without displaying the >>> prompt. You can type as many instructions as you'd like into your block. Just hit ENTER twice after the final instruction in your block. Here is a simple while loop example tested in the Interactive window:

```
>>> num=2
>>> while num<=20:
        print(num, num*num, sep="\t")
        num = num + 3


2       4
5       25
8       64
11      121
14      196
17      289
20      400
>>>
```

Before the loop, the variable **num** is initialized to **2**. Since **2** is less than **20**, the *Condition* is *True* and *Action* will be executed. In this case, the *Action* is two instructions, display **num** and its square, then increment num by **3**. This process continues until num eventually reaches a value of **23**, which is greater than **20**, and causes the *Condition* to be *False*.

Let's look at a full example of the while loop in action. The problem we want to solve: How long it will take to have half of a loan paid off? Before we jump into the code, let's take a look at a flowchart that models a solution to this problem. Flowcharting a while loop looks very similar to that of an if statement. The *Condition* is represented by a diamond and arrows labeled **True** and **False** must exit this diamond.

In our solution, we will first ask the user to input the amount of the **loan**, the annual interest **rate**, and the monthly **payment** amount. We will initialize a **balance** variable to the loan amount and set a **month** variable to 1. Our *Condition* will check to see if the remaining **balance** is still greater than half of the original **loan**. For the *Action* of this while loop, the program will compute the **interest**, determine the new **balance**, and then output

the **month**, **interest**, **payment** and **balance**. To compute the interest, the balance is multiplied by the annual interest rate. It is divided by 12 to convert it to a monthly rate and then divided by 100 to convert it from a percentage to a decimal number. The final step is to add **1** to the **month** before looping back to the *Condition*.
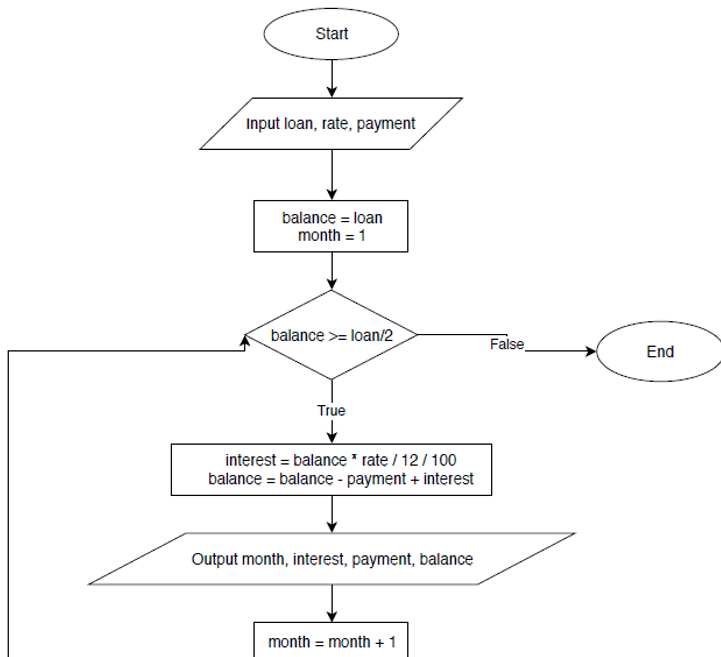
Flowchart Modeling a Loan Payoff.

Next, we will use the flowchart solution to show the Python program that solves this problem:

```
# Program to compute how long to have a loan half paid off
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX

loan = float(input("What is the loan amount? "))
rate = float(input("Annual interest rate (i.e., 7.5)? "))
payment = float(input("Annual payment? "))

balance = loan
month = 1

print ("Month", "Int.", "Payment", "Balance", sep="\t")

while balance >= loan/2:
    interest = balance * rate / 12 / 100
    balance = balance - payment + interest
    print(month, "%.2f"%interest, "%.2f"%payment,
              "%.2f"%balance, sep="\t")
    month = month + 1
```

Below is the output of a sample run of the program. Suppose you bought a "new" car and took out a loan for $14,000. The annual interest rate was 6.25% and you decided you would make monthly payments of $500. You can see from the program run that you'll be halfway to paying off that loan in just 16 months!

```
What is the loan amount? 14000
Annual interest rate (i.e., 7.5)? 6.25
Annual payment? 500
Month     Int.      Payment  Balance
1         72.92     500.00   13572.92
2         70.69     500.00   13143.61
3         68.46     500.00   12712.07
4         66.21     500.00   12278.27
5         63.95     500.00   11842.22
6         61.68     500.00   11403.90
7         59.40     500.00   10963.30
8         57.10     500.00   10520.40
9         54.79     500.00   10075.19
10        52.47     500.00   9627.67
11        50.14     500.00   9177.81
12        47.80     500.00   8725.61
13        45.45     500.00   8271.06
14        43.08     500.00   7814.14
15        40.70     500.00   7354.83
16        38.31     500.00   6893.14
>>>
```

The for Loop

Another common and very powerful repetition structure in Python is the **for** loop. The **for** loop will iterate, or repeat, once for each item in a sequence. Each time through the loop, you can access the current item in the sequence using a variable. For example, if the sequence was 1, 2, 3, 4, 5, then the **for** loop variable will take on 1 the first iteration of the loop, 2 in the second iteration of the loop, and so on. The syntax of the Python **for** loop looks like this:

for Variable in Sequence:

  Action

The **Sequence** in this statement can take on many forms. We will see later that the **Sequence** may contain items in a list or even lines

from a file. The most common use, though, is the **range** function. The reference *range(m,n)* – generates a *Sequence* m, m+1, m+2, ..., n-1. Let's take a look at a couple of examples from the Interactive mode:

```
>>> for number in range(3,7):
        print(number)


3
4
5
6
>>> for item in range(-3,3):
        print(item)


-3
-2
-1
0
1
2
>>>
```

In the first example, the range began at **3** and went up to **7**, but not including **7**. On each iteration of the loop, the current *Sequence* value was stored in the variable **number**, which was printed as the loop *Action*. In the second example, the variable **item** iterated from **-3** to **2** and was also displayed during the loop's Action.

Now would be a good time to see how we could use the **for** loop to solve a problem: Write a program that will display the world population through 2025. Assume that the population was 7 billion in 2011 and that it grows at a rate of 1.1% each year. The program should display the year and population for each year on a line together. Here is a look at one solution to this problem:

population = 7000000000
print("Year   Population")
for year in range(2011,2026):
    print(year, f"{round(population):15,d}")
    population = 1.011*population

Shown below is the output when the program was executed. Notice that this program did not involve any keyboard **input** from the user.

```
Year       Population
2011    7,000,000,000
2012    7,077,000,000
2013    7,154,847,000
2014    7,233,550,317
2015    7,313,119,370
2016    7,393,563,684
2017    7,474,892,884
2018    7,557,116,706
2019    7,640,244,990
2020    7,724,287,684
2021    7,809,254,849
2022    7,895,156,652
2023    7,982,003,375
2024    8,069,805,413
2025    8,158,573,272
>>>
```

There are a couple of things from the code above that probably need some explanation. First, since we wanted to iterate the year variable through **2025**, the ending value in the range needed to be the next integer, **2026**. To display the **population**, we first rounded it to the nearest integer. We then used an f-string to output using 15 total characters and commas between each 3-digit period.

Another way you could have written the formula for the population would be to say **population = population + .011*population**. In other words, the following year's population will be the current population plus 1.1% times the current population.

If you were thinking that you could have solved this problem using a while loop instead of a **for** loop, you are absolutely correct. Shown next is the comparable Python program using a **while** loop. It produces the exact same output as the program with the **for** loop. You should notice, though, that the **while** loop version is actually two lines longer than the **for** loop version since it was necessary to initialize the variable **year** to **2011** before the loop and increment year during the *Action* of the loop.

```
population = 7000000000
year = 2011
print("Year     Population")
while year < 2026:
   print(year, f"{round(population):15,d}")
   population = 1.011*population
   year = year + 1
```

In the previous for loop examples, the value of the control variable increased by one for each iteration. An alternate form of the **range** function can be used that allows a third parameter, usually referred to as the *step*. This will make much more sense after looking at a couple of examples using the Interactive mode:

```
>>> for number in range(13,20,2):
        print(number)


13
15
17
19
>>> for thing in range(0,30,5):
        print(thing)


0
5
10
15
20
25
>>> for item in range(-5,5,3):
        print(item)


-5
-2
1
4
>>>
```

In the first example, **number** began at **13** and stepped by increments of **2**. When it reached **21**, the loop terminated. In the second example, **thing** began at **0** and incremented by steps of **5** until

reaching the ending range value **30**. In the final example, item began at **-5**, incremented by steps of **3** and stopped when it reached **7**.

INTERACTIVE – Loopy fun!
Before running the program below, try to predict its output.

> 🖳 *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=32*

Finding Maximum

A frequent problem to solve in computing involves finding a maximum (or minimum) value of a list of numbers. The algorithm to accomplish this is pretty straightforward:

1. Set *maximum* to an artificially low value (smaller than any possible **input**)
2. Loop once for each *value*

   1. Input a *value*

   1. If *value* is larger than *maximum*, then it becomes the new *maximum*

If you know in advance how many values you will have, a for loop is a good choice. Suppose we would like to ask the user how many values will be entered first. Then, the user will enter each value. You should note that these values could be exam scores, stock prices, fish weights, temperatures, etc. Our algorithm will work for any application. Here is a Python program that will implement a solution based on the algorithm above. The output for a sample run of the program is also provided.

```
maximum = -1
count = int(input("How many values do you have? "))
for num in range(1,count+1):
    print("Enter value #", num, ": ", sep="", end="")
    value = float(input())
    if value>maximum:
        maximum=value
print("The maximum is", maximum)
```

How many values do you have? 4
Enter value #1: 83
Enter value #2: 72
Enter value #3: 91
Enter value #4: 88
The maximum is 91.0
>>>

A couple of observations from this program:

- In the **range**, **count+1** was used in order to loop **count** times.
- To make a more sophisticated prompt for an **input** statement, you can precede a no-parameter **input** with a **print** statement that provides the prompt.
- Because the program casts each **input** to a **float**, the user can enter integers or floating point numbers.

Chapter Review Exercises:

7.1. How many lines will get output when the following Python code is executed?

thing = 2
while thing < 10:
  print(thing)
  thing = thing + 1

7.2. How many lines will get output when the following Python code is executed?

thing = 2

```
while thing < 10:
    print(thing)
```

7.3. How many lines will get output when the following Python code is executed?

```
thing = 2
while thing > 10:
    print(thing)
    thing = thing + 1
```

7.4. How many lines will get output when the following Python code is executed?

```
for num in range(3,8):
    print(num)
```

7.5. How many lines will get output when the following Python code is executed?

```
for num in range(5,17,3):
    print(num)
```

Programming Projects:

7.1. Write a program that creates a table to display Fahrenheit-to-Celsius conversions. Use the following formula to do the conversions:

$$Celsius = \frac{5}{9}(Fahrenheit - 32)$$

You should ask the user for start, end, and step values for Fahrenheit temperatures. The program should print column headings and the values printed should be displayed using 2 decimal places.

The sample interaction should look like this:

```
Enter start value: -40
Enter ending value: 120
Enter step: 10

   Fahren    Celsius
  -40.00    -40.00
  -30.00    -34.44
  -20.00    -28.89
  -10.00    -23.33
    0.00    -17.78
   10.00    -12.22
   20.00     -6.67
   30.00     -1.11
   40.00      4.44
   50.00     10.00
   60.00     15.56
   70.00     21.11
   80.00     26.67
   90.00     32.22
  100.00     37.78
  110.00     43.33
>>>
```

8. Chapter 8: User-defined Functions

USER-DEFINED FUNCTIONS

Topics Covered:

- Functions
- Return values
- Parameters
- Arguments

Functions

Throughout the first seven chapters, we learned about many of Python's built-in *functions*, including **print**, **input**, and **round**. Often, especially as our computer programs get longer and more

complex, it is convenient to write our own functions. It is worth considering the reasons to create functions, which are sometimes called **subroutines** or **subprograms**.

Advantages of including user-defined functions in a project:

- **Code efficiency**: Instead of duplicating a block of code, you can write a function that implements that block and calls it multiple times.
- **Simplify problem solving**: You can take a complex problem and break it into smaller tasks, and then write a function for each of the tasks.
- **Code readability**: It is easier to read a structured program consisting of multiple functions than a long listing of code containing no subroutines.
- **Reuse of code**: Functions that perform a specific task can often by recycled or reused in multiple programs.
- **Ease of debugging**: When troubleshooting and testing code, it is easier to focus on one subroutine at a time than to try to examine one big, long program.
- **Teamwork**: Software developers frequently work in groups on large programming projects. Breaking the projects into subroutines naturally simplifies the distribution of the workload among team members.

To write your own Python function, you use the word **def** (short for define), followed by the **FunctionName** and a **ParameterList** enclosed in parenthesis, ending with a colon. Similar to an **if** statement or loop, the **Action** block of instructions included in the function must be indented at the same level. The final instruction in the function is an optional **ReturnStatement**.

def FunctionName (ParameterList):
  Action
  ReturnStatement

To get a better feel for how functions work, we will take a look at several examples. In the first example, we define a function

called **printFruit** that will simply print "Apple", "Banana", and "Cherry", each on a separate line. The **printFruit** function has no parameters and does not explicitly return a value. This definition creates the function, but we actually need to call it for the function to get triggered and run. To call the function, we provide the function name followed by parentheses. In the example below, we make three calls to the **printFruit** function.

```
# Define a function to print some fruit
def printFruit():
    print("Apple")
    print("Banana")
    print("Cherry")
# Make 3 calls to the printFruit function
printFruit()
printFruit()
printFruit()
```

The output from running the above program is displayed here:

```
Apple
Banana
Cherry
Apple
Banana
Cherry
Apple
Banana
Cherry
>>>
```

Some functions, such as **printFruit**, perform a task, but don't return an explicit value. Other functions computer a result (and may also

perform a task) and return that single result to the user. A Python function can actually return multiple values.

In our next example, we will write a function that converts a Fahrenheit temperature to its Celsius equivalent. You can think of a function as a mini program. The inputs to the function are its parameters. A **_parameter_** is simply a placeholder variable that is listed in parenthesis in the heading of the function definition. The output of the function is produced using a **_return value_**, which is a data item that is sent back to the function call.

In the **convertToCelsius** function, a parameter named **fahren** is passed to the function. The formula **(5/9)*(fahren-32)** is applied to that parameter and the result is stored in a variable named **cels**. This **cels** variable is then returned by the function.

After the function definition, the program illustrates three different ways you might make a call to **convertToCelsius**. Since the function has a parameter, the function call must send an argument to the function. An **_argument_** is the actual value passed to the function within parentheses when the function is called. (Note: Sometimes the parameter shown in the function definition is referred to as the **_formal parameter_** and the argument that is supplied when the function is invoked is referred to as the **_actual parameter_**.)

In the first function call, we actually **_hard-coded_** the number **65** as the function argument and included the function call within a print statement. The value that returned the function, **18.333**, was displayed. In the second function call, we passed **90** as the function argument, but this time we stored the return from the function, **32.222**, to the variable **amount**. In the third example, we used keyboard input from the variable **fahrenheit** as the argument to the function. With a user input of **28.3**, the result, **-2.055**, was returned to a variable named **celsius**.

```
# Function to convert from Fahrenheit to Celsius
def convertToCelsius (fahren):
    cels = (5/9)*(fahren-32)
    return cels

# Making calls to the convertToCelsius function

print("65 Fahrenheit is", convertToCelsius(65), "Celsius")

amount = convertToCelsius(90)
print("90 Fahrenheit is", amount, "Celsius")

fahrenheit = float(input("What is Fahrenheit temperature? "))
celsius = convertToCelsius(fahrenheit)
print(fahrenheit, "Fahrenheit is", celsius, "Celsius")
```

The output when running the above program with the user entering
28.3 at the keyboard when prompted:

```
65 Fahrenheit is 18.333333333333336 Celsius
90 Fahrenheit is 32.22222222222222 Celsius
What is Fahrenheit temperature? 28.3
28.3 Fahrenheit is -2.0555555555555554 Celsius
>>>
```

Modeling a Function using a Flowchart

When using flowcharts to design or document a program, you
should create one flowchart for each of the program's functions
and another flowchart for the main program. Later, we will show
how you can actually put the main body of the program in its
own function. A new rectangular symbol, labeled as "***process***" by
the **draw.io** website, is used to model a function call.

You should include the function name, parameters, as well as
a stored variable where appropriate in this shape. In the actual
function flowchart, the initial ellipse should include the function
name (with parameters) instead of the label "***start***." The final ellipse
should include the return statement instead of the label "***end***." The
flowchart below models the **convertToCelsius** program, although
only the third example is illustrated:
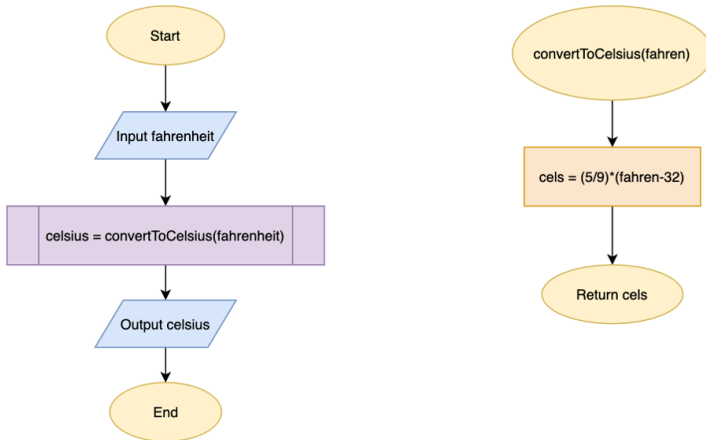
The following example uses a function **pay** to compute the weekly payroll. The hourly **wage** and the total **hours** are input to the function as parameters. The pay amount is computed, providing overtime pay for hours worked over 40. Once computed, that amount is returned.

The program asks the user to input **empWage** and **empHours** at the keyboard. These values are passed to the function as arguments. The amount returned is store to the variable **empPay** and then printed. The complete program and a sample run are shown next:

```python
# Function to compute the weekly pay
def pay (wage, hours):
    if hours <= 40:
        amount = wage * hours
    else:
        amount = (40*wage) + 1.5*wage*(hours-40)
    return amount

# Program to call the pay function

empWage = float(input("What is your hourly wage? "))
empHours = float(input("How many hours did you work? "))

empPay = pay(empWage, empHours)

print("Your pay is $", "%.2f"%empPay, sep="")
```

Here is the interaction that takes place when the program is run, and the user enters 9.35 as the hourly wage, and 27 as the weekly hours worked.

```
What is your hourly wage? 9.35
How many hours did you work? 27
Your pay is $252.45
>>>
```

INTERACTIVE – Function fun!

The program below illustrates several simple functions and calls made to them. Try to predict the output of the program and then run it to see if you nailed it!

> 🖳 *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=34*

The ***scope*** of a variable is the section of the program where that variable is visible. A variable that is first initialized in a block is said to be ***local*** to that block. It is not visible, or available, outside of the block. A variable that is accessible in the entire file is said to have ***global*** scope. The ***global keyword*** can be used to declare a variable as global, even if this declaration is made inside of a block. The following code illustrates several examples of scope:

```
# var1 is in the global namespace
var1 = 5

def funcA():
    # var2 is in the local namespace of funcA
    var2 = 6
    print(var1, var2)          # We can access var1 and var2

    def funcB():
        # var3 is in local namespace of funcB
        var3 = 7
       global var4 = 19        # Define var4 to be a global
variable
        print(var1, var2, var3) # We can access var1, var2, and
var3

    # Here we are in funcA - we cannot access var3

# Here we are outside of the function definitions

print (var1, var4)

# We cannot access var2 or var3 out here
```

Chapter Review Exercises:

8.1. Name six advantages of creating user-defined functions.

Programming Projects:

8.1. Modify Programming Project 6.2 in the following ways:

- Write a copyCost function that will have one argument, the number of copies. It should compute the cost based on the given formula and then return that cost.
- Write a main function that will ask for the customer's name and the number of copies. It will then make a call to your copyCost function and store the result to a variable. Finally, it should print the customer's name and the copy cost (using a dollar sign and 2 decimal places).
- Create a flowchart to model your solution.

9. Chapter 9: Lists and Dictionaries

LISTS AND DICTIONARIES

01100111 11111100 01111101 11011101 11011001 11001010 11111001 10100000 10010111 00100000 00010111 01000011 00011100
11100010 10011100 01100010 01011111 11010011 10001100 10001101 01110101 10010000 01011011 01110000 10111110 10110010 10110101 10011001
11001011 01000011 11100010 01011011 11000101 10001011 01000011 01000011 00111110 01001101 00000101 00010011 11001011 10011011
00100011 00011111 00001000 01011011 01001111 10100101 01111001 00111001 11001011 01101010 01001011 10111011 11001100 11001001
10111001 01010010 01000011 00000100 00011011 10011100 00101000 00010110 00111110 10000100 10001001 10101100 10011101 01100111 00000101
00100011 11001010 10000111 00011000 10011101 01110011 01011010 10001100 10101010 11101100 11100001 10100100 01001000 00110111 00100001
01000101 01010001 00100101 11110001 00101010 01010110 10101010 10101100 00111001 00110011 01101100 01011101 11011001
11001010 11101000 10011110 11101111 10100000 10010111 00100000 00010111 01000011 00011100 10011011 11110000 10011001 1110011 11010001
10001100 10001101 01110101 10010000 01011011 01110000 10111110 10110010 10110101 10011001 10011001 11001011 01000011 11100010 01010111 11001011
10010011 01000011 11001011 00111100 01001101 00000010 11100100 00011010 10010010 10000010 00100011 00011111 00001000 01010111 01001111
10100101 01111001 00111001 11001011 01101010 01001011 10111011 11001100 11001001 10111001 01010010 01000011 00000100 00011011 10011100
00101000 00010110 00111110 10000100 10001001 10101100 10011101 01100111 00000101 00100011 11001010 10000111 00011000 10011101 01110011
01011010 10001100 10101010 11101100 11100001 10100100 01001000 00110111 00100001 01000101 01010001 00100101 11110001 00101010 01010110
10101010 10101100 00111001 00110011 01101100 01011101 11011001 11001010 11101000 10011110 11101111 10100000 10010111 00100000 00010111
01000011 00011100 10011011 11110000 10011001 11100011 11010001 10001100 10001101 01110101 10010000 01011011 01110000 10111110 10110010
10110101 10011001 10011001 11001011 01000011 11100010 01010111 11001011 10010011 01000011 11001011 00111100 01001101 00000010 11100100
00011010 10010010 10000010 00100011 00011111 00001000 01010111 01001111 10100101 01111001 00111001 11001011 01101010 01001011 10111011
11001100 11001001 10111001 01010010 01000011 00000100 00011011 10011100 00101000 00010110 00111110 10000100 10001001 10101100 10011101
01100111 00000101 00100011 11001010 10000111 00011000 10011101 01110011 01011010 10001100 10101010 11101100 11100001 10100100 01001000
00110111 00100001 01000101 01010001 00100101 11110001 00101010 01010110 10101010 10101100 00111001 00110011 01101100 01011101 11011001
11001010 11101000 10011110 11101111 10100000 10010111 00100000 00010111 01000011 00011100 10011011 11110000 10011001 11100011 11010001
10001100 10001101 01110101 10010000 01011011 01110000 10111110 10110010 10110101 10011001 10011001 11001011 01000011 11100010 01010111
11001011 10010011 01000011 11001011 00111100 01001101 00000010 11100100 00011010 10010010 10000010 00100011 00011111 00001000 01010111
01001111 10100101 01111001 00111001 11001011 01101010 01001011 10111011 11001100 11001001 10111001 01010010 01000011 00000100 00011011
10011100 00101000 00010110 00111110 10000100 10001001 10101100 10011101 01100111 00000101 00100011 11001010 10000111 00011000 10011101
01110011 01011010 10001100 10101010 11101100 11100001 10100100 01001000 00110111 00100001 01000101 01010001 00100101 11110001 00101010
01010110 10101010 10101100 00111001 00110011 01101100 01011101 11011001 11001010 11101000 10011110 11101111 10100000 10010111 00100000
00010111 01000011 00011100 10011011 11110000 10011001 11100011 11010001 10001100 10001101 01110101 10010000 01011011 01110000 10111110
10110010 10110101 10011001 10011001 11001011 01000011 11100010 01010111 11001011 10010011 01000011 11001011 00111100 01001101 00000010
11100100 00011010 10010010 10000010 00100011 00011111 00001000 01010111 01001111 10100101 01111001 00111001 11001011 01101010 01001011
10111011 11001100 11001001 10111001 01010010 01000011 00000100 00011011 10011100 00101000 00010110 00111110 10000100 10001001 10101100
10011101 01100111 00000101 00100011 11001010 10000111 00011000 10011101 01110011 01011010 10001100 10101010 11101100 11100001 10100100
01001000 00110111 00100001 01000101 01010001 00100101 11110001 00101010 01010110 10101010 10101100 00111001 00110011 01101100 01011101

Topics Covered:

- List basics
- Slicing a list
- Deleting from a list
- Adding to a list
- Processing a list
- List operators

List basics

A **_list_** is an ordered sequence of objects. You can use a list to store multiple items using a single variable. The Python list has many similarities to an array, which is provided by most programming

languages. The main difference is that every item in an array must have the same data type while a list may contain objects of different types. If you know the values that you want to store in a list, you can create a list by enclosing the set of values, separated by commas, between square brackets, like this:

```
>>> school = ["UNA", 8014, "Lions", 1879, "Florence"]
>>> print(school)
['UNA', 8014, 'Lions', 1879, 'Florence']
>>>
```

The variable school now stores five data items associated with a university. Similar to a string, you can access individual elements of the list by using an *index*, or *subscript*. In the example below, we access the first element (using index **0**) and the fourth element (using index **3**). When attempting to access the element using index **5**, Python indicates an error has occurred since the index is out of range:

```
>>> school[0]
'UNA'
>>> school[3]
1879
>>> school[5]
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    school[5]
IndexError: list index out of range
>>>
```

Python provides several statistical functions that come in handy when processing data in a list. In the code below, we create two more lists and then try out the **len()**, **max()**, **min()**, and **sum()** functions. These functions are pretty straightforward. As discussed in chapter three, the max and min of strings use ASCII codes. The program ran into trouble when it tried to find the **sum()** of a list of strings. The **sum()** function only works on numeric lists.

```
>>> nums=[19,47,3,19,26]
>>> animals=["mouse","zebra","dog","aardvark","fish","cat","horse"]
>>> len(nums)
5
>>> len(animals)
7
>>> max(nums)
47
>>> min(nums)
3
>>> max(animals)
'zebra'
>>> min(animals)
'aardvark'
>>> sum(nums)
114
>>> sum(animals)
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    sum(animals)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Slicing a List

In the same way you can use square brackets **[]** and the colon (:) operator to extract slices of strings, you can retrieve slices of lists. The example below shows several slice examples. Examine each one closely to verify your understanding of how slices work.

```
>>> animals
['mouse', 'zebra', 'dog', 'aardvark', 'fish', 'cat', 'horse']
>>> animals[3:6]
['aardvark', 'fish', 'cat']
>>> animals[2:]
['dog', 'aardvark', 'fish', 'cat', 'horse']
>>> animals[:4]
['mouse', 'zebra', 'dog', 'aardvark']
>>> animals[0:7:2]
['mouse', 'dog', 'fish', 'horse']
>>> animals[::-1]
['horse', 'cat', 'fish', 'aardvark', 'dog', 'zebra', 'mouse']
>>>
```

To further illustrate the concept of object-oriented programming, we will take a look at a couple of functions, or *methods*, that are associated with the *list* data type. As we saw when using methods for string variables, to use a method, you reference the list variable, followed by a dot, followed by the method name. Using the same **nums** list from previous examples, first we examine

the **count()** method. You provide a value to the **count()** method as an argument, and it will return the number of occurrences of that value. The second example is the **index()** method. Again, you pass a value as an argument, but the **index()** method will return the position of that value in the list. If the value occurs multiple times, it returns the position of the first occurrence. If the value does not exist in the list, you can see from the example that it will cause the program to crash.

```
>>> nums = [19,47,3,19,26]
>>> nums.count(19)
2
>>> nums.count(47)
1
>>> nums.count(88)
0
>>> nums.index(26)
4
>>> nums.index(19)
0
>>> nums.index(55)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    nums.index(55)
ValueError: 55 is not in list
>>>
```

Deleting from a List

There are two ways to remove an item from a list. You can use the **del** instruction if you want to delete an item based on its subscript. In the next example, we first delete the fourth (using subscript **3**) item from the **fruit** list. After that, we get an error message since we tried to delete the tenth item (using subscript **9**), but there were only four items in the list.

Instead of using the subscript, you can also remove an item based on a value. In this case, you use the **remove()** method and pass the

value to remove as the argument. If the item exists more than once in the list, only the first occurrence will be deleted. If the item does not exist in the list, an error will occur, as shown in the example that follows:

```
>>> fruit = ["apple", "Banana", "Cherry", "Grape", "Orange"]
>>> fruit
['apple', 'Banana', 'Cherry', 'Grape', 'Orange']
>>> del fruit[3]
>>> fruit
['apple', 'Banana', 'Cherry', 'Orange']
>>> del fruit[9]
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    del fruit[9]
IndexError: list assignment index out of range
>>> fruit.remove("Banana")
>>> fruit
['apple', 'Cherry', 'Orange']
>>> fruit.remove("Melon")
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    fruit.remove("Melon")
ValueError: list.remove(x): x not in list
>>>
```

Adding to a List

There are multiple ways to add items to a list. The **append()** method is a simple technique that just adds the required argument to the end of a list. In our next example, we use append to add the value **33** to the end of a five-item list named **things**.

The **insert()** method has two arguments, the position in the list to place the new item and the value to be inserted. In the second example, we inserted the value "tiger" at position **2**, which is actually the third item in the list. When the new list is displayed, "tiger" squeezes into position **2** and all of the items after it are pushed back one slot.

A third way to add items to a list is to use the **extend()** method. This function is similar to the **append()** method except the argument to be passed is another list instead of a single data item. In the example, we created a four-item list named **birds** and then

passed that list to the **extend()** method. The new contents of **things** include the seven items it previously stored plus the four items of the **birds** list.

```
>>> things = [1,2,3,4,5]
>>> things
[1, 2, 3, 4, 5]
>>> things.append(33)
>>> things
[1, 2, 3, 4, 5, 33]
>>> things.insert(2,"tiger")
>>> things
[1, 2, 'tiger', 3, 4, 5, 33]
>>> birds = ["cardinal", "oriole", "blue jay", "robin"]
>>> things.extend(birds)
>>> things
[1, 2, 'tiger', 3, 4, 5, 33, 'cardinal', 'oriole', 'blue jay', 'robin']
>>>
```

List Processing

The code below illustrates three more useful Python list methods. The first one, sort, arranges the list items in ascending order. The **sort()** method can also be applied on a list of strings. The **reverse()** method does exactly what you might guess. It just flips the order of the list. Finally, the **clear()** method will delete all of the items from a list. You can see from the example below that a list is empty when it displays as just two brackets [].

```
>>> items = [47, 82, 23, 91, 52, 75, 38, 60]
>>> items
[47, 82, 23, 91, 52, 75, 38, 60]
>>> items.sort()
>>> items
[23, 38, 47, 52, 60, 75, 82, 91]
>>> items.reverse()
>>> items
[91, 82, 75, 60, 52, 47, 38, 23]
>>> items.clear()
>>> items
[]
>>>
```

List Operators

Similar to the concatenation of strings, the plus (+) operator can be used to combine two lists. In the first example below, you can observe the instruction **first + second** is used to combine those two lists.

```
>>> first = ["x",10]
>>> second = ["y",25,"z"]
>>> first + second
['x', 10, 'y', 25, 'z']
>>>
```

In the final example, we use the multiplication **(*)** operator to perform repetition. Once again, this mimics the repetition functionality this operator applies with strings. Although the order does not matter, the operation requires a list and an integer. As you can see from the examples below, the list **[1,2,3]** is repeated five times. It is often convenient to use this repetition operator to initialize a list. For example, the final instruction initializes a 100-item list named **pay** to 100 values of zero.

```
>>> [1,2,3]*5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5*[1,2,3]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> pay = [0]*100
>>> pay
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>>
```

To conclude the chapter, we will build a full application to solve a problem:

Problem: The annual Lion Bass Fishing Tournament has hired you to write an application to help process the tournament statistics. Specifically, each team will use your application to input the weight of each fish they catch. After entering a fish weight, the program will ask the user if there are any more fish weights to enter. When the tournament concludes, the program will display summary statistics

for that team. This will include a table with the weights of all fish caught, as well as the fish count, the total weight, the heaviest fish, the lightest fish, and the average weight.

On the next page, we will first model a solution to this problem with a flowchart. This is the coolest and most complex flowchart we have seen so far. Of course, it begins with the "**Start**" symbol and then initializes variables **more** and **weights**. You should be able to follow along with each symbol of the flowchart and then see how the flowchart naturally translates into the working Python program.
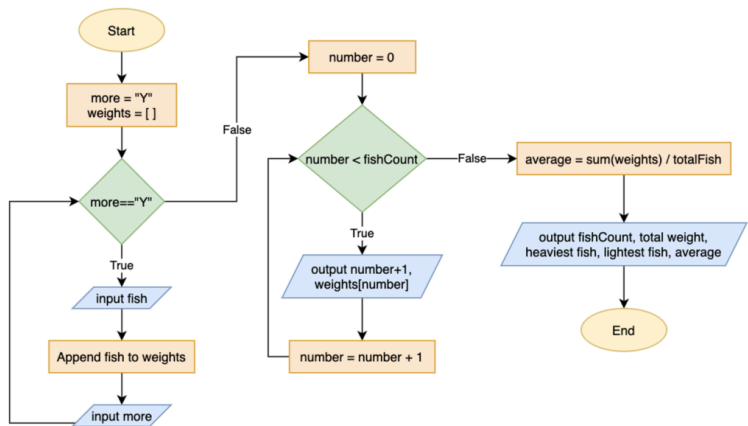
Flowchart to Model the Bass Fishing Tournament.

```
# Program to generated bass fishing tournament summary
# Written by XXX XXXXX
# CS101 — Introduction to Computer Programming
# Date: XX/XX/XXXX
print("Bass Fishing Tournament")
print()
weights = []
more = "Y"
while more.upper()=="Y":
    fish = float(input("Weight of the fish? "))
    weights.append(fish)
    more = input("Any more fish (Y/N)? ")

print()
totalCaught = len(weights)
print("No. Weight")
for number in range(0,totalCaught):
    print("%2d"%(number+1), "%6.1f"%weights[number])

print()
print("Total fish caught:", totalCaught)
print("Total weight:", "%.1f"%sum(weights))
print("Heaviest fish:", max(weights))
print("Lightest fish:", min(weights))
average = sum(weights)/totalCaught
print("Average weight:", "%.1f"%average)
```

In the program, you will notice how we took advantage of many of the list functions and methods discussed in the chapter. To add each fish to the **weights** list, we used the **append** method. To find the fish count, total weight, heaviest fish, and lightest fish, we used the **len**, **sum**, **max**, and **min** functions, respectively. Below, we will show the output of a sample run of the program. Notice how we used **upper** string method to convert all responses to **more** to uppercase to make our program more user friendly, so the keyboard user doesn't need to worry about the 'Y' or 'N' input being upper or lower case.

```
Bass Fishing Tournament

Weight of the fish? 9.7
Any more fish (Y/N)? Y
Weight of the fish? 11.8
Any more fish (Y/N)? y
Weight of the fish? 10
Any more fish (Y/N)? Y
Weight of the fish? 13.5
Any more fish (Y/N)? y
Weight of the fish? 12
Any more fish (Y/N)? N

No. Weight
 1    9.7
 2   11.8
 3   10.0
 4   13.5
 5   12.0

Total fish caught: 5
Total weight: 57.0
Heaviest fish: 13.5
Lightest fish: 9.7
Average weight: 11.4
>>>
```

Dictionaries

Similar to a list, a **dictionary** is another Python data type that is used to store a collection of data. A a data structure that is also known as an associative array, the dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

In the example below, a dictionary named **school** is created with four key-value pairs. To initialize a dictionary, you use curly brackets to enclose the data and a colon to separate each key-value pair. The example shows how to access the value of an item, as well as illustrates how to use dictionary functions and methods.

```python
school = {
    "name": "Univerity of North Alabama",
    "founded": 1830,
    "enrollment": 8832,
    "nickname": "Lions"
}

print ("The school dictionary looks like this:")
print (school)
print ()

school["enrollment"] = 8847
print ("The new school enrollment is:")
print (school["enrollment"])

print ()
print("Length:", len(school))
print ()
print("Keys:", school.keys())
print ()
print("Values:", school.values())
print ()
print("Items:", school.items())
print()
for piece in school:
    print("key:", piece, "value:", school[piece])

school.pop("enrollment")
print(school)
```

Program illustrating a Python Dictionary.

The output when the above program is executed is seen here:

```
The school dictionary looks like this:
{'name': 'Univerity of North Alabama', 'founded': 1830, 'enrollment': 8832,
'nickname': 'Lions'}

The new school enrollment is:
8847

Length: 4

Keys: dict_keys(['name', 'founded', 'enrollment', 'nickname'])

Values: dict_values(['Univerity of North Alabama', 1830, 8847, 'Lions'])

Items: dict_items([('name', 'Univerity of North Alabama'), ('founded', 1830)
, ('enrollment', 8847), ('nickname', 'Lions')])

key: name value: Univerity of North Alabama
key: founded value: 1830
key: enrollment value: 8847
key: nickname value: Lions
{'name': 'Univerity of North Alabama', 'founded': 1830, 'nickname': 'Lions'}
```

Program output.

Often times, we like to use the key as a way to access data directly. In the example below, we will create a dictionary named **states** that stores the population of U.S. states from the year 2000 through 2019. This data was found at a web site called Kaggle, that has thousand of freely accessible data sets that you can use for programming projects. A small chunk of this comma-separated data file is shown here:

```
state_name,state_FIPS,pop_2000,pop_2001,pop_2002,pop_2003
Alabama,1000,4451849,4464034,4472420,4490591,4512190,4545(
Alaska,2000,627499,633316,642691,650884,661569,669488,677.
Arizona,4000,5166697,5304417,5452108,5591206,5759425,5974{
Arkansas,5000,2678288,2691068,2704732,2722291,2746161,277(
California,6000,33994571,34485623,34876194,35251107,35558{
Colorado,8000,4328070,4433068,4504265,4548775,4599681,466(
Connecticut,9000,3411726,3428433,3448382,3467673,3474610,.
Delaware,10000,786411,794620,804131,814905,826639,839906,{
District of Columbia,11000,571744,578042,579585,577777,57!
Florida,12000,16047118,16353869,16680309,16981183,1737525!
Georgia,13000,8230161,8419594,8585535,8735259,8913676,909'
Hawaii,15000,1211566,1218305,1228069,1239298,1252782,1266.
Idaho,16000,1299551,1321170,1342149,1364109,1391718,142581
Illinois,17000,12437645,12507833,12558229,12597981,1264529
Indiana,18000,6091649,6124967,6149007,6181789,6214454,625.
```

The populations.csv data file.

In our example, we use the state name (data[0]) as the dictionary

key. The dictionary value is actually a 22-component list that consists of the state name, a state code, and then the populations for that state between 2000 and 2019. In our program, we just print the state name and the state's 2019 population. Once this loop completes, we illustrate how you can obtain all of the data for a given state, taking the user input and using it as the dictionary key.

```python
# state_populations_2000_to_2019.csv
# https://www.kaggle.com/yassershrief/us-population-20002019

states = {}
linenum=0
print("STATE                   2019 POPULATION")
infile = open("populations.csv","r")
for item in infile:
    if linenum != 0:
        data = item.split(",")
        states[data[0]] = item
        population2019 = "{:,}".format(int(data[21]))
        print("%-20s"%data[0], "%15s"%population2019)
    linenum += 1


myState=input("Enter a state: ")
print(states[myState])
```

Python program that reads CSV file and stores data to a dictionary.

A portion of the program output is shown below. In this example, the user entered Ohio as input and the program displayed the data for that state.

```
Oregon                  4,217,737
Pennsylvania           12,801,989
Rhode Island            1,059,361
South Carolina          5,148,714
South Dakota              884,659
Tennessee               6,829,174
Texas                  28,995,881
Utah                    3,205,958
Vermont                   623,989
Virginia                8,535,519
Washington              7,614,893
West Virginia           1,792,147
Wisconsin               5,822,434
Wyoming                   578,759
Enter a state: Ohio
Ohio,39000,11363844,11396874,11420981,11445180,11464593,11475262,11492495,11
520815,11528072,11542645,11539327,11543463,11548369,11576576,11602973,116178
50,11635003,11664129,11689442,11689100
```

Program output.

INTERACTIVE – Python collection types

Four common data types to store collections of data in Python are illustrated in the program below. Observe the code and then run the program. Do a quick Internet search to see if you can identify the properties that differentiate these four collections.

> 📥 *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=36*

Chapter Review Exercises:

9.1. What is the output of the following Python code?

myList = [8, 3, 14, 5, 10]
print (len(myList))

9.2. What is the output of the following Python code?

myList = [8, 3, 14, 5, 10]
print (sum(myList))

9.3. What is the output of the following Python code?

myList = [8, 3, 14, 5, 10]
myList.clear()
print (len(myList))

10. Chapter 10: Data Files

DATA FILES

Topics Covered:

- Data file as input
- Creating a text file
- Reading from a text file
- Data file as output

Data File as Input

In all of the programs we have written so far, the user input has come from the keyboard. There are many scenarios, however, when we will instead have the program input come from a data file. We will learn to have our Python programs read data from a ***text file***, which is sometimes called an **ASCII *file***. This type of file simply

stores all of the data as raw text, one byte per character, using ASCII codes.

Here are some uses and advantages of having a text file the source for program input:

- **Ease of testing**: When thoroughly testing a program, you typically need to enter several input values from the keyboard. After making a correction to your program, you need to type those input values in again. Each time you run your program, you type those values over and over. By using a text file input, you can type the input into a file one time and your input is complete.
- **Large inputs**: You may need to write a program in which the input is simply too large to practically and accurately enter it from the keyboard.
- **Import data from other systems**: Many times, you can use input text files from other sources, such as a web site or another application's output. Quite often, data from spreadsheets and databases can be saved to a text file and then used as the input to your program.

Creating a Text File

Most computer systems include a **text editor**, which is a program that allows you to create, edit, save, and retrieve text files. Windows-based computer systems provide the **Notepad** program, while Macintosh computers ship with the **TextEdit** application. Your Python programs, by the way, are text files. You can even use your **Idle** environment to create a text file. Suppose you have a text file named **"artists.txt"** that looks like this:

Bob Dylan
Elvis Presley
Chuck Berry
Jimi Hendrix
James Brown

Aretha Franklin
Ray Charles

Reading from a Text File

Python makes it very easy to read data from a text file. The first step when accessing a text file is to open the file. The **open** function has two parameters, the name of the external data file and the file access mode. The file should be stored in the same directory, or folder, as your Python program. If the file is stored in another location, the path to the file must accompany the file name. In our first code example, the external data file is named *"artists.txt"*.

To indicate the file access mode, a single-character string is used based on whether the file will be opened in read (r), write (w), or append (a) mode. In this section we are reading from text files, so we will use mode "r". The **open** function returns a *file handle* that must be stored in a variable. In the example below, we stored the file handle as **infile**.

In our example program, we can use a **for** loop to read the file, one line at a time. In the **for** loop, we will store each line to a list named **dataList**. Once the program is done reading data from the file the **close()** method of the file handle should be invoked to disconnect the file from your application. In our program below, we include a second **for** loop that simply prints out the elements of **dataList**. In each iteration of the **for** loop, one item is retrieved from the list and stored in **artist**.

```
infile = open("artists.txt", "r")
dataList = []
for line in infile:
    dataList.append(line)
infile.close()
for artist in dataList:
    print(artist)
```

The output when this program is executed:

```
Bob Dylan

Elvis Presley

Chuck Berry

Jimi Hendrix

James Brown

Aretha Franklin

Ray Charles

>>>
```

A couple of observations from running the program: (1) When hitting

the **F5** key to run the program, it immediately executed and showed the results with no keyboard input, and (2) the output appeared with blank lines between each artist. When each line was read from the file, the newline character '\n' remained at the end of the string input. Since the print function automatically adds a newline when displaying output, that second newline created the extra space, which gave a "double spacing effect.

A common technique to remove the newline from the end of a string is to apply the **rstrip()** string method. This removes any trailing whitespace characters at the end of the string. We will apply this method in our next example, as well as add a counter and some string formatting to create a table with simple headings.

To break up a string into multiple pieces based on some delimiting character or string, we can use the **split()** method. The method has a single parameter, the delimiting string, such as a space, comma, or tab. The result of the split method is a list. The list will contain each of the broken-up pieces of the string without the delimiting strings. Let's look at an example. After the split, the list **pieces** now contains four strings.

```
>>> thing = "Python,programming,is,fun"
>>> pieces = thing.split(",")
>>> pieces
['Python', 'programming', 'is', 'fun']
>>>
```

Here's the new and improved version of our program for displaying the contents of the **"artists.txt"** file:

```
infile = open("artists.txt", "r")
dataList = []

for line in infile:
    dataList.append(line.rstrip())
infile.close()

count=1
print("No. First    Last")
for artist in dataList:
    data = artist.split(" ")
    print("%-3d"%count, "%-8s"%data[0], "%-8s"%data[1])
    count = count + 1
```

The output of the program run is shown next:

```
No. First    Last
1   Bob      Dylan
2   Elvis    Presley
3   Chuck    Berry
4   Jimi     Hendrix
5   James    Brown
6   Aretha   Franklin
7   Ray      Charles
>>>
```

INTERACTIVE – Dealing with a fruit file

Okay, now that you have seen some examples of a data file being used as input, see if you can predict the exact output of the program below. Look closely at the tabs in the Trinket window so you can examine the "fruits.txt" data file.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=38

Data File as Output

As with input, there are many uses and advantages to having your program output get sent to a text file:

- **Too much output**: Sometimes your program will produce so much output that it is difficult to view on the screen as it scrolls on by. Saving the output to a file allows you to view it at your leisure.
- **Have permanent record**: When you're done running your code and you shut down Python, all of your program output disappears. If you would like to store the results of your program runs permanently, you could store that output to a file.
- **Export program output for various uses**: Once your program output is saved as a text file, you can then open that file with other software, such as a word processor, spreadsheet, statistical tool, etc.

When your program is going to create an output file, it must first use the **open()** function to create a file handle, using the "w" parameter to indicate you will be writing to the file. If you use write mode and the external data file already exists, it will be overwritten, and the former file contents will be lost. The append (a) mode is used if you want your program to just tack data onto the end of an existing file. If that external data file does not exist in append mode, it will be created.

The write method is used to send a string parameter to a text file. The program below will open a file named **"test.txt"** in the write mode. It then writes two strings to that file and closes the file.

```
dataFile = open("test.txt", "w")
dataFile.write("CS101")
dataFile.write("Rocks!")
dataFile.close()
```

The program output:

>>>
· · ·

Since this program had no print statements in it, there was no screen output from the program. When you hit the **F5** key to run the program, you will see the >>> prompt. This is a little disconcerting at first. To evaluate our program, though, we just need to use a text editor to examine the output file **"test.txt"**. Below, we show the contents of this file. Differing from the **print** function, the **write()** method does not add a newline to the end of the output.

The test.txt data file:

CS101Rocks!

Problem: The Roaring Lions Run Club has hired you to develop a running activity log application. When a runner completes a run, she will need to input: (1) the date, (2) the distance in miles, and the time in (3) minutes and (4) seconds. The program will compute the average mile pace for the run. It will then output this average mile pace to a data file, along with the four inputs. Each running activity should get logged as a comma-separated line at the end of the file. Having comma separated values (CSV) allows this text file to be opened by many common programs, such as a spreadsheet.

Knowing that this single file will have data added to it over a long period of times indicates to a programmer to use the append (a) file access mode when the file is opened. There is a little math involved when we calculate the average mile pace. Before the coding process begins, we must have a plan for this computation. Let's take a look at a test case and go through the necessary formulas using paper and pencil:

INPUTS: Miles $\boxed{3.1}$ Minutes $\boxed{29}$ Seconds $\boxed{47}$

$$totalMinutes = minutes + seconds/60 = 29 + \frac{47}{60} \approx \mathbf{29.783}$$

$$minute\ Pace = \left\lfloor totalMinutes/miles \right\rfloor \approx \left\lfloor \frac{29.783}{3.1} \right\rfloor = \boxed{9}$$

$$seconds\ Pace = \frac{totalMinutes - minutePace * miles}{miles} * 60 \approx$$

$$\frac{29.783 - 9*3.1}{3.1} * 60 \approx \boxed{47}$$

Illustrating the Computation of Run Pace with a Test Case.

The main construct in this program is a **while** loop that allows the user to log as many runs as they wish. During each iteration of the loop, the user will provide the four input values discussed above. The average mile pace is then computed as illustrated in the test case. Note that we needed to import the **math** module in order to use the **floor()** method. An **outputString** is created to store the five items that will be sent to the file along with the comma separators. This string terminates with a newline so that each logged activity will appear on its own line. Since that instruction was so long, Python allows you to continue it on a second line by using the backslash (\) character. When the user has no more activities to log, the loop concludes and the **close()** method for the **dataFile** is called. The full program is shown next:

```
# Program to log running activities
# Written by XXX XXXXX
# CS101 — Introduction to Computer Programming
# Date: XX/XX/XXXX
import math
print("Running Log App")
print()
dataFile = open("runningLog.csv", "a")
more = "Y"

while more.upper()=="Y":
    date = input("Date of your run (MM/DD/YYYY): ")
    miles = float(input("Distance of run (in miles): "))
    minutes = int(input("Time of your run (type minutes and hit ENTER): "))
    seconds = int(input("Type seconds and hit ENTER: "))
    totalMinutes = minutes+seconds/60.0
    minutesPace = math.floor(totalMinutes / miles)
    secondsPace = round((totalMinutes-minutesPace*miles)/miles*60)
    print("Average pace: ", minutesPace, ":", "%02d"%secondsPace, sep="")
    outputString=date+","+str(miles)+","+str(minutes)+","+str(seconds)+","  \
                +str(minutesPace)+":"+"%02d"%secondsPace+"\n"
    dataFile.write(outString)
    more = input("Any more runs to log (Y/N)? ")

dataFile.close()
```

Each time the program is executed, new run records are added to the **"activities.csv"** data file. Suppose the user had already entered a couple of runs before they ran the program with the following interaction:

```
Running Log App

Date of your run (MM/DD/YYYY): 3/20/2021
Distance of run (in miles): 3.1
Time of your run (type minutes and hit ENTER): 28
Type seconds and hit ENTER: 33
Average pace: 9:13
Any more runs to log (Y/N)? y
Date of your run (MM/DD/YYYY): 3/22/2021
Distance of run (in miles): 3.1
Time of your run (type minutes and hit ENTER): 28
Type seconds and hit ENTER: 09
Average pace: 9:05
Any more runs to log (Y/N)? n
>>>
```

After running the program multiple times, **"activities.csv"** looks like this:

```
3/13/2021,3.1,29,47,9:36
3/15/2021,4,38,40,9:40
3/20/2021,3.1,28,33,9:13
3/22/2021,3.1,28,9,9:05
```

As mentioned, a CSV file is an ASCII file that can be opened or imported into many applications. We opened **"activities.csv"** using Excel and it was pretty easy to make the chart below:
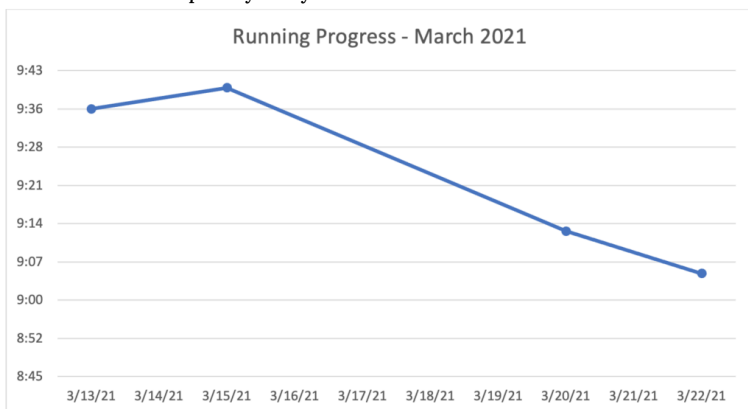
In chapter 12, we will look at how you could create charts like the one above using the Python Turtle graphics module. In chapter 13, we examine how to turn that text-based interaction into a graphical user interface (GUI) that you are used to using with everyday apps. Finally, in chapter 14, we take a look at how to turn this app into a web-based application.

Chapter Review Exercises:

10.1. Name three uses or advantages of using a text file for program input.

10.2. Name three uses or advantages of using a text file for program output.

Programming Projects:

10.1. A text file named "superbowl.txt" has a list that includes the winner of every Super Bowl. The first few lines of the file look like this:

```
Packers
Packers
Jets
Chiefs
Colts
Cowboys
Dolphins
   …
```

Write a Python program that will read the text file and print out all of the winners, along with the Super Bowl number and the year. The 1st Super Bowl occurred during the 1966 season. Display tabs between each column and include column headings. The first few lines of the program output should look like this:

```
No.   Year Winner
1     1966 Packers
2     1967 Packers
3     1968 Jets
4     1969 Chiefs
5     1970 Colts
6     1971 Cowboys
7     1972 Dolphins
…     …    …
```

11. Chapter 11: Making Computer Games

MAKING COMPUTER GAMES
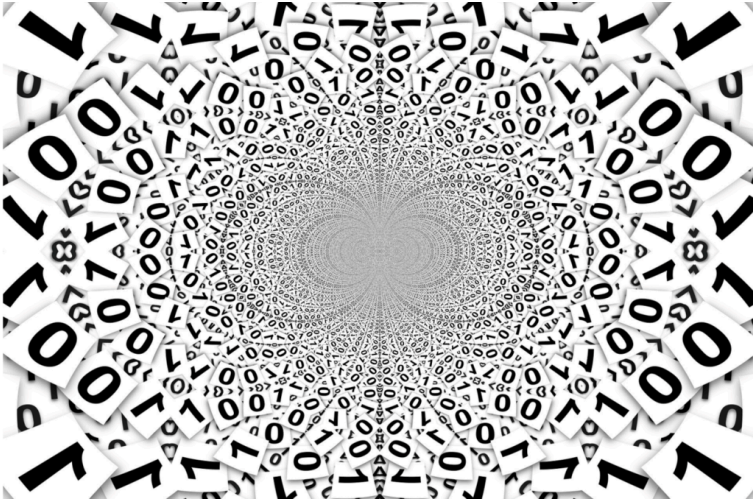
Topics Covered:

- Random numbers
- Guess the number game
- Paper rock scissors game

In this chapter, we are going to discuss creating computer games. Of course, playing computer games is fun and entertaining. Many computer games also have skill-building and educational aspects, too. Before we create our first computer game, we must first talk about random numbers.

Random Numbers

If you are playing a dice game, you wouldn't want the same dice rolls to show up every game. In a card game, you don't want to be dealt the same hand every game. If you're chasing zombies, you wouldn't want them to follow the same path every time. To make games more fun, interesting and challenging, you need random, unpredictable events to occur. Most programming languages include a random number generation module that provides one or more functions for producing random numbers. Not only do random numbers play an important role in computer-based game programs, they are also widely used to create programs that simulate real-world situations, including business and scientific applications.

Python provides a module named **random** that you can use to create random events. To be clear, a computer's random number generator is not truly random. It is simply a mathematical algorithm that will produce numbers or events that appear to be random. Because of this face, we often call one of these functions a *pseudorandom number generator* (PRNG). To use the random number features, your first must import the random module.

Although the reference **random.random()** may initially seem awkward, we are just calling the **random()** method from the **random** module. The random method will generate a floating point number in the range [0..1), which means a number greater than or equal to **0** but less than **1**. In the example below, we make three calls to the random method:

```
>>> import random
>>> random.random()
0.03558254816490958
>>> random.random()
0.48436256882937345
>>> random.random()
0.9900826631175004
>>>
...
```

Most programming languages will use the computer's clock to initialize, or *seed*, the random number generator. This gives you different random numbers every time you run your program, which is usually what you want. Occasionally, you want to reproduce the same random numbers. In this case, you can manually give the random number generator a seed, which is an integer, to initialize the PRNG. Below, we seed the PRNG with the value of 100 and call the random method twice. Then, we again seed the function with 100 and call the random method two more times. As you can see, the numbers generated are identical.

```
>>> random.seed(100)
>>> random.random()
0.14566692551041303
>>> random.random()
0.45492700451402135
>>>
>>>
>>> random.seed(100)
>>> random.random()
0.14566692551041303
>>> random.random()
0.45492700451402135
>>>
```

If you want random integers generated instead of floating point numbers, you can call the **randint()** method. It accepts two parameters, **a** and **b**, and generates an integer in the range [**a..b**]. In the example below, we make three calls of **randint(10,20)** to generate integers in the range **[10..20]**. Since this range is in square brackets **[]**, that means the end points are included. It's possible that the value **10** or **20** could be generated.

```
>>> random.randint(10,20)
12
>>> random.randint(10,20)
16
>>> random.randint(10,20)
15
>>>
```

The **randrange()** method will generate random integers using three parameters, **a**, **b**, and **step**. It will generate numbers in the range **[a..b)** such that each number is **a+k*step**, where **k** is a non-negative integer, and the number is less than **b**. Below, we generate four multiples of **10** that are less than **50**:

```
>>> random.randrange(0,50,10)
20
>>> random.randrange(0,50,10)
30
>>> random.randrange(0,50,10)
10
>>> random.randrange(0,50,10)
0
>>>
```

Here we generate three random odd numbers between 1 and 99, inclusive:

```
20
>>> random.randrange(1,100,2)
31
>>> random.randrange(1,100,2)
57
>>> random.randrange(1,100,2)
81
>>>
```

Guess the Number Game

We know enough about random numbers at this point to develop our first game. In the "Guess the Number" game, the computer will generate a random number between 1 and 100 and the user's job is to guess it. Each time the user guesses, the program will notify the user if his guess was "too high", "too low", or if he guessed it correctly. Before looking at the code, first we will check out a flowchart modeling the solution:
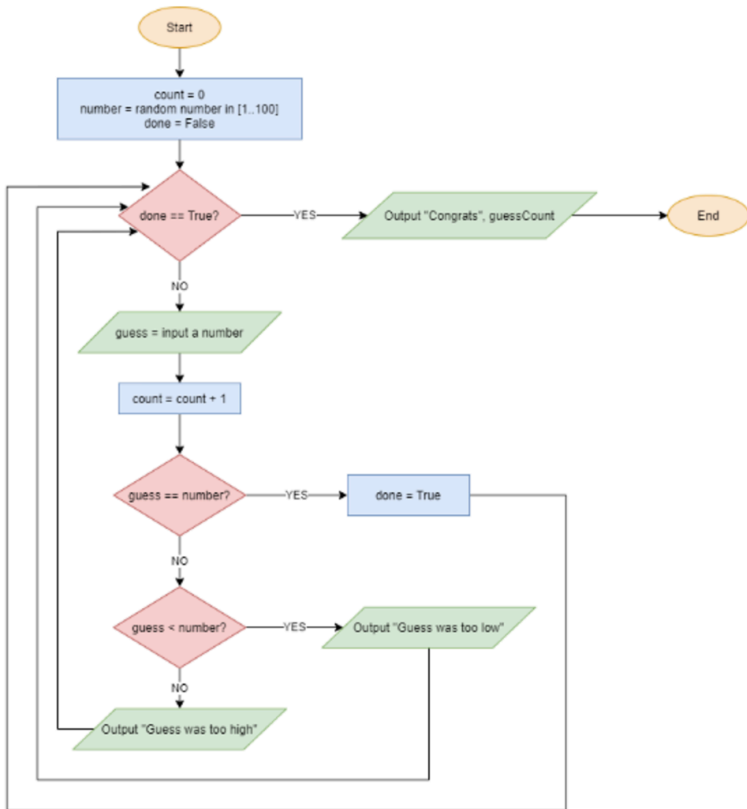
Flowchart Modeling the Guess the Number Game

We will use a Boolean variable named **done** as the condition of our **while** loop. The program should continue looping until the user successfully guesses the **number**. Our loop first asks the user for his **guess**. It increments a guess **count** variable. It then goes through a series of **if** statements that compare the user **guess** to the **number**. When the **guess** is correct, **done** is set to **True**, the program exits the loop and lets the user know how many guesses he took. The code for the program is shown here:

```
# The Guess the Number Game program - Version 1
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX
import random

num = random.randint(1,100)
guesses = 0
done = False
print ("I am thinking of a number between 1 and 100.")
print ("Can you guess it?")
while (not done):
    guesses = guesses + 1
    guess = int(input ("guess? "))
    if (guess == num):
        done = True
    elif (guess < num):
        print ("Your guess was too low")
    else:
        print ("Your guess was too high")

print ("Yea! It took you", guesses, "guesses.")
```

Here is a sample run of the program:

```
I am thinking of a number between 1 and 100.
Can you guess it?
guess? 37
Your guess was too high
guess? 19
Your guess was too low
guess? 24
Your guess was too low
guess? 29
Your guess was too high
guess? 27
Yea! It took you 5 guesses.
>>>
```

```
# The Guess the Number Game program - Version 2
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX
import random
def game():
    num = random.randint(1,100)
    guesses = 0
    done = False
    print ("I am thinking of a number between 1 and 100.")
    print ("Can you guess it?")
    while (not done):
        guesses = guesses + 1
        guess = int(input ("guess? "))
        if (guess == num):
            done = True
        elif (guess < num):
            print ("Too low")
        else:
            print ("Too high")
    print ("Yea! It took you", guesses, "guesses.")
def main():
    goAgain="yes"
    while goAgain=="yes":
        game()
        goAgain=input("Play again (yes/no)? ")
        goAgain = goAgain.lower()

main()
```

Allowing Multiple Turns

Now that we have a working game, we can play all day and put your guessing skills to the test. Each time we want to play, though, we have to run the program again. In version 2 of the guessing game, we will allow the user to play as many games as they wish with just a single run of the program.

You might be thinking that all we need to do is enclose another loop around the code that plays the game. This will work, but our code is starting to get cluttered. A cleaner and more structured way to do this is to convert our game code into a function. In chapter 8, we talked about how many Python programmers will include their main program in a *main() function*. By doing this, we can separate the loop that allows multiple turns from the game itself. The only thing left to do is make a call to **main()**. Be careful that you do not indent this line since it will be the first instruction that will get executed. You can see the code listing of this version below. After

each game concludes, the user will now enter either "yes" or "no", depending on if they would like to play again or not.

INTERACTIVE – Playing the lottery

Suppose we wanted to generate a random 3-digit number. Take a look at the program below. That should do the trick! Make sure you examine the code closely and understand how it works. Also, run the program multiple times. Does it produce the same output each time? Now, see if you can modify the program so that the user will first input an integer, then the program will generate and print a random number with that many digits.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=40

Generating Random Strings

Often times, we would like to generate a random string. The **choice()** method will randomly pick an item from a list. To simulate flipping a coin three times, we create a list named **coins** with the strings **"*Heads*"** and **"*Tails*"**, and then call the **choice()** method passing **coins** as an argument.

```
>>> coins = ["Heads","Tails"]
>>> random.choice(coins)
'Heads'
>>> random.choice(coins)
'Tails'
>>> random.choice(coins)
'Tails'
>>>
```

Maybe you have a game that needs to pick a random superhero. You could store superheroes in a list and use the **choice()** method to randomly pick one. In the example below, we illustrate the **choice()** method by calling it three times:

```
>>> superHeroes = ["Batman","Superman","Spider-Man","Wonder Woman","Captain America"]
>>> random.choice(superHeroes)
'Batman'
>>> random.choice(superHeroes)
'Captain America'
>>> random.choice(superHeroes)
'Spider-Man'
>>>
```

The **sample()** method can be used to randomly pick **k** unique items from a list, where the list itself and the value of **k** are provided as parameters to the method. The **sample()** method returns a list, which can be stored as a variable. In the example below, we first show a list of **3** randomly chosen superheroes. We also call sample and store the list to a variable **heroes**. To access the individual items from the **heroes** list, you could use a **for** loop:

```
>>> superHeroes = ["Batman","Superman","Spider-Man","Wonder Woman","Captain America"]
>>> random.sample(superHeroes,3)
['Batman', 'Spider-Man', 'Superman']
>>>
>>> heroes = random.sample(superHeroes,3)
>>> for hero in heroes:
        print(hero)


Spider-Man
Captain America
Wonder Woman
>>>
```

Another method that comes in handy with programs that include

PRNG of lists is **shuffle()**. This method mixes up the order of the list items, similar to shuffling a deck of cards. Below, we print the **superHeroes** list, both before and after the **shuffle()** call:

```
>>> superHeroes = ["Batman","Superman","Spider-Man","Wonder Woman","Captain America"]
>>> superHeroes
['Batman', 'Superman', 'Spider-Man', 'Wonder Woman', 'Captain America']
>>> random.shuffle(superHeroes)
>>> superHeroes
['Wonder Woman', 'Captain America', 'Batman', 'Spider-Man', 'Superman']
>>>
```

The Paper-Rock-Scissors Game

Problem: Your friend Leo really loves to play the "Paper-Rock-Scissors" game, but he is currently isolated and has no opponent to play. Your job is to write a game so that Leo can play "Paper-Rock-Scissors." Specifically, you need to:

– Create a user-defined function named **game()** that will play the game:

- Your Python program should generate and store to a variable named **computer** either "paper", "rock", or "scissors".
- The program should ask the user to type in either "paper", "rock", or "scissors" as keyboard input and store it to a variable named **player**.
- The program should use **if** statements to output either "Computer wins", "Player wins", or "It was a tie".

– A **main()** function should be created that will call the game function. Once the game is over, the program should ask the user if they would like to play again (yes/no). The **main()** function should continue looping until the user says "no".

– The interaction of the program should be case-insensitive and look like this:

```
Welcome to the Paper Rocks Scissors Game

Do you want paper, rock, or scissors? Rock
The computer chose rock
You tie
Play again (yes/no)? Yes
Do you want paper, rock, or scissors? paper
The computer chose rock
You win!!
Play again (yes/no)? YES
Do you want paper, rock, or scissors? ROCK
The computer chose paper
You lose :(
Play again (yes/no)? yes
Do you want paper, rock, or scissors? Paper
The computer chose scissors
You lose :(
Play again (yes/no)? no
>>>
```

The code that implements this game is shown below. The first thing you notice is the **main()** function looks the same as the one from the Guess the Number program. The **lower()** method is used on both strings that the user inputs in order to make the logic case-insensitive. The **choice** method is used to randomly select the computer's picks. To check for a tie, we look to see if **player** and **computer** are the same. There are three ways a player can win this game, so a compound condition was created to test those three cases. Finally, we can use the **else** statement to conclude a loss since the player did not win or tie.

```
# Paper-Rock-Scissors Game
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX
import random
def game():
    options = ["paper","rock","scissors"]
    computer = random.choice(options)
    player = input("Do you want paper, rock, or scissors? ").lower()
    print("The computer chose",computer)
    if computer==player:
        print("You tie")
    elif (computer=="paper" and player=="scissors") or \
         (computer=="rock"  and player=="paper") or \
         (computer=="scissors" and player=="rock"):
        print("You win!!")
    else:
        print("You lose :(")


def main():
    print("Welcome to the Paper Rocks Scissors Game")
    print()
    goAgain="yes"
    while goAgain=="yes":
        game()
        goAgain=input("Play again (yes/no)? ")
        goAgain = goAgain.lower()

main()
```

Chapter Review Exercises:

11.1. What Python function could generate numbers from a random sequence such as 10, 15, 20...45, 50?

11.2. What Python function will select one item out of a list such as ["cat", "dog", "fish", "horse", "monkey", "snake"]?

11.3. What Python function will generate a random floating point number between 0 and 1?

11.4. What Python function will select multiple items out of a list such as ["cat", "dog", "fish", "horse", "monkey", "snake"]?

11.5. What Python function, when you give it minimum and maximum values like 3 and 10, will generate a random integer in that range [3..10]?

11.6. What Python function will take a list like ["cat", "dog", "fish", "horse", "monkey", "snake"] and turn it into a new list that looks like ["horse", "snake", "monkey", "cat", "fish", "dog"]?

12. Chapter 12: Turtle Graphics

TURTLE GRAPHICS
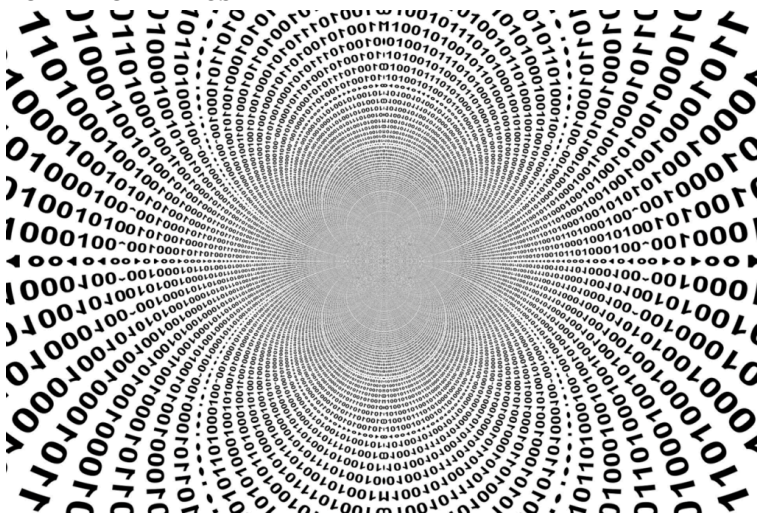
Topics Covered:

- Turtle graphics

Using **turtle graphics** is a fun way to hone your problem solving and programming skills, as well as a writing code that can generate graphics. It was part of the original Logo programming language developed in 1967.
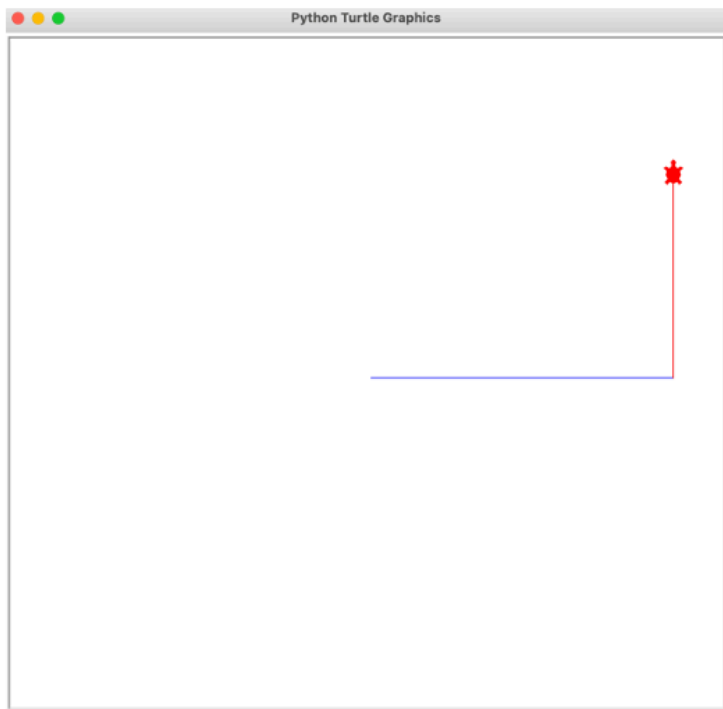
Importing the turtle module into your Python program allows you to create simple drawings on the screen. The name turtle, or turtle graphics, is a term in computing that means "using a relative cursor to draw on a Cartesian plane". The relative cursor is called the turtle. The Logo programming language compared the drawing capabilities

to that of a turtle, with a pen attached to its tail, crawling around on the screen. If the tail was down, a line was produced, but if the tail was raised, the movement of the turtle left no visible trace.

The graphic turtle begins at (0,0) in the x-y plane. By default, it is facing to the right. To begin drawing, you must first import the turtle module and then create a turtle object. The forward command will move the turtle a given number of pixels. Let's jump right to an example and start exploring the options and functions available.

```
import turtle
tom = turtle.Turtle()

tom.pencolor("blue")
tom.shape("turtle")
tom.forward(300)
tom.left(90)
tom.color("red")
tom.forward(200)
turtle.exitonclick()
```

When you run a turtle program, it will pop up a new "Python Turtle Graphics" window for your drawing. After importing the turtle module, you should create a turtle object by calling the **Turtle()** method. I named my turtle "Tom", but you can create the turtle object using any variable name.

In the sample program, the **pencolor()** method was used to change the turtle to **blue**. Next, the **shape** of the turtle was changed from a small arrowhead to an actual **turtle**. The **forward()** command will move the turtle straight ahead by a distance of the number of pixels passed as the argument. A *pixel*, short for picture element, is one of over a million little dots that make up your screen display. A blue line of 300 pixels was drawn left-to-right from the origin.

Tom the turtle turned **90** degrees to the **left()** and then changed its **color** to **red**. After the turn, Tom is now facing upwards and draws a vertical line of **200** pixels. The method **exitonclick()** will cause the window to disappear whenever the user clicks anywhere in the window.

In our next program, we will show two ways to draw a square. In the first example, Tina the turtle repeats the steps "move forward 200 pixels and turn 90 degrees left" four times. A second square, this one green, is created by using a **for** loop to repeat the **forward()** and **left()** methods.

```
import turtle
tina = turtle.Turtle()
tina.pensize(5)


tina.forward(200)
tina.left(90)
tina.forward(200)
tina.left(90)
tina.forward(200)
tina.left(90)
tina.forward(200)

tina.pencolor("green")
tina.penup()
tina.right(90)
tina.forward(100)
tina.pendown()

for count in range(0,4):
    tina.forward(250)
    tina.right(90)

turtle.exitonclick()
```
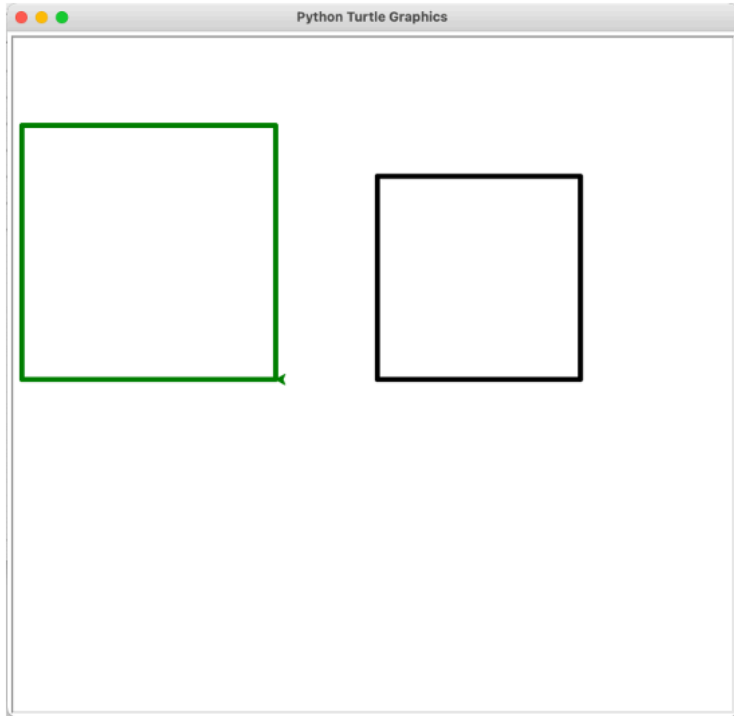
When the previous program is executed, it produces the window shown below. In order to avoid drawing a line that connected the two squares, the **penup()** method was used to move **100** pixels to the left of the black square. Before drawing the green square, the **pendown()** method was called to resume drawing. The output is shown next:

This third turtle graphics program introduces several new turtle methods:

```
import turtle
jill = turtle.Turtle()
jill.hideturtle()
jill.write("Hello")

jill.penup()
jill.goto(200,200)
jill.pencolor("red")
jill.write("Hooray",font=('Arial',24,"normal"))

jill.fillcolor("yellow")
jill.goto(-100,200)
jill.pendown()
jill.begin_fill()
jill.goto(-100,300)
jill.goto(-300,200)
jill.goto(-100,200)
jill.end_fill()

jill.penup()
jill.goto(100,0)
jill.pendown()
jill.dot()
jill.goto(200,0)
jill.dot()
jill.goto(300,0)
jill.dot()
turtle.exitonclick()
```

First, we show how the **hideturtle()** will make that shape invisible.
Next, we show how the **write()** method can be used to display a
string at the current turtle position. One optional parameter allows
you to change the **font**.

The **begin_fill()** and **end_fill()** methods can be used to change
the background color of a shape created using a concave set of
points. The **fillcolor()** method should be called to set the shape's
background color. You first use the **goto(x,y)** method to position
the turtle to the first point of your polygon. After that, a series
of **goto(x,y)** calls are used to create the shape. Once
the **end_fill()** method is called, the interior color of the shape is
filled.

Finally, we show how the **dot()** method can be used to draw a small
circle on the window. With the **pendown()** activated, this could be
used to create a line graph. With **penup()** active, you could use
the **dot()** method to create a scatter plot.
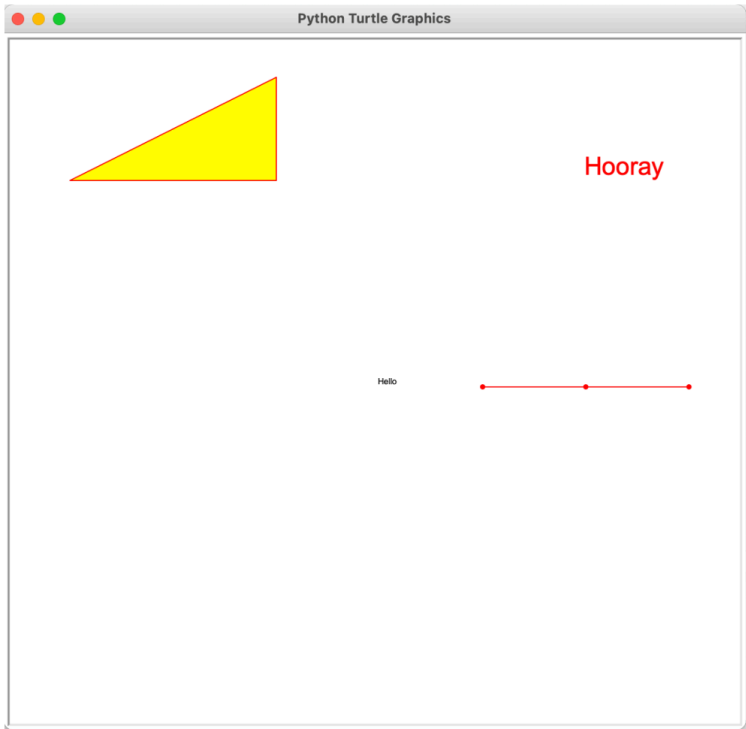
Common Turtle Graphics Methods

Method	Description	Examples
backward	Moves turtle backward by given pixels	`backwards(150)`
begin_fill	Turns the fill mode on for filling a shape	`begin_fill()`
dot	Places a small circle at current turtle position	`dot()`
end_fill	Turns the fill mode off for filling a shape	`end_fill()`
exitonclick	Causes window to close on a mouse click	`exitonclick()`
fillcolor	Changes the color of a shape's interior	`fillcolor("blue")` *See note below
forward	Moves turtle forward by given pixels	`forward(200)`
hideturtle	Hides the visibility of the turtle shape	`hideturtle()`
left	Turns turtle to the left by given degrees	`left(90)`
pencolor	Changes the color of lines and text	`pencolor("green")` *See note below
pendown	Turns drawing mode on for future moves	`pendown()`
pensize	Modifies the width of the pen by given pixels	`pensize(3)`
penup	Turns drawing mode off for future moves	`penup()`
right	Turns turtle to the right by given degrees	`right(45)`

shape	Changes the shape of the drawing turtle	`shape("arrow")`
		`shape("circle")`
		`shape("classic")`
		`shape("square")`
		`shape("triangle")`
		`shape("turtle")`

showturtle	Enables the turtle shape to be visible	`showturtle()`

write	Draws the given string at current position	`write("fun")`
		`write("Hooray",`
		`    font=('Arial',`
		`    24,"normal"))`

* Note: There are hundreds of different color names that you can use. Also, there are literally millions of colors that you can create by using RGB or hex codes. The following web site provides a great way to check out many of the available Turtle colors:

https://trinket.io/docs/colors

INTERACTIVE – What does the turtle say?

Check out the Turtle graphics program below, but do not run it. Try to figure out what the output will look like first. Then, go ahead and run it and verify if you are the turtle master or not.

> 📄 *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://una.pressbooks.pub/python-textbook/?p=42*

Chapter Review Exercises:

12.1. What is a pixel and approximately how many of them appear on a screen?

12.2. Explain what each of the following turtle graphic methods do?

- left
- forward
- goto
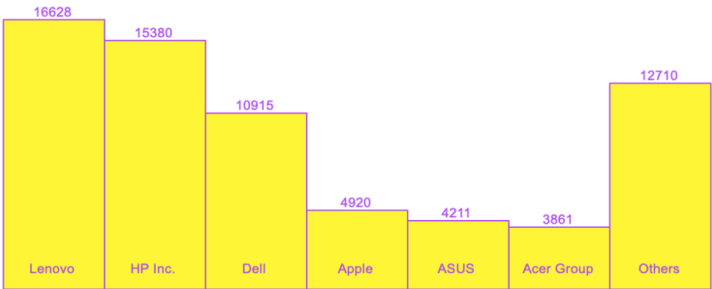- write
- penup
- endfill
- dot

Programming Projects:

12.1. Using Turtle Graphics, write a program that asks the user to type in the following inputs:

- Number of sides
- Length of each side
- Color of the sides
- The pen size

Your program should then draw a regular polygon with those properties.

12.2. Write a Python program to generate the graph below:

Worldwide PC Vendor Unit Shipment Estimates for 4Q18
(in thousands of units shipped)

13. Chapter 13: Graphical User Interfaces using Tkinter

GRAPHICAL USER INTERFACES
  USING TKINTER

Topics Covered:

- **Graphical user interface**
- **Event handlers**
- **Accessing the date and time**

Graphical User Interfaces

All of the programs we looked at so far used a text-based user interface (TUI). In this chapter, we learn how to develop ***grapical***

user interface (GUI) programs. In Python, a GUI program features a window full of widgets.

A *widget* is a visual component such as a label, button, entry, or image that you can place on the window. To create a GUI program, we will need to import *tkinter* (pronounced "t", "k", "inter"), which is a Python module that defines these GUI widgets.

Each widget is an object that has properties and responds to events, such as a key press or a mouse click. The widget properties can be assigned initial values when the program starts up. These properties can also be modified while the program is running.

To code a GUI program using TK, you typically follow these four steps:

No.	Step	Description
1	`from tkinter import *`	Import all of the widgets
2	`window = Tk()`	Create a TK object and name it; we'll use the name **window** for our examples
3	Add widgets and modify their properties/attributes	This can be done statically before the program starts or dynamically during program execution
4	`window.mainloop()`	This instruction acts like an infinite loop that continuously looks for events until you exit the program

Suppose you wanted to build the GUI window shown below. The window consists of three widgets. We will describe each of these widgets, as well as how we can add the cool title at the top of the window.

After importing the tkinter widgets and creating a Tk object named window, we use the **title()** method to place a caption at the top of

the window. Next, we create three widgets and place each of them on the screen.

A *button* is a rectangle that is usually associated with an event-handler function that gets triggered when the user clicks on the button. To create the button, we use the **Button()** constructor, or function, to initialize the button text, background color, and width. When calling this constructor, you need to assign it to a variable; in this case, we stored it as **myButton**.

To place a widget on the screen, you use the **grid()** method and provide arguments of the window object, as well as several optional arguments. In this instance, we padded 100 pixels to the left and right of the button and padded 20 pixels above and below the button.

A *label* widget simply places text on the window. Arguments to the **Label()** constructor include the window object and the **text** to display. In our example, we set the optional arguments to change the background and foreground colors of the label. We used the **grid()** method to add the label to the screen using a 20 pixel border on all four sides of the label.

An *entry* widget is a rectangle that allows the user to input text using the keyboard. The **Entry()** constructor requires the window object as a parameter. In our example, we set the width of the entry to **20** characters.

The **mainloop()** method will "listen" for user events like keys pressed and mouse clicks, but we haven't defined any event handlers yet. We will show that next. You can click on the button, but nothing will happen.

```
from tkinter import *
window = Tk()
window.title("This is a cool title")

myButton = Button(window,text="Hit this button!",bg="yellow",width=40)
myButton.grid(padx=100,pady=20)

myLabel = Label(window, text="Cool Label", bg="pink", fg="green")
myLabel.grid(padx=20, pady=20)

myEntry = Entry(window, width=20)
myEntry.grid(padx=0,pady=10)

window.mainloop()
```

Event Handlers

As mentioned, you can change a property of a widget during program execution. This is done by referencing the widget variable with the property to change enclosed in square brackets **[ ]**. We illustrate this process by creating an event handler that will toggle the background color of the button between yellow and purple.

To create an event handler, you must first define a function that will determine which code executes when the event is triggered, and then add the name of that function to the widget's constructor. In our example, we defined a function named **toggleBgColor()** and added the command property when creating the button. Notice how the parentheses are not included at the end of **command=toggleBgColor** in this instruction.

```
from tkinter import *

def toggleBgColor():
    if myButton["bg"]=="yellow":
        myButton["bg"]="purple"
    else:
        myButton["bg"]="yellow"

window = Tk()
window.title("This is a cool title")

myButton = Button(window,text="Hit this button!",bg="yellow",
                  command=toggleBgColor,width=40)
myButton.grid(padx=100,pady=20)

myLabel = Label(window, text="Cool Label", bg="pink", fg="green")
myLabel.grid(padx=20, pady=20)

myEntry = Entry(window, width=20)
myEntry.grid(padx=0,pady=10)

myButton["fg"]="white"
myLabel["text"]="Hooray"

window.mainloop()
```

In addition to demonstrating the button event handler, we included two other examples of how you can dynamically modify a widget property. We changed the foreground color of the button to **white** and modified the text of the label to "**Hooray**". When running this modified version of the program and clicking on the button one time, the window looks like this:
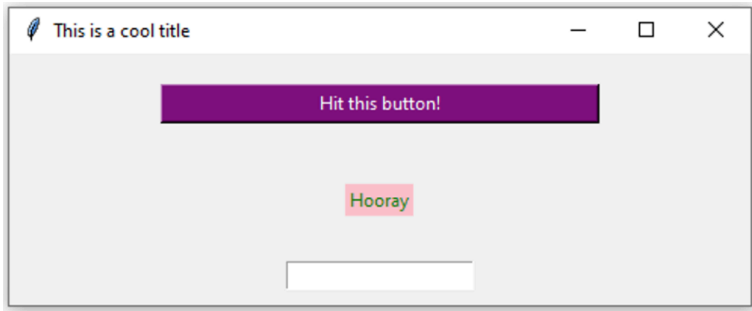
Problem: The Marketing Department at a local hardware store has asked you to develop a program for it. The managers are always asking about the ***percentage markup*** and the ***profit margin*** for various items in the inventory.

The program will require two inputs, the ***selling price*** and the ***purchase price*** of an item. The ***markup*** can be computed by subtracting the purchase price from the selling price. Other formulas needed are:

$$\text{percentage markup} = \frac{\text{markup}}{\text{purchase price}} \quad \text{and} \quad \text{profit margin} = \frac{\text{markup}}{\text{selling price}}$$

where these quotients are expressed as percentages. The user interface should look like this:

The solution to this problem is shown below. To retrieve data from the entries, we needed to create string variables and associate them with the entries. We used **"readonly"** entries to place the three outputs using the **set()** method of the string variables.

```
# Program to compute markup, percentage markup, and profit margin
# Written by XXX XXXXX
# CS101 - Introduction to Computer Programming
# Date: XX/XX/XXXX
from tkinter import *

def computeStuff():
    sellPrice = float(sellEntry.get())
    purchasePrice = float(purchaseEntry.get())
    markup=sellingPrice-purchasePrice
    markupString.set("$"+"%.2f"%markup)
    percentageMarkup = markup/purchasePrice*100
    percentageString.set("%.1f"%percentageMarkup+"%")
    profitMargin = markup/sellingPrice*100
    profitString.set("%.1f"%profitMargin+"%")

window = Tk()
window.title("Marketing Problem")

sellEntry = StringVar()
myLabel = Label(window, text="Selling Price: ", fg="blue", width=20)
myLabel.grid(row=0, column=0, padx=10, pady=10)
myEntry = Entry(window, width=20,textvariable=sellEntry)
myEntry.grid(row=0, column=1,padx=10)

purchaseEntry = StringVar()
myLabel2 = Label(window, text="Purchase Price", fg="blue")
myLabel2.grid(row=1, column=0, padx=10, pady=10)
myEntry2 = Entry(window, width=20,textvariable=purchaseEntry)
myEntry2.grid(row=1, column=1)


myButton = Button(window,text="Compute Stuff!", bg="blue", fg="white",
command=computeStuff)
myButton.grid(row=2, column=0, columnspan=2,padx=10, pady=10)

markupString = StringVar()
myLabel3 = Label(window, text="Markup: ",  fg="blue")
myLabel3.grid(row=3, column=0, padx=10, pady=10)
myEntry3 = Entry(window, width=20,state="readonly",textvariable=markupString)
myEntry3.grid(row=3, column=1)

percentageString = StringVar()
myLabel4 = Label(window, text="Percentage Markup: ",  fg="blue")
myLabel4.grid(row=4, column=0, padx=10, pady=10)
myEntry4 = Entry(window,
width=20,state="readonly",textvariable=percentageString)
myEntry4.grid(row=4, column=1)

profitString = StringVar()
myLabel5 = Label(window, text="Profit Margin: ",  fg="blue")
myLabel5.grid(row=5, column=0, padx=10, pady=10)
myEntry5 = Entry(window, width=20,state="readonly",textvariable=profitString)
myEntry5.grid(row=5, column=1)

window.mainloop()
```
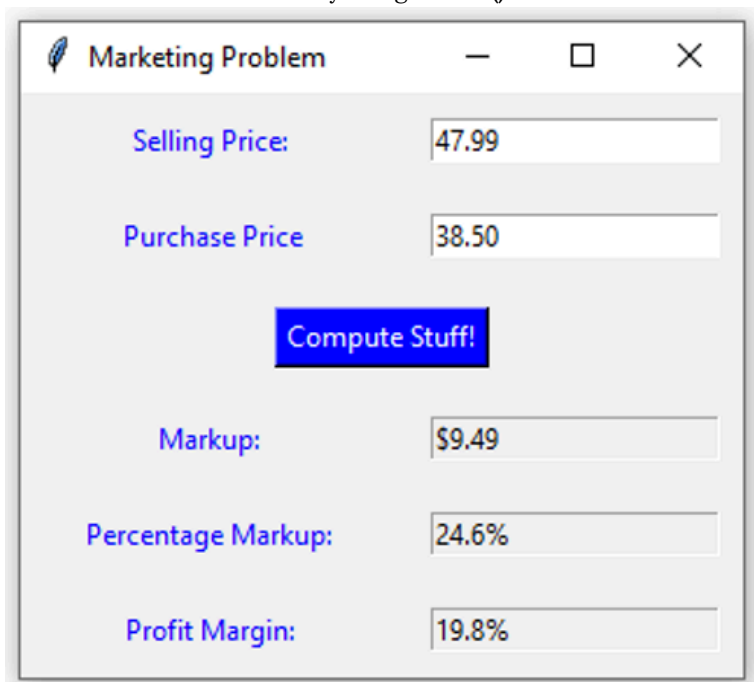
You will notice in this example that the placement of the widgets using the **grid()** method was done by using rows and columns. The top row is row 0 and the leftmost column is column 0. You can imagine the process as placing the widgets into a table or spreadsheet. You can also allow a widget to span across multiple columns as illustrated by the "Compute Stuff!" button in this program

Below is a sample run of the program with the store purchasing an item for $38.50 and selling it for $47.99. Notice how the three entry widgets that were defined to be **readonly** are "grayed out" on the screen. This indicates that the keyboard user isn't able to enter values into those fields, although the program is allowed to modify the text shown in the fields by using the **set()** method.

Accessing Date and Time

In applications that are being used to log logging activities, it is useful to access the current date or time. Python has a module called **datetime** that provides this capability. This module is quite versatile and provides numerous ways to format the current date and/or time. In the example below, we call the **today()** method and then print the current date in a MM-DD-YYYY format. We then use that same **thisDate** variable to print the current time in a HH:MM:SS

XM format. A great explanation with more examples can be viewed here:

https://www.w3schools.com/python/python_datetime.asp

```
>>> import datetime
>>> thisDate = datetime.datetime.today()
>>> thisDate.strftime('%m-%d-%Y')
'03-26-2021'
>>> thisDate.strftime("%I:%M:%S %p")
'12:02:31 PM'
>>>
```

Listbox Widget

The *listbox* widget is used to display a list of items from which a user can select any number of items. In the example below, we first create a list **colorList**, associate a string variable **colorString** with it, and then call the **Listbox()** constructor. We use the **set()** method to assign a value to **colorString**. The function **colorChange()** retrieves the selected item and we use that index to change color of the window background.

```
from tkinter import *
def colorChange():
    selected=myListBox.curselection()
    item = selected[0]
    window["bg"] = colorList[item]
window=Tk()
window.title("Fun with colors")
window.minsize(400,200)
window.configure(bg="white")
colorList = ["red","blue","yellow","orange","white"]
colorString = StringVar()
myListBox = Listbox(window,width=10,height=5,listvariable=colorString)
myListBox.grid(padx=20,pady=20)
colorString.set(colorList)
myListBox.select_set(0)
myButton = Button(window,text="Hit me",bg="light
blue",command=colorChange)
myButton.grid(padx=20,pady=20)
window.mainloop()
```

Sample program run:
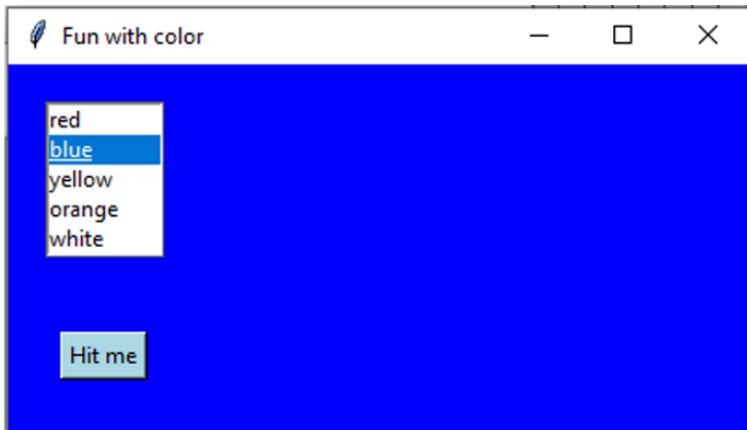
Radiobutton Widget

The *radiobutton* is a widget that is used to allow a user to select one out of a group of choices. On the next page, we present a program that serves as the cashier for the local bowling alley. It uses an entry for the games bowled and three radio buttons to choose the bowler type, one of child, adult, or senior. The **cashier()** function is called on a button click. It uses the **get()** method to determine the bowler type and the number of games. The label's **config()** method is used to change the label text based on the total cost. Notice how three radiobutton widgets all use the same integer variable **bowlerType** to ensure only one of them can be selected at any given time.

```
from tkinter import *
def cashier():
   bowler = bowlerType.get()
   if bowler ==1:
       cost=1.25
   elif bowler ==2:
       cost=3.50
   else:
       cost=2.00
   totalCost = cost * int(numberGames.get())
   selection = "Total Cost: $" + str("%.2f"%totalCost)
   costLabel.config(text = selection)

window = Tk()
window.title("North Alabama Bowling Lanes")
window.minsize(400,200)
theLabel= Label(window,text="Number of games?")
theLabel.grid(row=0,column=0)
numberGames=StringVar(value=1)
gamesEntry = Entry (window,textvariable=numberGames,width=4)
gamesEntry.grid(row=0,column=1)
bowlerType = IntVar()
childButton = Radiobutton(window, text="Child ($1.25)",
                             variable= bowlerType, value=1)
childButton.grid()
childButton.select()
adultButton = Radiobutton(window, text="Adult ($3.50)",
                             variable= bowlerType, value=2)
adultButton.grid()
seniorButton = Radiobutton(window, text="Senior ($2.00)",
                              variable= bowlerType, value=3)
seniorButton.grid()
totalButton = Button(window,text="ComputeTotal",
                       command=cashier,bg="yellow")
totalButton.grid()
costLabel = Label(window, text="Total Cost:")
costLabel.grid()
window.mainloop()
```
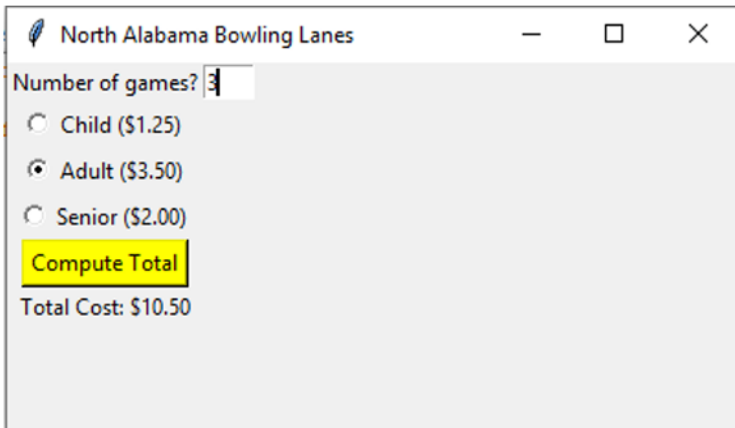
Sample program run:

Chapter Review Exercises:

13.1. What is a widget? Describe three common widgets.

Programming Projects:

13.1. Use the Python module *tkinter* to create the following graphical user interface. Do not worry about functionality at this point. You will add that in later. Be sure to modify your program so that it generates a match (title, colors, spacing, etc.) to this window:

- The default choices should be selected. That is, a burger with fries and a Coke.
- To allow for multiple list boxes to be selected, add the attribute for each one: **exportselection=0**
- Change the restaurant name to your own name. Notice the font size and bold face changes.
- Notice the Entry widget at the bottom is read only.
- Use the same colors as those shown above.

13.2. Add a function to your Project 13.1 program and connect it to the "Add to the order" button. When the user clicks this button, the function should add the prices of the selected entrees, sides, and drinks to the total and display that total (using a dollar sign and 2 decimal places) using the entry widget at the bottom of the screen.

13.3 Make the following changes to Project 13.2:

- Include a label and an entry widget that gets the customer's name
- When the user clicks the "Add to the order" button, print to the Interactive window a detailed receipt with labels (restaurant name, receipt message, current date, current time, customer name, entrée, side, drink, total)
- When the user clicks the "Add to the order" button, write the following data to a comma-delimited text file named "restaurantData.csv": current date, current time, customer name, entrée, side, drink, total

14. Chapter 14: Web Applications using Server-Side Scripting

WEB APPLICATIONS
   USING SERVER-SIDE SCRIPTING

Topics Covered:

- **HTML**
- **CSS**
- **Server-side scripting**

In chapter 13, we learned how to create a GUI to allow a more intuitive and natural interface for the user. In this chapter we will the web as the interface for our applications.

The *Internet* is a worldwide network of networks that was created in 1969. The **World Wide Web (WWW)**, developed in 1993, is a

popular application of the Internet. Others include email, file transfer, and networked computer games.

To access the WWW, all you need is an Internet connection and a *web browser*, which is a program that allows you to request and view web pages. Popular web browsers include Chrome, Edge, Firefox, and Safari.

When a user enters a web address, or *uniform resource locator (URL)*, it must first be converted to an *Internet Protocol (IP)* address, since that is the mechanism that Internet routers use to forward data traffic around the globe. A *domain name server (DNS)* is a computer that will provide that translation.

Once the request reaches the web server, it fetches the requested file from the server's hard disk, ships it back over the Internet, and the web browser *renders*, or displays the web page. This entire 7-step process is illustrated below:

Basic client/server request/response sequence
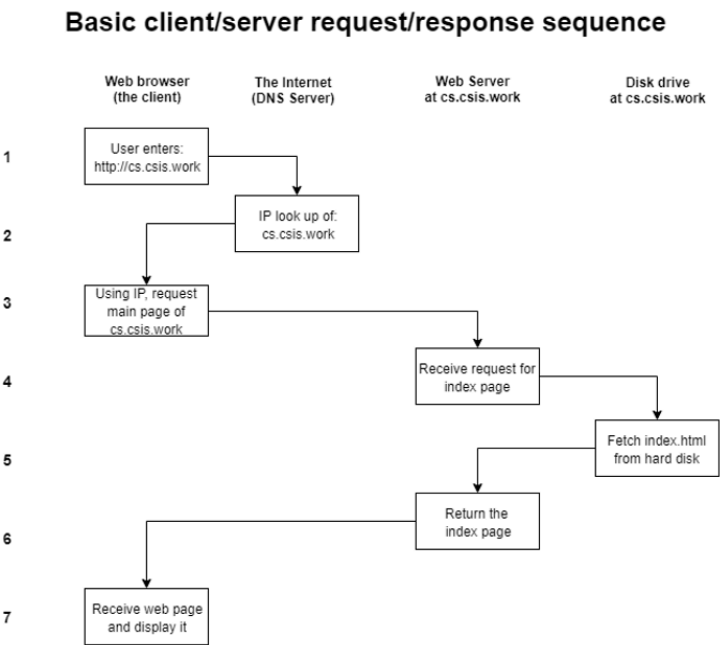
Web pages are simple text (ASCII) files that are embedded with

special tags using **HyperText Markup Language (HTML)**. The HTML tags provide a structure to help organize the web page into different sections and components. There are a lot of great free resources on the Internet to learn more about HTML. We recommend the following web site: https://www.w3schools.com/html

You can create web pages using a text editor on your own computer and view them locally. Once you are ready to share them with the world, you can **publish** your pages to a server for the world to see. More about that later. There are free text editors available, such as Notepad++, that provide color syntax highlighting to make creating and editing files much easier.

If you are interested in changing the formatting, such as spacing, fonts, and colors, it is recommended that you use a **cascading style sheet (CSS)**. A common naming scheme for many web servers is to name your initial web page as **index.html**. Here is a simple example that illustrates several common HTML tags:

```html
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <link rel="stylesheet" href="styles.css" type="text/css">
      <title>Mark Terwilliger's Web Page</title>
</head>

<body>
    <img src="peeps.jpg" alt="Me and my peeps" />

    <h1>Welcome to Mark Terwilliger's Very Dumb Web Page</h1>
    <p>This is my<br/> web page</p>
    <h2>Academics</h2>

    <p>I am a computer science major. I like computer programming.
    I am a student at the <a href="http://una.edu">University of
    North Alabama</a>. </p>

    <h2>Hobbies</h2>

    <ul>
          <li>Tennis</li>
          <li>Computers</li>
          <li>Taking Quizzes</li>
          <li>Golf</li>
    </ul>

</body>

</html>
```

Cascading Style Sheets (CSS)

A Cascading Style Sheet, or CSS file, is a special file that allows you to override the default meanings of the HTML tags that are used in your web page. While you can modify the meanings within your web page, it makes things a lot easier if you instead use a CSS. It not only removes excess clutter from your basic web page, but it also lets you change the appearance of your page without needing to modify the page contents. In fact, if you had hundreds, or even thousands, of web pages referencing the same CSS file, you can change the look of your entire web site by just modifying that one CSS file. Notice how a CSS file named **styles.css** is referenced in the <head> portion of the HTML file.

The <h1> tag creates a level one heading. By default, the web browser makes that heading black, bigger than the normal text, bold face, and puts it on a separate line. If you wanted to add other features to the <h1> tag, you can do that with CSS. In the example below, we make all h1 headings green.

```
h1    { color:green; }
body  { color:navy; background-color:yellow; }
img   { float:right; }
```

An image named **peeps.jpg** was also referenced in the web page. The HTML, CSS, and image files must all be located in the same folder for the page to render correctly. You can view a web page, as shown below, by opening it in your browser.

Server-Side Scripting

The WWW provides more than just the delivery of static files. Computer programs, often called **scripts**, can run on your computer (client-side) or on the web server (server-side) to make the web experience more dynamic or interactive. JavaScript is a popular client-side scripting language. On the server side, popular languages include PHP, Ruby, and NodeJS. Of course, Python is a popular choice as well and that is what we will demonstrate.

Before we can learn about writing scripts that process data coming from a web browser, we must first talk about how the user will input that data. A **web page form** consists of input components such as text boxes, check boxes, radio buttons, and drop-down menus.

In the example below, we will begin creating a tip calculator, which may be helpful when you are at a restaurant and trying to determine how much of a tip to leave for your wait person. The form will consist of three text boxes to gather input for: (1) the cost of the meal, (2) the tip percentage, and (3) the wait person's name.

At the beginning of the form, you must include the **action**, which is the name of the Python script that will receive and process this input data. In the example, we named the script as **processtips.py**. At the end of the form, we include a submit button, which will send the data to the server when the user clicks on the button.

```html
<!DOCTYPE html>
<html lang="en">
<head>
     <meta charset="UTF-8">
     <link rel="stylesheet" href="styles.css" type="text/css">
     <title>The Tip Calculator</title>
</head>

<body>

     <h1>Welcome to the Tip Calculator</h1>

     <form action="processtips.py" method="get">

     <p>Cost of meal: <input type="text" name="mealcost"> </p>
     <p>Tip percentage: <input type="text" name="tippercentage"> </p>
     <p>Name of waitperson: <input type="text" name="waitperson"> </p>
     <p><input type="submit" value="Compute the tip!"> </p>

     </form>

</body>
</html>
```
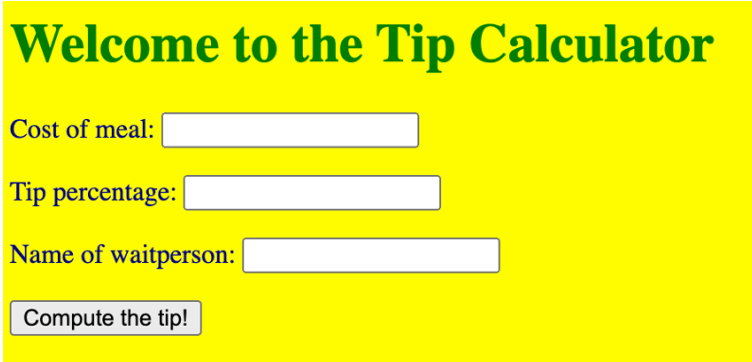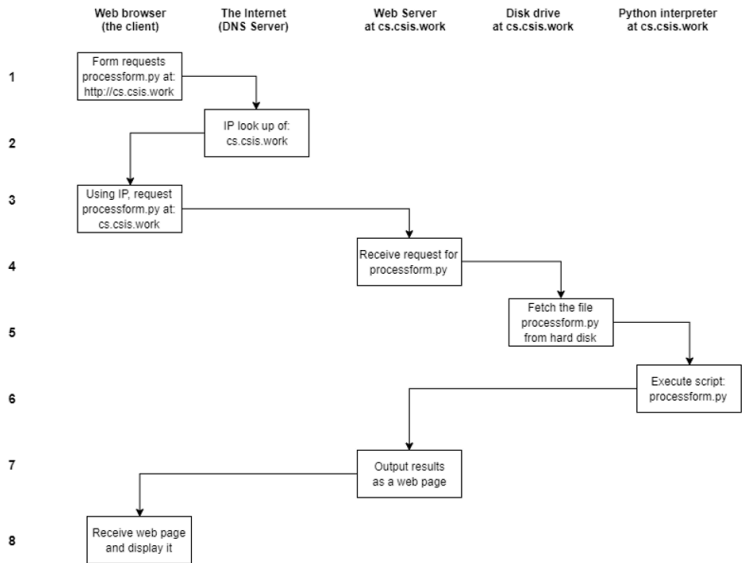
You will notice that the web page form has the same basic structure as your index page. In fact, in our form page, we even included the same CSS link. By doing that, our form page has the same color and style scheme as our index page:

Before we take a look at the server-side script, let's see how this new step changes the sequence of events in the client/server sequence. Now, the server has to send the form input to the Python script (i.e., processform.py), execute the script, and finally send the script output back to the client to be rendered by the web browser:

Dynamic client/server request/response sequence

	Web browser (the client)	The Internet (DNS Server)	Web Server at cs.csis.work	Disk drive at cs.csis.work	Python interpreter at cs.csis.work

1 Form requests processform.py at: http://cs.csis.work

2 IP look up of: cs.csis.work

3 Using IP, request processform.py at: cs.csis.work

4 Receive request for processform.py

5 Fetch the file processform.py from hard disk

6 Execute script: processform.py

7 Output results as a web page

8 Receive web page and display it

When writing the Python server-side script, there are several things we need to be aware of:

- The location of the Python interpreter must be included on the first line
- We need to import two modules, **cgi** and **cgitb**
- We use the getvalue() method to retrieve data from the web page form and store to variables. These names (**mealcost**, **tippercentage**, **waitperson**) must match exactly with the names used in the web page form
- We are actually writing a program to print a web page!
- In this example, we did some server-side data validation, which is a fancy way of saying we made sure the user didn't leave any fields blank

The **processtips.py** script is shown below:

```
#!/usr/local/bin/python

import cgi, cgitb
cgitb.enable()

form = cgi.FieldStorage()

meal    = form.getvalue('mealcost')
percent = form.getvalue('tippercentage')
waiter  = form.getvalue('waitperson')

print ("Content-Type: text/html")
print()
print("<!DOCTYPE html>")
print("<html lang='en'>")
print("<head>")
print("<link rel='stylesheet' href='styles.css'
type='text/css'>")
print("<title>Tip calculator</title>")
print("</head>")
print("<body>")
print("<h1>The tip calculator</h1>")

if (meal==None or percent==None or waiter==None):
    print("<p>You have left fields blank - hit Back and try
again</p>")
else:
    tipAmount = round(float(meal)*float(percent)/100, 2)
    print("<p>", waiter, "gets a tip of", tipAmount, "</p>")

print("</body>")
print("</html>")
```
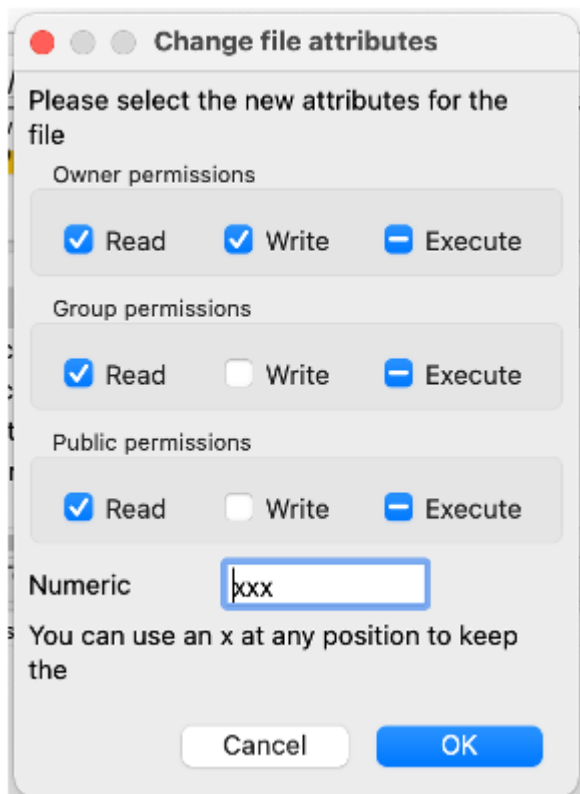
To test your server-side script, you will need to upload your files to a web server. To do this you need to use a **Secure File Transfer Protocol (SFTP)** client program. Although there are many free versions available, we will use FileZilla. When the program opens up, you need to enter the server domain name, the port number (22), and your username and password on the server.

Once you are successfully authenticated, the screen will look like that shown below. The files on the left side represent your local computer and on the right side displays your files on the server. To upload a file, you right-click on the local file and choose "Upload." Before uploading, you may have to change into a web page directory (i.e., **public_html**).
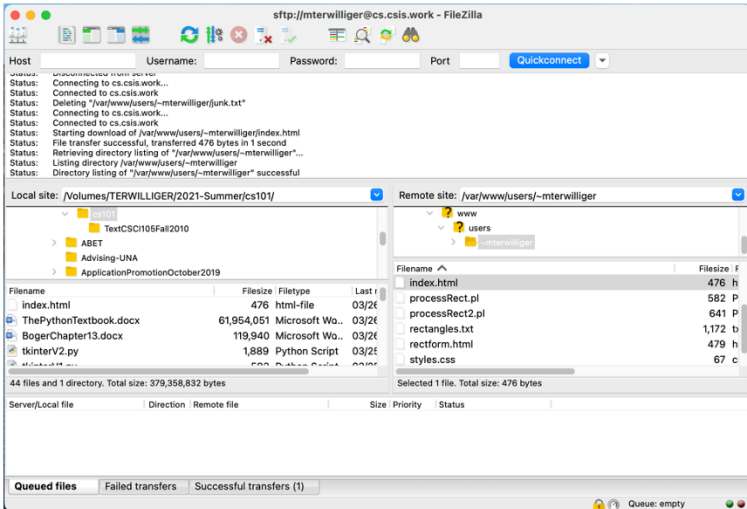
Finally, you need to change the permissions of your Python script so that anyone can run your application. To do this, you right-click on the script on the remote side, choose file permissions, and then check all the execute checkboxes as shown below:

Change file attributes

Please select the new attributes for the file

Owner permissions

☑ Read ☑ Write ⊟ Execute

Group permissions

☑ Read ☐ Write ⊟ Execute

Public permissions

☑ Read ☐ Write ⊟ Execute

Numeric [xxx]

You can use an x at any position to keep the

Cancel   OK

Chapter Review Exercises:

14.1. Define the following terms:

- Internet
- World Wide Web
- Web browser
- SFTP
- Web server
- HTML
- CSS

Programming Projects:

14.1. Create a web page about yourself that is a brief biographical sketch. Your biographical sketch should include information about your hobbies, interests, accomplishments, and affiliations (clubs, sororities, etc.) with appropriate hyperlinks. Hyperlinks could be to the web site of a club you belong to or a web page describing a hobby. You may link to social media sites where you have accounts (Facebook, Twitter, etc.) or other web pages about yourself. Feel free to talk about your major, courses in which you are enrolled, or about your computer/technology background. Save your file as **index.html**.

14.2. Create a style page named **styles.css** that defines at least 3 HTML tags. Add a link in **index.html** that references your new style page.

$$future = P\left(1 + \frac{r}{100}\right)^{n}$$

14.3. Create a web page named **fvform.html** that includes a form with the three inputs (principal, rate, years) as text boxes. Set the action to a "futurevalue.py" and set the method to "get". Also, make sure you add a submit button with an appropriate value for the caption. Include the link to your CSS page in the head section of this web page.

14.4. Create a CGI Python script named **futurevalue.py** that will read the three form values, compute the future value and total interest, and then print those to a web page with appropriate formatting and labels.